Refinement to CDCL with Watched Literals

Mathias Fleury, Jasmin Blanchette, and Peter Lammich

August 1, 2023

Contents

theory *WB-More-IICF-LLVM*
imports
Isabelle-LLVM.IICF
Isabelle-LLVM.Sepref-HOL-Bindings
More-Sepref.WB-More-Refinement
begin

This is not part of the multiset setup **lemma** *prod-assn-id-assn-destroy*:
fixes R :: ‹- ⇒ - ⇒ - ⇒ *bool*›
shows ‹R^d $*_a$ *id-assn*^{d} = (R $\times_a$ *id-assn*)^d›
apply (*auto simp*: *hfprod-def prod-assn-def*[*abs-def*] *invalid-assn-def pure-def intro*!: *ext*)
apply (*metis* (*full-types*) *pred-lift-extract-simps*(*2*) *pure-true-conv sep.add.right-neutral*)
by (*metis Sep-Algebra-Add.pure-part-pure pred-lift-extract-simps*(*1*) *pred-lift-extract-simps*(*2*)
pure-part-split-conj)

lemma *Exists-eq-simp*: ‹(∃ x. (P x ∧* ↑ (x = b)) s) ⟷ P b s›
apply (*auto*)
apply (*metis* (*full-types*) *Sep-Algebra-Add.pure-part-pure pure-partI pure-part-split-conj pure-true-conv sep.add.right-neutral*)
by (*metis* (*full-types*)*pure-true-conv sep-conj-sep-emptyE*)

lemma ‹(↑(x = b)) s ⟷ x = b ∧ □ s›
by (*auto simp*: *pred-lift-def*)
lemma *split-conj-is-pure*:
assumes ‹*Sepref-Basic.is-pure* B›
shows ‹(B b bi ∧* R) s = ((bi, b) ∈ *the-pure* B ∧ R s)› (**is** *?A*)
proof −
obtain B' **where** R: ‹B = *pure* B'› **using** *assms* **unfolding** *is-pure-conv* **..**
then have R': ‹B' = *the-pure* B› **by** *simp*
show A: *?A*
unfolding R'[*symmetric*] **unfolding** R *pure-def pred-lift-extract-simps*
by *auto*
qed
lemma *split-conj-is-pure2*:
assumes ‹*Sepref-Basic.is-pure* B›
shows
‹($R1$ ∧* B b bi ∧* R) s = ((bi, b) ∈ *the-pure* B ∧ ($R1$ ∧* R) s)› (**is** *?B*)
apply (*subst sep-algebra-class.sep-conj-left-commute*)
apply (*subst split-conj-is-pure*[*OF assms*])
apply *simp*
done

lemma *nempty-list-mset-rel-iff*: ‹$M \neq \{\#\} \Longrightarrow$
  $(xs, M) \in$ *list-mset-rel* $\longleftrightarrow$ $(xs \neq [] \wedge hd\ xs \in\# M \wedge$
    $(tl\ xs,$ *remove1-mset* $(hd\ xs)\ M) \in$ *list-mset-rel*)›
  by (*cases xs*) (*auto simp*: *list-mset-rel-def br-def dest*!: *multi-member-split*)

This function does not change the size of the underlying array.

definition *take1* **where**
  ‹*take1 xs* = *take 1 xs*›

The following two abbreviation are variants from $\lambda f.$ *uncurry2* (*uncurry2 f*) and $\lambda f.$ *uncurry2* (*uncurry2* (*uncurry2 f*)). The problem is that *uncurry2* (*uncurry2 f*) and *uncurry2* (*uncurry2 f*) are the same term, but only the latter is folded to $\lambda f.$ *uncurry2* (*uncurry2 f*).

abbreviation *uncurry4'* **where**
  uncurry4' f $\equiv$ *uncurry2* (*uncurry2 f*)

abbreviation *uncurry6'* **where**
  uncurry6' f $\equiv$ *uncurry2* (*uncurry4' f*)

lemma *hrp-comp-Id2*[*simp*]: ‹*hrp-comp A Id* = *A*›
  unfolding *hrp-comp-def* **by** *auto*

lemma *list-rel-update*:
  assumes *rel*: ‹$(xs, ys) \in \langle$*the-pure R*$\rangle$*list-rel*› **and**
   h: ‹*R b bi s*› **and**
   p: ‹*is-pure R*›
  shows ‹(*list-update xs ba bi*, *list-update ys ba b*) $\in \langle$*the-pure R*$\rangle$*list-rel*›
proof −
  obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
    using *p* **by** *fastforce*
  have [*simp*]: ‹(*bi*, *b*) $\in$ *the-pure R*›
    using *h p* **by** (*auto simp*: *R R' pure-app-eq pred-lift-extract-simps*)
  have ‹*length xs* = *length ys*›
    using *assms list-rel-imp-same-length* **by** *blast*

  then show *?thesis*
    using *rel*
    by (*induction xs ys arbitrary*: *ba rule*: *list-induct2*) (*auto split*: *nat.splits*)
qed

end

theory *WB-Sort*
 imports *More-Sepref.WB-More-Refinement More-Sepref.WB-More-Refinement-List HOL−Library.Rewrite*
begin

Chapter 1

Sorting

The correctness proof is due to Maximilian Wuttke.

Remark that a more efficient can be found as part of Isabelle_LLVM. It has a few advantages. First, it does fewer recursive calls (which is an issue in the SAT solver). Second, it is more general (see the examples of string sorting that requires ownership knowledge). Finally, it is as fast as C++ sorting.

The implemented sorting is quicksort.

Every element between *lo* and *hi* can be chosen as pivot element.

definition *choose-pivot* :: ‹('b ⇒ 'b ⇒ *bool*) ⇒ ('a ⇒ 'b) ⇒ 'a *list* ⇒ *nat* ⇒ *nat* ⇒ *nat nres*› **where**
‹*choose-pivot* - - - *lo hi* = *SPEC*(λ*k*. *k* ≥ *lo* ∧ *k* ≤ *hi*)›

The element at index *p* partitions the subarray *lo..hi*. This means that every element

definition *isPartition-wrt* :: ‹('b ⇒ 'b ⇒ *bool*) ⇒ 'b *list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*isPartition-wrt R xs lo hi p* ≡ (∀ *i*. *i* ≥ *lo* ∧ *i* < *p* ⟶ *R* (*xs*!*i*) (*xs*!*p*)) ∧ (∀ *j*. *j* > *p* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*p*) (*xs*!*j*))›

lemma *isPartition-wrtI*:
‹(⋀ *i*. ⟦*i* ≥ *lo*; *i* < *p*⟧ ⟹ *R* (*xs*!*i*) (*xs*!*p*)) ⟹ (⋀ *j*. ⟦*j* > *p*; *j* ≤ *hi*⟧ ⟹ *R* (*xs*!*p*) (*xs*!*j*)) ⟹ *isPartition-wrt R xs lo hi p*›
by (*simp add*: *isPartition-wrt-def*)

definition *isPartition* :: ‹'*a* :: *order list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*isPartition xs lo hi p* ≡ *isPartition-wrt* (≤) *xs lo hi p*›

abbreviation *isPartition-map* :: ‹('b ⇒ 'b ⇒ *bool*) ⇒ ('a ⇒ 'b) ⇒ 'a *list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*isPartition-map R h xs i j k* ≡ *isPartition-wrt* (λ*a b*. *R* (*h a*) (*h b*)) *xs i j k*›

lemma *isPartition-map-def'*:
‹*lo* ≤ *p* ⟹ *p* ≤ *hi* ⟹ *hi* < *length xs* ⟹ *isPartition-map R h xs lo hi p* = *isPartition-wrt R* (*map h xs*) *lo hi p*›
by (*auto simp add*: *isPartition-wrt-def conjI*)

Example: 6 is the pivot element (with index 4); 7::'*a* is equal to the *length xs* − *1*.

lemma ‹*isPartition* [*0*,*5*,*3*,*4*,*6*,*9*,*8*,*10*::*nat*] *0 7 4*›
by (*auto simp add*: *isPartition-def isPartition-wrt-def nth-Cons'*)

definition *sublist* :: ‹*'a list* ⇒ *nat* ⇒ *nat* ⇒ *'a list*› **where**
‹*sublist xs i j* ≡ *take* (*Suc j* − *i*) (*drop i xs*)›

lemma *take-Suc0*:
l≠[] ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
0 < *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
Suc n ≤ *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
by (*cases l*, *auto*)+

lemma *sublist-single*: ‹*i* < *length xs* ⟹ *sublist xs i i* = [*xs*!*i*]›
by (*cases xs*) (*auto simp add*: *sublist-def take-Suc0*)

lemma *insert-eq*: ‹*insert a b* = *b* ∪ {*a*}›
by *auto*

lemma *sublist-nth*: ‹⟦*lo* ≤ *hi*; *hi* < *length xs*; *k*+*lo* ≤ *hi*⟧ ⟹ (*sublist xs lo hi*)!*k* = *xs*!(*lo*+*k*)›
by (*simp add*: *sublist-def*)

lemma *sublist-length*: ‹⟦*i* ≤ *j*; *j* < *length xs*⟧ ⟹ *length* (*sublist xs i j*) = *1* + *j* − *i*›
by (*simp add*: *sublist-def*)

lemma *sublist-not-empty*: ‹⟦*i* ≤ *j*; *j* < *length xs*; *xs* ≠ []⟧ ⟹ *sublist xs i j* ≠ []›
apply *simp*
apply (*rewrite List.length-greater-0-conv*[*symmetric*])
apply (*rewrite sublist-length*)
by *auto*

lemma *sublist-app*: ‹⟦*i1* ≤ *i2*; *i2* ≤ *i3*⟧ ⟹ *sublist xs i1 i2* @ *sublist xs* (*Suc i2*) *i3* = *sublist xs i1 i3*›
unfolding *sublist-def*
by (*smt Suc-eq-plus1-left Suc-le-mono append.assoc le-SucI le-add-diff-inverse le-trans same-append-eq take-add*)

definition *sorted-sublist-wrt* :: ‹(*'b* ⇒ *'b* ⇒ *bool*) ⇒ *'b list* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*sorted-sublist-wrt R xs lo hi* = *sorted-wrt R* (*sublist xs lo hi*)›

definition *sorted-sublist* :: ‹*'a* :: *linorder list* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*sorted-sublist xs lo hi* = *sorted-sublist-wrt* (≤) *xs lo hi*›

abbreviation *sorted-sublist-map* :: ‹(*'b* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *nat* ⇒ *nat* ⇒ *bool*›
where
‹*sorted-sublist-map R h xs lo hi* ≡ *sorted-sublist-wrt* (λ*a b*. *R* (*h a*) (*h b*)) *xs lo hi*›

lemma *sorted-sublist-map-def'*:
‹*lo* < *length xs* ⟹ *sorted-sublist-map R h xs lo hi* ≡ *sorted-sublist-wrt R* (*map h xs*) *lo hi*›
apply (*simp add*: *sorted-sublist-wrt-def*)
by (*simp add*: *drop-map sorted-wrt-map sublist-def take-map*)

lemma *sorted-sublist-wrt-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist-wrt R xs i i*›
by (*auto simp add*: *sorted-sublist-wrt-def sublist-single*)

lemma *sorted-sublist-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist xs i i*›
by (*auto simp add*: *sorted-sublist-def sorted-sublist-wrt-refl*)

lemma *sorted-sublist-map-refl*: ‹$i < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ i\ i$›
by (*auto simp add*: *sorted-sublist-wrt-refl*)

lemma *sublist-map*: ‹$sublist\ (map\ f\ xs)\ i\ j = map\ f\ (sublist\ xs\ i\ j)$›
apply (*auto simp add*: *sublist-def*)
by (*simp add*: *drop-map take-map*)

lemma *take-set*: ‹$j \leq length\ xs \Longrightarrow x \in set\ (take\ j\ xs) \equiv (\exists\ k.\ k < j \wedge xs!k = x)$›
apply (*induction xs*)
apply *simp*
by (*meson in-set-conv-iff less-le-trans*)

lemma *drop-set*: ‹$j \leq length\ xs \Longrightarrow x \in set\ (drop\ j\ xs) \equiv (\exists k.\ j \leq k \wedge k < length\ xs \wedge xs!k = x)$›
by (*smt Misc.in-set-drop-conv-nth*)

lemma *sublist-el*: ‹$i \leq j \Longrightarrow j < length\ xs \Longrightarrow x \in set\ (sublist\ xs\ i\ j) \equiv (\exists\ k.\ k < Suc\ j - i \wedge xs!(i+k) = x)$›
apply (*simp add*: *sublist-def*)
by (*auto simp add*: *take-set*)

lemma *sublist-el'*: ‹$i \leq j \Longrightarrow j < length\ xs \Longrightarrow x \in set\ (sublist\ xs\ i\ j) \equiv (\exists\ k.\ i \leq k \wedge k \leq j \wedge xs!k = x)$›
apply (*simp add*: *sublist-el*)
by (*smt Groups.add-ac(2) le-add1 le-add-diff-inverse less-Suc-eq less-diff-conv nat-less-le order-refl*)

lemma *sublist-lt*: ‹$hi < lo \Longrightarrow sublist\ xs\ lo\ hi = []$›
by (*auto simp add*: *sublist-def*)

lemma *nat-le-eq-or-lt*: ‹$(a :: nat) \leq b = (a = b \vee a < b)$›
by *linarith*

lemma *sorted-sublist-wrt-le*: ‹$hi \leq lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
apply (*auto simp add*: *nat-le-eq-or-lt*)
unfolding *sorted-sublist-wrt-def*
subgoal apply (*rewrite sublist-single*) **by** *auto*
subgoal by (*auto simp add*: *sublist-lt*)
done

Elements in a sorted sublists are actually sorted

lemma *sorted-sublist-wrt-nth-le*:
assumes ‹*sorted-sublist-wrt R xs lo hi*› **and** ‹$lo \leq hi$› **and** ‹$hi < length\ xs$› **and**
‹$lo \leq i$› **and** ‹$i < j$› **and** ‹$j \leq hi$›
shows ‹$R\ (xs!i)\ (xs!j)$›
proof −
have *A*: ‹$lo < length\ xs$› **using** *assms(2) assms(3)* **by** *linarith*
obtain i' **where** *I*: ‹$i = lo + i'$› **using** *assms(4) le-Suc-ex* **by** *auto*
obtain j' **where** *J*: ‹$j = lo + j'$› **by** (*meson assms(4) assms(5) dual-order.trans le-iff-add less-imp-le-nat*)
show *?thesis*
using *assms(1)* **apply** (*simp add*: *sorted-sublist-wrt-def I J*)
apply (*rewrite sublist-nth*[*symmetric*, **where** $k=i'$, **where** *lo*=*lo*, **where** *hi*=*hi*])
using *assms* **apply** *auto* **subgoal using** *I* **by** *linarith*
apply (*rewrite sublist-nth*[*symmetric*, **where** $k=j'$, **where** *lo*=*lo*, **where** *hi*=*hi*])

using *assms* **apply** *auto* **subgoal using** *J* **by** *linarith*
apply (*rule sorted-wrt-nth-less*)
apply *auto*
subgoal using *I J nat-add-left-cancel-less* **by** *blast*
subgoal apply (*simp add*: *sublist-length*) **using** *J* **by** *linarith*
done
qed

We can make the assumption $i < j$ weaker if we have a reflexivie relation.

lemma *sorted-sublist-wrt-nth-le'*:
assumes *ref*: ‹$\bigwedge$ *x*. *R x x*›
and ‹*sorted-sublist-wrt R xs lo hi*› **and** ‹*lo* ≤ *hi*› **and** ‹*hi* < *length xs*›
and ‹*lo* ≤ *i*› **and** ‹*i* ≤ *j*› **and** ‹*j* ≤ *hi*›
shows ‹*R* (*xs*!*i*) (*xs*!*j*)›
proof −
have ‹*i* < *j* ∨ *i* = *j*› **using** ‹*i* ≤ *j*› **by** *linarith*
then consider (*a*) ‹*i* < *j*› |
(*b*) ‹*i* = *j*› **by** *blast*
then show *?thesis*
proof *cases*
case *a*
then show *?thesis*
using *assms*(*2*−*5*,*7*) *sorted-sublist-wrt-nth-le* **by** *blast*
next
case *b*
then show *?thesis*
by (*simp add*: *ref*)
qed
qed

lemma *sorted-sublist-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist xs lo hi*›
by (*auto simp add*: *sorted-sublist-def sorted-sublist-wrt-le*)

lemma *sorted-sublist-map-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist-map R h xs lo hi*›
by (*auto simp add*: *sorted-sublist-wrt-le*)

lemma *sublist-cons*: ‹*lo* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo hi* = *xs*!*lo* # *sublist xs* (*Suc lo*) *hi*›
apply (*simp add*: *sublist-def*)
by (*metis Cons-nth-drop-Suc Suc-diff-le le-trans less-imp-le-nat not-le take-Suc-Cons*)

lemma *sorted-sublist-wrt-cons'*:
‹*sorted-sublist-wrt R xs* (*lo*+*1*) *hi* ⟹ *lo* ≤ *hi* ⟹ *hi* < *length xs* ⟹ (∀ *j*. *lo*<*j*∧*j*≤*hi* ⟶ *R* (*xs*!*lo*) (*xs*!*j*)) ⟹ *sorted-sublist-wrt R xs lo hi*›
apply (*simp add*: *sorted-sublist-wrt-def*)
apply (*auto simp add*: *nat-le-eq-or-lt*)
subgoal by (*simp add*: *sublist-single*)
apply (*auto simp add*: *sublist-cons sublist-el*)
by (*metis Suc-lessI ab-semigroup-add-class.add.commute less-add-Suc1 less-diff-conv*)

lemma *sorted-sublist-wrt-cons*:

assumes *trans*: ‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*)› **and**
‹*sorted-sublist-wrt R xs* (*lo*+*1*) *hi*› **and**
‹*lo* ≤ *hi*› **and** ‹*hi* < *length xs* › **and** ‹*R* (*xs*!*lo*) (*xs*!(*lo*+*1*))›
shows ‹*sorted-sublist-wrt R xs lo hi*›
proof −
show *?thesis*
apply (*rule sorted-sublist-wrt-cons'*) **using** *assms* **apply** *auto*
subgoal premises *assms'* **for** *j*
proof −
have *A*: ‹*j*=*lo*+*1* ∨ *j*>*lo*+*1*› **using** *assms'*(*5*) **by** *linarith*
show *?thesis*
using *A* **proof**
assume *A*: ‹*j*=*lo*+*1*› **show** *?thesis*
by (*simp add*: *A assms'*)
next
assume *A*: ‹*j*>*lo*+*1*› **show** *?thesis*
apply (*rule trans*)
apply (*rule assms*(*5*))
apply (*rule sorted-sublist-wrt-nth-le*[*OF assms*(*2*), **where** *i*=‹*lo*+*1*›, **where** *j*=*j*])
subgoal using *A assms'*(*6*) **by** *linarith*
subgoal using *assms'*(*3*) *less-imp-diff-less* **by** *blast*
subgoal using *assms'*(*5*) **by** *auto*
subgoal using *A* **by** *linarith*
subgoal by (*simp add*: *assms'*(*6*))
done
qed
qed
done
qed

lemma *sorted-sublist-map-cons*:
‹(⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)) ⟹
sorted-sublist-map R h xs (*lo*+*1*) *hi* ⟹ *lo* ≤ *hi* ⟹ *hi* < *length xs* ⟹ *R* (*h* (*xs*!*lo*)) (*h* (*xs*!(*lo*+*1*)))
⟹ *sorted-sublist-map R h xs lo hi*›
by (*blast intro*: *sorted-sublist-wrt-cons*)

lemma *sublist-snoc*: ‹*lo* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo hi* = *sublist xs lo* (*hi*−*1*) @ [*xs*!*hi*]›
apply (*simp add*: *sublist-def*)
proof −
assume *a1*: *lo* < *hi*
assume *hi* < *length xs*
then have *take lo xs* @ *take* (*Suc hi* − *lo*) (*drop lo xs*) = (*take lo xs* @ *take* (*hi* − *lo*) (*drop lo xs*)) @ [*xs* ! *hi*]
using *a1* **by** (*metis* (*no-types*) *Suc-diff-le add-Suc-right hd-drop-conv-nth le-add-diff-inverse less-imp-le-nat take-add take-hd-drop*)
then show *take* (*Suc hi* − *lo*) (*drop lo xs*) = *take* (*hi* − *lo*) (*drop lo xs*) @ [*xs* ! *hi*]
by *simp*
qed

lemma *sorted-sublist-wrt-snoc'*:
‹*sorted-sublist-wrt R xs lo* (*hi*−*1*) ⟹ *lo* ≤ *hi* ⟹ *hi* < *length xs* ⟹ (∀ *j*. *lo*≤*j*∧*j*<*hi* ⟶ *R* (*xs*!*j*) (*xs*!*hi*)) ⟹ *sorted-sublist-wrt R xs lo hi*›
apply (*simp add*: *sorted-sublist-wrt-def*)
apply (*auto simp add*: *nat-le-eq-or-lt*)
subgoal by (*simp add*: *sublist-single*)

apply (*auto simp add*: *sublist-snoc sublist-el sorted-wrt-append ac-simps*)
by (*metis add.commute add-cancel-right-right less-add-same-cancel1 less-diff-conv not-gr-zero*)

lemma *sorted-sublist-wrt-snoc*:
assumes *trans*: ‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*)› **and**
‹*sorted-sublist-wrt R xs lo* (*hi*−*1*)› **and**
‹*lo* ≤ *hi*› **and** ‹*hi* < *length xs*› **and** ‹(*R* (*xs*!(*hi*−*1*)) (*xs*!*hi*))›
shows ‹*sorted-sublist-wrt R xs lo hi*›
proof −
show *?thesis*
apply (*rule sorted-sublist-wrt-snoc'*) **using** *assms* **apply** *auto*
subgoal premises *assms'* **for** *j*
proof −
have *A*: ‹*j*=*hi*−*1* ∨ *j*<*hi*−*1*› **using** *assms'*(*6*) **by** *linarith*
show *?thesis*
using *A* **proof**
assume *A*: ‹*j*=*hi*−*1*› **show** *?thesis*
by (*simp add*: *A assms'*)
next
assume *A*: ‹*j*<*hi*−*1*› **show** *?thesis*
apply (*rule trans*)
apply (*rule sorted-sublist-wrt-nth-le*[*OF assms*(*2*), **where** *i*=*j*, **where** *j*=‹*hi*−*1*›])
prefer *6*
apply (*rule assms*(*5*))
apply *auto*
subgoal using *A assms'*(*5*) **by** *linarith*
subgoal using *assms'*(*3*) *less-imp-diff-less* **by** *blast*
subgoal using *assms'*(*5*) **by** *auto*
subgoal using *A* **by** *linarith*
done
qed
qed
done
qed

lemma *sorted-sublist-map-snoc*:
‹(⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)) ⟹
sorted-sublist-map R h xs lo (*hi*−*1*) ⟹
lo ≤ *hi* ⟹ *hi* < *length xs* ⟹ (*R* (*h* (*xs*!(*hi*−*1*))) (*h* (*xs*!*hi*))) ⟹ *sorted-sublist-map R h xs lo hi*›
by (*blast intro*: *sorted-sublist-wrt-snoc*)

lemma *sublist-split*: ‹*lo* ≤ *hi* ⟹ *lo* < *p* ⟹ *p* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo p* @ *sublist xs* (*p*+*1*) *hi* = *sublist xs lo hi*›
apply (*auto simp add*: *sublist-def*)
by (*smt Suc-leI append-assoc append-eq-append-conv diff-Suc-Suc drop-take-drop-drop le-SucI le-trans nat-less-le*)

lemma *sublist-split-part*: ‹*lo* ≤ *hi* ⟹ *lo* < *p* ⟹ *p* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo* (*p*−*1*) @ *xs*!*p* # *sublist xs* (*p*+*1*) *hi* = *sublist xs lo hi*›
apply (*auto simp add*: *sublist-split*[*symmetric*])
apply (*rewrite sublist-snoc*[**where** *xs*=*xs*,**where** *lo*=*lo*,**where** *hi*=*p*])
by *auto*

A property for partitions (we always assume that R is transitive.

lemma *isPartition-wrt-trans*:
‹($\bigwedge$ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*) ⟹
isPartition-wrt R xs lo hi p ⟹
(∀ *i j*. *lo* ≤ *i* ∧ *i* < *p* ∧ *p* < *j* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*i*) (*xs*!*j*))›
by (*auto simp add*: *isPartition-wrt-def*)

lemma *isPartition-map-trans*:
‹($\bigwedge$ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)) ⟹
hi < *length xs* ⟹
isPartition-map R h xs lo hi p ⟹
(∀ *i j*. *lo* ≤ *i* ∧ *i* < *p* ∧ *p* < *j* ∧ *j* ≤ *hi* ⟶ *R* (*h* (*xs*!*i*)) (*h* (*xs*!*j*)))›
by (*auto simp add*: *isPartition-wrt-def*)

lemma *merge-sorted-wrt-partitions-between′*:
‹*lo* ≤ *hi* ⟹ *lo* < *p* ⟹ *p* < *hi* ⟹ *hi* < *length xs* ⟹
isPartition-wrt R xs lo hi p ⟹
sorted-sublist-wrt R xs lo (*p*−1) ⟹ *sorted-sublist-wrt R xs* (*p*+1) *hi* ⟹
(∀ *i j*. *lo* ≤ *i* ∧ *i* < *p* ∧ *p* < *j* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*i*) (*xs*!*j*)) ⟹
sorted-sublist-wrt R xs lo hi›
apply (*auto simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-def sorted-sublist-wrt-def sublist-map*)
apply (*simp add*: *sublist-split-part*[*symmetric*])
apply (*auto simp add*: *List.sorted-wrt-append*)
subgoal by (*auto simp add*: *sublist-el*)
subgoal by (*auto simp add*: *sublist-el*)
subgoal by (*auto simp add*: *sublist-el′*)
done

lemma *merge-sorted-wrt-partitions-between*:
‹($\bigwedge$ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*) ⟹
isPartition-wrt R xs lo hi p ⟹
sorted-sublist-wrt R xs lo (*p*−1) ⟹ *sorted-sublist-wrt R xs* (*p*+1) *hi* ⟹
lo ≤ *hi* ⟹ *hi* < *length xs* ⟹ *lo* < *p* ⟹ *p* < *hi* ⟹ *hi* < *length xs* ⟹
sorted-sublist-wrt R xs lo hi›
by (*simp add*: *merge-sorted-wrt-partitions-between′ isPartition-wrt-trans*)

The main theorem to merge sorted lists

lemma *merge-sorted-wrt-partitions*:
‹*isPartition-wrt R xs lo hi p* ⟹
sorted-sublist-wrt R xs lo (*p* − *Suc 0*) ⟹ *sorted-sublist-wrt R xs* (*Suc p*) *hi* ⟹
lo ≤ *hi* ⟹ *lo* ≤ *p* ⟹ *p* ≤ *hi* ⟹ *hi* < *length xs* ⟹
(∀ *i j*. *lo* ≤ *i* ∧ *i* < *p* ∧ *p* < *j* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*i*) (*xs*!*j*)) ⟹
sorted-sublist-wrt R xs lo hi›
subgoal premises *assms*
proof −
have *C*: ‹*lo*=*p*∧*p*=*hi* ∨ *lo*=*p*∧*p*<*hi* ∨ *lo*<*p*∧*p*=*hi* ∨ *lo*<*p*∧*p*<*hi*›
using *assms* **by** *linarith*
show *?thesis*
using *C* **apply** *auto*
subgoal — lo=p=hi
apply (*rule sorted-sublist-wrt-refl*)
using *assms* **by** *auto*
subgoal — lo=p<hi

using *assms* **by** (*simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-wrt-cons'*)
subgoal — lo<p=hi
using *assms* **by** (*simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-wrt-snoc'*)
subgoal — lo<p<hi
using *assms*
apply (*rewrite merge-sorted-wrt-partitions-between'*[**where** *p*=*p*])
by *auto*
done
qed
done

theorem *merge-sorted-map-partitions*:
‹(⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)) ⟹
isPartition-map R h xs lo hi p ⟹
sorted-sublist-map R h xs lo (*p* − *Suc 0*) ⟹ *sorted-sublist-map R h xs* (*Suc p*) *hi* ⟹
lo ≤ *hi* ⟹ *lo* ≤ *p* ⟹ *p* ≤ *hi* ⟹ *hi* < *length xs* ⟹
sorted-sublist-map R h xs lo hi›
apply (*rule merge-sorted-wrt-partitions*) **apply** *auto*
by (*simp add*: *merge-sorted-wrt-partitions isPartition-map-trans*)

lemma *partition-wrt-extend*:
‹*isPartition-wrt R xs lo' hi' p* ⟹
hi < *length xs* ⟹
lo ≤ *lo'* ⟹ *lo'* ≤ *hi* ⟹ *hi'* ≤ *hi* ⟹
lo' ≤ *p* ⟹ *p* ≤ *hi'* ⟹
(⋀ *i*. *lo*≤*i* ⟹ *i* <*lo'* ⟹ *R* (*xs*!*i*) (*xs*!*p*)) ⟹
(⋀ *j*. *hi'*<*j* ⟹ *j*≤*hi* ⟹ *R* (*xs*!*p*) (*xs*!*j*)) ⟹
isPartition-wrt R xs lo hi p›
unfolding *isPartition-wrt-def*
apply *auto*
subgoal by (*meson not-le*)
subgoal by (*metis nat-le-eq-or-lt nat-le-linear*)
done

lemma *partition-map-extend*:
‹*isPartition-map R h xs lo' hi' p* ⟹
hi < *length xs* ⟹
lo ≤ *lo'* ⟹ *lo'* ≤ *hi* ⟹ *hi'* ≤ *hi* ⟹
lo' ≤ *p* ⟹ *p* ≤ *hi'* ⟹
(⋀ *i*. *lo*≤*i* ⟹ *i* <*lo'* ⟹ *R* (*h* (*xs*!*i*)) (*h* (*xs*!*p*))) ⟹
(⋀ *j*. *hi'*<*j* ⟹ *j*≤*hi* ⟹ *R* (*h* (*xs*!*p*)) (*h* (*xs*!*j*))) ⟹
isPartition-map R h xs lo hi p›
by (*auto simp add*: *partition-wrt-extend*)

lemma *isPartition-empty*:
‹(⋀ *j*. ⟦*lo* < *j*; *j* ≤ *hi*⟧ ⟹ *R* (*xs* ! *lo*) (*xs* ! *j*)) ⟹
isPartition-wrt R xs lo hi lo›
by (*auto simp add*: *isPartition-wrt-def*)

lemma *take-ext*:
‹(∀ *i*<*k*. *xs'*!*i*=*xs*!*i*) ⟹
k < *length xs* ⟹ *k* < *length xs'* ⟹

take k xs′ = take k xs›
by (*simp add*: *nth-take-lemma*)

lemma *drop-ext′*:
‹(∀ *i*. *i*≥*k* ∧ *i*<*length xs* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
0<*k* ⟹ *xs*≠[] ⟹ — These corner cases will be dealt with in the next lemma
length xs′=*length xs* ⟹
drop k xs′ = drop k xs›
apply (*rewrite* **in** ‹*drop* - ⌑ = -› *List.rev-rev-ident*[*symmetric*])
apply (*rewrite* **in** ‹- = *drop* - ⌑› *List.rev-rev-ident*[*symmetric*])
apply (*rewrite* **in** ‹⌑ = -› *List.drop-rev*)
apply (*rewrite* **in** ‹- = ⌑› *List.drop-rev*)
apply *simp*
apply (*rule take-ext*)
by (*auto simp add*: *nth-rev*)

lemma *drop-ext*:
‹(∀ *i*. *i*≥*k* ∧ *i*<*length xs* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
length xs′=*length xs* ⟹
drop k xs′ = drop k xs›
apply (*cases xs*)
apply *auto*
apply (*cases k*)
subgoal by (*simp add*: *nth-equalityI*)
subgoal apply (*rule drop-ext′*) **by** *auto*
done

lemma *sublist-ext′*:
‹(∀ *i*. *lo*≤*i*∧*i*≤*hi* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
length xs′ = length xs ⟹
lo ≤ *hi* ⟹ *Suc hi* < *length xs* ⟹
sublist xs′ lo hi = sublist xs lo hi›
apply (*simp add*: *sublist-def*)
apply (*rule take-ext*)
by *auto*

lemma *lt-Suc*: ‹(*a* < *b*) = (*Suc a* = *b* ∨ *Suc a* < *b*)›
by *auto*

lemma *sublist-until-end-eq-drop*: ‹*Suc hi* = *length xs* ⟹ *sublist xs lo hi* = *drop lo xs*›
by (*simp add*: *sublist-def*)

lemma *sublist-ext*:
‹(∀ *i*. *lo*≤*i*∧*i*≤*hi* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
length xs′ = length xs ⟹
lo ≤ *hi* ⟹ *hi* < *length xs* ⟹
sublist xs′ lo hi = sublist xs lo hi›
apply (*auto simp add*: *lt-Suc*[**where** *a*=*hi*])
subgoal by (*auto simp add*: *sublist-until-end-eq-drop drop-ext*)
subgoal by (*auto simp add*: *sublist-ext′*)
done

lemma *sorted-wrt-lower-sublist-still-sorted*:
assumes ‹*sorted-sublist-wrt R xs lo* (*lo′* − *Suc 0*)› **and**

```
    ‹lo ≤ lo'› and ‹lo' < length xs› and
    ‹(∀ i. lo≤i∧i<lo' ⟶ xs'!i=xs!i)› and ‹length xs' = length xs›
  shows ‹sorted-sublist-wrt R xs' lo (lo' − Suc 0)›
proof −
  have l: ‹lo < lo' − 1 ∨ lo ≥ lo'−1›
    by linarith
  show ?thesis
    using l apply auto
    subgoal — lo < lo' - 1
      apply (auto simp add: sorted-sublist-wrt-def)
      apply (rewrite sublist-ext[where xs=xs])
      using assms by (auto simp add: sorted-sublist-wrt-def)
    subgoal — lo >= lo' - 1
      using assms by (auto simp add: sorted-sublist-wrt-le)
    done
qed

lemma sorted-map-lower-sublist-still-sorted:
  assumes ‹sorted-sublist-map R h xs lo (lo' − Suc 0)› and
    ‹lo ≤ lo'› and ‹lo' < length xs› and
    ‹(∀ i. lo≤i∧i<lo' ⟶ xs'!i=xs!i)› and ‹length xs' = length xs›
  shows ‹sorted-sublist-map R h xs' lo (lo' − Suc 0)›
  using assms by (rule sorted-wrt-lower-sublist-still-sorted)

lemma sorted-wrt-upper-sublist-still-sorted:
  assumes ‹sorted-sublist-wrt R xs (hi'+1) hi› and
    ‹lo ≤ lo'› and ‹hi < length xs› and
    ‹∀ j. hi'<j∧j≤hi ⟶ xs'!j=xs!j› and ‹length xs' = length xs›
  shows ‹sorted-sublist-wrt R xs' (hi'+1) hi›
proof −
  have l: ‹hi' + 1 < hi ∨ hi' + 1 ≥ hi›
    by linarith
  show ?thesis
    using l apply auto
    subgoal — hi' + 1 < h
      apply (auto simp add: sorted-sublist-wrt-def)
      apply (rewrite sublist-ext[where xs=xs])
      using assms by (auto simp add: sorted-sublist-wrt-def)
    subgoal — hi ≤ hi' + 1
      using assms by (auto simp add: sorted-sublist-wrt-le)
    done
qed

lemma sorted-map-upper-sublist-still-sorted:
  assumes ‹sorted-sublist-map R h xs (hi'+1) hi› and
    ‹lo ≤ lo'› and ‹hi < length xs› and
    ‹∀ j. hi'<j∧j≤hi ⟶ xs'!j=xs!j› and ‹length xs' = length xs›
  shows ‹sorted-sublist-map R h xs' (hi'+1) hi›
  using assms by (rule sorted-wrt-upper-sublist-still-sorted)
```

The specification of the partition function

```
definition partition-spec :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ 'a list ⇒ nat ⇒ nat ⇒ 'a list ⇒ nat ⇒
bool› where
  ‹partition-spec R h xs lo hi xs' p ≡
    mset xs' = mset xs ∧ — The list is a permutation
    isPartition-map R h xs' lo hi p ∧ — We have a valid partition on the resulting list
```

$lo \le p \wedge p \le hi \wedge$ — The partition index is in bounds
$(\forall\ i.\ i<lo \longrightarrow xs'!i=xs!i) \wedge (\forall\ i.\ hi<i \wedge i<length\ xs' \longrightarrow xs'!i=xs!i)$ — Everything else is unchanged.

lemma *mathias*:
  assumes
      Perm: ‹*mset xs'* = *mset xs*›
    and *I*: ‹*lo*≤*i*› ‹*i*≤*hi*› ‹*xs'*!*i*=*x*›
    and *Bounds*: ‹*hi* < *length xs*›
    and *Fix*: ‹$\bigwedge$ *i*. *i*<*lo* ⟹ *xs'*!*i* = *xs*!*i*› ‹$\bigwedge$ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs'*!*j* = *xs*!*j*›
  shows ‹∃ *j*. *lo*≤*j*∧*j*≤*hi* ∧ *xs*!*j* = *x*›
proof −
  define *xs1 xs2 xs3 xs1' xs2' xs3'* **where**
    ‹*xs1* = *take lo xs*› **and**
    ‹*xs2* = *take* (*Suc hi* − *lo*) (*drop lo xs*)› **and**
    ‹*xs3* = *drop* (*Suc hi*) *xs*› **and**
    ‹*xs1'* = *take lo xs'*› **and**
    ‹*xs2'* = *take* (*Suc hi* − *lo*) (*drop lo xs'*)› **and**
    ‹*xs3'* = *drop* (*Suc hi*) *xs'*›
  have [*simp*]: ‹*length xs'* = *length xs*›
    using *Perm* **by** (*auto dest*: *mset-eq-length*)
  have [*simp*]: ‹*mset xs1* = *mset xs1'*›
    using *Fix*(*1*) **unfolding** *xs1-def xs1'-def*
    by (*metis Perm le-cases mset-eq-length nth-take-lemma take-all*)
  have [*simp*]: ‹*mset xs3* = *mset xs3'*›
    using *Fix*(*2*) **unfolding** *xs3-def xs3'-def mset-drop-upto*
    by (*auto intro*: *image-mset-cong2*)
  have ‹*xs* = *xs1* @ *xs2* @ *xs3*› ‹*xs'* = *xs1'* @ *xs2'* @ *xs3'*›
    using *I* **unfolding** *xs1-def xs2-def xs3-def xs1'-def xs2'-def xs3'-def*
    by (*metis append.assoc append-take-drop-id le-SucI le-add-diff-inverse order-trans take-add*)+
  moreover have ‹*xs* ! *i* = *xs2* ! (*i* − *lo*)› ‹*i* ≥ *length xs1*›
    using *I Bounds* **unfolding** *xs2-def xs1-def* **by** (*auto simp*: *nth-take min-def*)
  moreover have ‹*x* ∈ *set xs2'*›
    using *I Bounds* **unfolding** *xs2'-def*
    by (*auto simp*: *in-set-take-conv-nth*
      intro!: *exI*[*of* - ‹*i* − *lo*›])
  ultimately have ‹*x* ∈ *set xs2*›
    using *Perm I* **by** (*auto dest*: *mset-eq-setD*)
  then obtain *j* **where** ‹*xs* ! (*lo* + *j*) = *x*› ‹*j* ≤ *hi* − *lo*›
    unfolding *in-set-conv-nth xs2-def*
    by *auto*
  then show *?thesis*
    using *Bounds I*
    by (*auto intro*: *exI*[*of* - ‹*lo*+*j*›])
qed

If we fix the left and right rest of two permutated lists, then the sublists are also permutations.

But we only need that the sets are equal.

lemma *mset-sublist-incl*:
  assumes *Perm*: ‹*mset xs'* = *mset xs*›
    and *Fix*: ‹$\bigwedge$ *i*. *i*<*lo* ⟹ *xs'*!*i* = *xs*!*i*› ‹$\bigwedge$ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs'*!*j* = *xs*!*j*›
    and *bounds*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›
  shows ‹*set* (*sublist xs' lo hi*) ⊆ *set* (*sublist xs lo hi*)›
proof

fix *x*
assume ‹*x* ∈ *set* (*sublist xs′ lo hi*)›
then have ‹∃ *i*. *lo*≤*i*∧*i*≤*hi* ∧ *xs′*!*i*=*x*›
by (*metis assms*(*1*) *bounds*(*1*) *bounds*(*2*) *size-mset sublist-el′*)
then obtain *i* **where** *I*: ‹*lo*≤*i*› ‹*i*≤*hi*› ‹*xs′*!*i*=*x*› **by** *blast*
have ‹∃ *j*. *lo*≤*j*∧*j*≤*hi* ∧ *xs*!*j*=*x*›
using *Perm I bounds*(*2*) *Fix* **by** (*rule mathias*, *auto*)
then show ‹*x* ∈ *set* (*sublist xs lo hi*)›
by (*simp add*: *bounds*(*1*) *bounds*(*2*) *sublist-el′*)
qed

lemma *mset-sublist-eq*:
assumes ‹*mset xs′* = *mset xs*›
and ‹⋀ *i*. *i*<*lo* ⟹ *xs′*!*i* = *xs*!*i*›
and ‹⋀ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs′*!*j* = *xs*!*j*›
and *bounds*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›
shows ‹*set* (*sublist xs′ lo hi*) = *set* (*sublist xs lo hi*)›
proof
show ‹*set* (*sublist xs′ lo hi*) ⊆ *set* (*sublist xs lo hi*)›
apply (*rule mset-sublist-incl*)
using *assms* **by** *auto*
show ‹*set* (*sublist xs lo hi*) ⊆ *set* (*sublist xs′ lo hi*)›
apply (*rule mset-sublist-incl*)
by (*metis assms size-mset*)+
qed

Our abstract recursive quicksort procedure. We abstract over a partition procedure.

definition *quicksort* :: ‹('*b* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *nat* × *nat* × '*a list* ⇒ '*a list nres*› **where**
‹*quicksort R h* = (λ(*lo*,*hi*,*xs0*). *do* {
RECT (λ*f* (*lo*,*hi*,*xs*). *do* {
ASSERT(*lo* ≤ *hi* ∧ *hi* < *length xs* ∧ *mset xs* = *mset xs0*); — Premise for a partition function
(*xs*, *p*) ← *SPEC*(*uncurry* (*partition-spec R h xs lo hi*)); — Abstract partition function
ASSERT(*mset xs* = *mset xs0*);
xs ← (*if p*−*1*≤*lo then RETURN xs else f* (*lo*, *p*−*1*, *xs*));
ASSERT(*mset xs* = *mset xs0*);
if hi≤*p*+*1 then RETURN xs else f* (*p*+*1*, *hi*, *xs*)
}) (*lo*,*hi*,*xs0*)
})›

As premise for quicksor, we only need that the indices are ok.

definition *quicksort-pre* :: ‹('*b* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a list* ⇒ *nat* ⇒ *nat* ⇒ '*a list* ⇒ *bool*›
where
‹*quicksort-pre R h xs0 lo hi xs* ≡ *lo* ≤ *hi* ∧ *hi* < *length xs* ∧ *mset xs* = *mset xs0*›

definition *quicksort-post* :: ‹('*b* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *nat* ⇒ *nat* ⇒ '*a list* ⇒ '*a list* ⇒ *bool*›
where
‹*quicksort-post R h lo hi xs xs′* ≡
mset xs′ = *mset xs* ∧
sorted-sublist-map R h xs′ lo hi ∧
(∀ *i*. *i*<*lo* ⟶ *xs′*!*i* = *xs*!*i*) ∧
(∀ *j*. *hi*<*j*∧*j*<*length xs* ⟶ *xs′*!*j* = *xs*!*j*)›

Convert Pure to HOL

lemma *quicksort-postI*:

```
  ‹⟦mset xs' = mset xs; sorted-sublist-map R h xs' lo hi; (⋀ i. ⟦i<lo⟧ ⟹ xs'!i = xs!i); (⋀ j. ⟦hi<j; j<length xs⟧ ⟹ xs'!j = xs!j)⟧ ⟹ quicksort-post R h lo hi xs xs'›
  by (auto simp add: quicksort-post-def)
```

The first case for the correctness proof of (abstract) quicksort: We assume that we called the partition function, and we have $p - (1::'a) \leq lo$ and $hi \leq p + (1::'a)$.

```
lemma quicksort-correct-case1:
  assumes trans: ‹⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)› and lin: ‹⋀x y. R (h x) (h y) ∨ R (h y) (h x)›
    and pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
    and ifs: ‹p−1 ≤ lo› ‹hi ≤ p+1›
  shows ‹quicksort-post R h lo hi xs xs'›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs›
    using pre by (auto simp add: quicksort-pre-def)

  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)


  have sorted-lower: ‹sorted-sublist-map R h xs' lo (p − Suc 0)›
  proof −
    show ?thesis
      apply (rule sorted-sublist-wrt-le)
      subgoal using ifs(1) by auto
      subgoal using ifs(1) mset-eq-length part(1) pre(1) pre(2) by fastforce
      done
  qed

  have sorted-upper: ‹sorted-sublist-map R h xs' (Suc p) hi›
  proof −
    show ?thesis
      apply (rule sorted-sublist-wrt-le)
      subgoal using ifs(2) by auto
      subgoal using ifs(1) mset-eq-length part(1) pre(1) pre(2) by fastforce
      done
  qed

  have sorted-middle: ‹sorted-sublist-map R h xs' lo hi›
  proof −
    show ?thesis
      apply (rule merge-sorted-map-partitions[where p=p])
      subgoal by (rule trans)
      subgoal by (rule part)
      subgoal by (rule sorted-lower)
      subgoal by (rule sorted-upper)
      subgoal using pre(1) by auto
      subgoal by (simp add: part(4))
      subgoal by (simp add: part(5))
```

```
    subgoal by (metis part(1) pre(2) size-mset)
    done
  qed

  show ?thesis
  proof (intro quicksort-postI)
    show ‹mset xs' = mset xs›
      by (simp add: part(1))
  next
    show ‹sorted-sublist-map R h xs' lo hi›
      by (rule sorted-middle)
  next
    show ‹⋀i. i < lo ⟹ xs' ! i = xs ! i›
    using part(6) by blast
  next
    show ‹⋀j. ⟦hi < j; j < length xs⟧ ⟹ xs' ! j = xs ! j›
      by (metis part(1) part(7) size-mset)
  qed
qed
```

In the second case, we have to show that the precondition still holds for (p+1, hi, x') after the partition.

```
lemma quicksort-correct-case2:
  assumes
      pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
    and ifs: ‹¬ hi ≤ p + 1›
  shows ‹quicksort-pre R h xs0 (Suc p) hi xs'›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs = mset xs0›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)
  show ?thesis
    unfolding quicksort-pre-def
  proof (intro conjI)
    show ‹Suc p ≤ hi›
      using ifs by linarith
    show ‹hi < length xs'›
      by (metis part(1) pre(2) size-mset)
    show ‹mset xs' = mset xs0›
      using pre(3) part(1) by (auto dest: mset-eq-setD)
  qed
qed
```

```
lemma quicksort-post-set:
  assumes ‹quicksort-post R h lo hi xs xs'›
    and bounds: ‹lo ≤ hi› ‹hi < length xs›
```

```
  shows ‹set (sublist xs' lo hi) = set (sublist xs lo hi)›
proof −
  have ‹mset xs' = mset xs› ‹⋀ i. i<lo ⟹ xs'!i = xs!i› ‹⋀ j. ⟦hi<j; j<length xs⟧ ⟹ xs'!j = xs!j›
    using assms by (auto simp add: quicksort-post-def)
  then show ?thesis
    using bounds by (rule mset-sublist-eq, auto)
qed
```

In the third case, we have run quicksort recursively on (p+1, hi, xs') after the partition, with hi<=p+1 and p-1<=lo.

```
lemma quicksort-correct-case3:
  assumes trans: ‹⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)› and lin: ‹⋀x y. R (h x)
(h y) ∨ R (h y) (h x)›
    and pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
    and ifs: ‹p − Suc 0 ≤ lo› ‹¬ hi ≤ Suc p›
    and IH1': ‹quicksort-post R h (Suc p) hi xs' xs''›
  shows ‹quicksort-post R h lo hi xs xs''›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs = mset xs0›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)
  have IH1: ‹mset xs'' = mset xs'› ‹sorted-sublist-map R h xs'' (Suc p) hi›
      ‹⋀ i. i<Suc p ⟹ xs'' ! i = xs' ! i› ‹⋀ j. ⟦hi < j; j < length xs'⟧ ⟹ xs'' ! j = xs' ! j›
    using IH1' by (auto simp add: quicksort-post-def)
  note IH1-perm = quicksort-post-set[OF IH1']

  have still-partition: ‹isPartition-map R h xs'' lo hi p›
  proof(intro isPartition-wrtI)
    fix i assume ‹lo ≤ i› ‹i < p›
    show ‹R (h (xs'' ! i)) (h (xs'' ! p))›
```

This holds because this part hasn't changed

```
      using IH1(3) ‹i < p› ‹lo ≤ i› isPartition-wrt-def part(3) by fastforce
    next
      fix j assume ‹p < j› ‹j ≤ hi›
```

Obtain the position *posJ* where $xs'' ! j$ was stored in xs'.

```
      have ‹xs''!j ∈ set (sublist xs'' (Suc p) hi)›
        by (metis IH1(1) Suc-leI ‹j ≤ hi› ‹p < j› less-le-trans mset-eq-length part(1) pre(2) sublist-el')
      then have ‹xs''!j ∈ set (sublist xs' (Suc p) hi)›
        by (metis IH1-perm ifs(2) nat-le-linear part(1) pre(2) size-mset)
      then have ‹∃ posJ. Suc p≤posJ∧posJ≤hi ∧ xs''!j = xs'!posJ›
        by (metis Suc-leI ‹j ≤ hi› ‹p < j› less-le-trans part(1) pre(2) size-mset sublist-el')
      then obtain posJ :: nat where PosJ: ‹Suc p≤posJ› ‹posJ≤hi› ‹xs''!j = xs'!posJ› by blast

      then show ‹R (h (xs'' ! p)) (h (xs'' ! j))›
        by (metis IH1(3) Suc-le-lessD isPartition-wrt-def lessI part(3))
  qed
```

```
  have sorted-lower: ‹sorted-sublist-map R h xs'' lo (p − Suc 0)›
  proof −
    show ?thesis
      apply (rule sorted-sublist-wrt-le)
      subgoal by (simp add: ifs(1))
      subgoal using IH1(1) mset-eq-length part(1) part(5) pre(2) by fastforce
      done
  qed

  note sorted-upper = IH1(2)

  have sorted-middle: ‹sorted-sublist-map R h xs'' lo hi›
  proof −
    show ?thesis
      apply (rule merge-sorted-map-partitions[where p=p])
      subgoal by (rule trans)
      subgoal by (rule still-partition)
      subgoal by (rule sorted-lower)
      subgoal by (rule sorted-upper)
      subgoal using pre(1) by auto
      subgoal by (simp add: part(4))
      subgoal by (simp add: part(5))
      subgoal by (metis IH1(1) part(1) pre(2) size-mset)
      done
  qed


  show ?thesis
  proof (intro quicksort-postI)
    show ‹mset xs'' = mset xs›
      using part(1) IH1(1) by auto — I was faster than sledgehammer :-)
  next
    show ‹sorted-sublist-map R h xs'' lo hi›
      by (rule sorted-middle)
  next
    show ‹⋀i. i < lo ⟹ xs'' ! i = xs ! i›
      using IH1(3) le-SucI part(4) part(6) by auto
  next show ‹⋀j. hi < j ⟹ j < length xs ⟹ xs'' ! j = xs ! j›
      by (metis IH1(4) part(1) part(7) size-mset)
  qed
qed
```

In the 4th case, we have to show that the premise holds for (lo, $p - (1::'b)$, xs'), in case $\neg\ p - (1::'a) \leq lo$

Analogous to case 2.

```
lemma quicksort-correct-case4:
  assumes
       pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
    and ifs: ‹¬ p − Suc 0 ≤ lo ›
  shows ‹quicksort-pre R h xs0 lo (p−Suc 0) xs'›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them

to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs0 = mset xs›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)

  show ?thesis
    unfolding quicksort-pre-def
  proof (intro conjI)
    show ‹lo ≤ p − Suc 0›
      using ifs by linarith
    show ‹p − Suc 0 < length xs'›
      using mset-eq-length part(1) part(5) pre(2) by fastforce
    show ‹mset xs' = mset xs0›
      using pre(3) part(1) by (auto dest: mset-eq-setD)
  qed
qed
```

In the 5th case, we have run quicksort recursively on (lo, p-1, xs').

```
lemma quicksort-correct-case5:
  assumes trans: ‹⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)› and lin: ‹⋀x y. R (h x)
(h y) ∨ R (h y) (h x)›
    and pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
    and ifs: ‹¬ p − Suc 0 ≤ lo› ‹hi ≤ Suc p›
    and IH1': ‹quicksort-post R h lo (p − Suc 0) xs' xs''›
  shows ‹quicksort-post R h lo hi xs xs''›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)
  have IH1: ‹mset xs'' = mset xs'› ‹sorted-sublist-map R h xs'' lo (p − Suc 0)›
    ‹⋀ i. i<lo ⟹ xs''!i = xs'!i› ‹⋀ j. ⟦p−Suc 0<j; j<length xs'⟧ ⟹ xs''!j = xs'!j›
    using IH1' by (auto simp add: quicksort-post-def)
  note IH1-perm = quicksort-post-set[OF IH1']


 have still-partition: ‹isPartition-map R h xs'' lo hi p›
  proof(intro isPartition-wrtI)
    fix i assume ‹lo ≤ i› ‹i < p›
```

Obtain the position *posI* where $xs'' ! i$ was stored in xs'.

```
      have ‹xs''!i ∈ set (sublist xs'' lo (p−Suc 0))›
       by (metis (no-types, lifting) IH1(1) Suc-leI Suc-pred ‹i < p› ‹lo ≤ i› le-less-trans less-imp-diff-less
mset-eq-length not-le not-less-zero part(1) part(5) pre(2) sublist-el')
      then have ‹xs''!i ∈ set (sublist xs' lo (p−Suc 0))›
```

by (*metis IH1-perm ifs(1) le-less-trans less-imp-diff-less mset-eq-length nat-le-linear part(1) part(5) pre(2)*)
then have ‹∃ *posI*. *lo*≤*posI*∧*posI*≤*p*−*Suc 0* ∧ *xs''*!*i* = *xs'*!*posI*›
proof − — sledgehammer
have *p* − *Suc 0* < *length xs*
by (*meson diff-le-self le-less-trans part(5) pre(2)*)
then show *?thesis*
by (*metis* (*no-types*) ‹*xs''* ! *i* ∈ *set* (*sublist xs' lo* (*p* − *Suc 0*))› *ifs(1) mset-eq-length nat-le-linear part(1) sublist-el'*)
qed
then obtain *posI* :: *nat* **where** *PosI*: ‹*lo*≤*posI*› ‹*posI*≤*p*−*Suc 0*› ‹*xs''*!*i* = *xs'*!*posI*› **by** *blast*
then show ‹*R* (*h* (*xs''* ! *i*)) (*h* (*xs''* ! *p*))›
by (*metis* (*no-types*, *lifting*) *IH1(4)* ‹*i* < *p*› *diff-Suc-less isPartition-wrt-def le-less-trans mset-eq-length not-le not-less-eq part(1) part(3) part(5) pre(2) zero-less-Suc*)
next
fix *j* **assume** ‹*p* < *j*› ‹*j* ≤ *hi*›
then show ‹*R* (*h* (*xs''* ! *p*)) (*h* (*xs''* ! *j*))›

This holds because this part hasn't changed

by (*smt IH1(4) add-diff-cancel-left' add-diff-inverse-nat diff-Suc-eq-diff-pred diff-le-self ifs(1) isPartition-wrt-def le-less-Suc-eq less-le-trans mset-eq-length nat-less-le part(1) part(3) part(4) plus-1-eq-Suc pre(2)*)
qed

note *sorted-lower* = *IH1(2)*

have *sorted-upper*: ‹*sorted-sublist-map R h xs''* (*Suc p*) *hi*›
proof −
show *?thesis*
apply (*rule sorted-sublist-wrt-le*)
subgoal by (*simp add*: *ifs(2)*)
subgoal using *IH1(1) mset-eq-length part(1) part(5) pre(2)* **by** *fastforce*
done
qed

have *sorted-middle*: ‹*sorted-sublist-map R h xs''* *lo hi*›
proof −
show *?thesis*
apply (*rule merge-sorted-map-partitions*[**where** *p*=*p*])
subgoal by (*rule trans*)
subgoal by (*rule still-partition*)
subgoal by (*rule sorted-lower*)
subgoal by (*rule sorted-upper*)
subgoal using *pre(1)* **by** *auto*
subgoal by (*simp add*: *part(4)*)
subgoal by (*simp add*: *part(5)*)
subgoal by (*metis IH1(1) part(1) pre(2) size-mset*)
done
qed

show *?thesis*
proof (*intro quicksort-postI*)
show ‹*mset xs''* = *mset xs*›

```
    by (simp add: IH1(1) part(1))
  next
    show ‹sorted-sublist-map R h xs'' lo hi›
      by (rule sorted-middle)
  next
    show ‹⋀i. i < lo ⟹ xs'' ! i = xs ! i›
      by (simp add: IH1(3) part(6))
  next
    show ‹⋀j. hi < j ⟹ j < length xs ⟹ xs'' ! j = xs ! j›
      by (metis IH1(4) diff-le-self dual-order.strict-trans2 mset-eq-length part(1) part(5) part(7))
  qed
qed
```

In the 6th case, we have run quicksort recursively on (lo, p-1, xs'). We show the precondition on the second call on (p+1, hi, xs")

```
lemma quicksort-correct-case6:
  assumes
      pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
    and ifs:  ‹¬ p − Suc 0 ≤ lo› ‹¬ hi ≤ Suc p›
    and IH1: ‹quicksort-post R h lo (p − Suc 0) xs' xs''›
  shows ‹quicksort-pre R h xs0 (Suc p) hi xs''›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs0 = mset xs›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)
  have IH1: ‹mset xs'' = mset xs'› ‹sorted-sublist-map R h xs'' lo (p − Suc 0)›
    ‹⋀ i. i<lo ⟹ xs''!i = xs'!i› ‹⋀ j. ⟦p−Suc 0<j; j<length xs'⟧ ⟹ xs''!j = xs'!j›
    using IH1 by (auto simp add: quicksort-post-def)

  show ?thesis
    unfolding quicksort-pre-def
  proof (intro conjI)
    show ‹Suc p ≤ hi›
      using ifs(2) by linarith
    show ‹hi < length xs''›
      using IH1(1) mset-eq-length part(1) pre(2) by fastforce
    show ‹mset xs'' = mset xs0›
      using pre(3) part(1) IH1(1) by (auto dest: mset-eq-setD)
  qed
qed
```

In the 7th (and last) case, we have run quicksort recursively on (lo, p-1, xs'). We show the postcondition on the second call on (p+1, hi, xs")

```
lemma quicksort-correct-case7:
  assumes trans: ‹⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)› and lin: ‹⋀x y. R (h x)
(h y) ∨ R (h y) (h x)›
    and pre: ‹quicksort-pre R h xs0 lo hi xs›
    and part: ‹partition-spec R h xs lo hi xs' p›
```

```
    and ifs: ‹¬ p − Suc 0 ≤ lo› ‹¬ hi ≤ Suc p›
    and IH1': ‹quicksort-post R h lo (p − Suc 0) xs' xs''›
    and IH2': ‹quicksort-post R h (Suc p) hi xs'' xs'''›
  shows ‹quicksort-post R h lo hi xs xs'''›
proof −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)
  have IH1: ‹mset xs'' = mset xs'› ‹sorted-sublist-map R h xs'' lo (p − Suc 0)›
    ‹⋀ i. i<lo ⟹ xs''!i = xs'!i› ‹⋀ j. ⟦p−Suc 0<j; j<length xs'⟧ ⟹ xs''!j = xs'!j›
    using IH1' by (auto simp add: quicksort-post-def)
  note IH1-perm = quicksort-post-set[OF IH1']
  have IH2: ‹mset xs''' = mset xs''› ‹sorted-sublist-map R h xs''' (Suc p) hi›
    ‹⋀ i. i<Suc p ⟹ xs'''!i = xs''!i› ‹⋀ j. ⟦hi<j; j<length xs''⟧ ⟹ xs'''!j = xs''!j›
    using IH2' by (auto simp add: quicksort-post-def)
  note IH2-perm = quicksort-post-set[OF IH2']
```

We still have a partition after the first call (same as in case 5)

```
  have still-partition1: ‹isPartition-map R h xs'' lo hi p›
  proof(intro isPartition-wrtI)
    fix i assume ‹lo ≤ i› ‹i < p›
```

Obtain the position *posI* where $xs'' ! i$ was stored in xs'.

```
      have ‹xs''!i ∈ set (sublist xs'' lo (p−Suc 0))›
       by (metis (no-types, lifting) IH1(1) Suc-leI Suc-pred ‹i < p› ‹lo ≤ i› le-less-trans less-imp-diff-less
mset-eq-length not-le not-less-zero part(1) part(5) pre(2) sublist-el')
      then have ‹xs''!i ∈ set (sublist xs' lo (p−Suc 0))›
          by (metis IH1-perm ifs(1) le-less-trans less-imp-diff-less mset-eq-length nat-le-linear part(1)
part(5) pre(2))
      then have ‹∃ posI. lo≤posI∧posI≤p−Suc 0 ∧ xs''!i = xs'!posI›
      proof − — sledgehammer
        have p − Suc 0 < length xs
          by (meson diff-le-self le-less-trans part(5) pre(2))
        then show ?thesis
         by (metis (no-types) ‹xs'' ! i ∈ set (sublist xs' lo (p − Suc 0))› ifs(1) mset-eq-length nat-le-linear
part(1) sublist-el')
      qed
      then obtain posI :: nat where PosI: ‹lo≤posI› ‹posI≤p−Suc 0› ‹xs''!i = xs'!posI› by blast
      then show ‹R (h (xs'' ! i)) (h (xs'' ! p))›
           by (metis (no-types, lifting) IH1(4) ‹i < p› diff-Suc-less isPartition-wrt-def le-less-trans
mset-eq-length not-le not-less-eq part(1) part(3) part(5) pre(2) zero-less-Suc)
    next
      fix j assume ‹p < j› ‹j ≤ hi›
      then show ‹R (h (xs'' ! p)) (h (xs'' ! j))›
```

This holds because this part hasn't changed

```
     by (smt IH1(4) add-diff-cancel-left' add-diff-inverse-nat diff-Suc-eq-diff-pred diff-le-self ifs(1) isPar-
tition-wrt-def le-less-Suc-eq less-le-trans mset-eq-length nat-less-le part(1) part(3) part(4) plus-1-eq-Suc
pre(2))
```

qed

We still have a partition after the second call (similar as in case 3)

have *still-partition2*: ‹*isPartition-map R h xs''' lo hi p*›
proof(*intro isPartition-wrtI*)
fix *i* **assume** ‹*lo* ≤ *i*› ‹*i* < *p*›
show ‹*R* (*h* (*xs'''* ! *i*)) (*h* (*xs'''* ! *p*))›

This holds because this part hasn't changed

using *IH2*(*3*) ‹*i* < *p*› ‹*lo* ≤ *i*› *isPartition-wrt-def still-partition1* **by** *fastforce*
next
fix *j* **assume** ‹*p* < *j*› ‹*j* ≤ *hi*›

Obtain the position *posJ* where *xs'''* ! *j* was stored in *xs'''*.

have ‹*xs'''*!*j* ∈ *set* (*sublist xs'''* (*Suc p*) *hi*)›
by (*metis IH1*(*1*) *IH2*(*1*) *Suc-leI* ‹*j* ≤ *hi*› ‹*p* < *j*› *ifs*(*2*) *nat-le-linear part*(*1*) *pre*(*2*) *size-mset sublist-el'*)
then have ‹*xs'''*!*j* ∈ *set* (*sublist xs''* (*Suc p*) *hi*)›
by (*metis IH1*(*1*) *IH2-perm ifs*(*2*) *mset-eq-length nat-le-linear part*(*1*) *pre*(*2*))
then have ‹∃ *posJ*. *Suc p*≤*posJ*∧*posJ*≤*hi* ∧ *xs'''*!*j* = *xs''*!*posJ*›
by (*metis IH1*(*1*) *ifs*(*2*) *mset-eq-length nat-le-linear part*(*1*) *pre*(*2*) *sublist-el'*)
then obtain *posJ* :: *nat* **where** *PosJ*: ‹*Suc p*≤*posJ*› ‹*posJ*≤*hi*› ‹*xs'''*!*j* = *xs''*!*posJ*› **by** *blast*

then show ‹*R* (*h* (*xs'''* ! *p*)) (*h* (*xs'''* ! *j*))›
proof − — sledgehammer
have ∀ *n na as p*. (*p* (*as* ! *na*::'*a*) (*as* ! *posJ*) ∨ *posJ* ≤ *na*) ∨ ¬ *isPartition-wrt p as n hi na*
by (*metis* (*no-types*) *PosJ*(*2*) *isPartition-wrt-def not-less*)
then show *?thesis*
by (*metis IH2*(*3*) *PosJ*(*1*) *PosJ*(*3*) *lessI not-less-eq-eq still-partition1*)
qed
qed

We have that the lower part is sorted after the first recursive call

note *sorted-lower1* = *IH1*(*2*)

We show that it is still sorted after the second call.

have *sorted-lower2*: ‹*sorted-sublist-map R h xs''' lo* (*p*−*Suc 0*)›
proof −
show *?thesis*
using *sorted-lower1* **apply** (*rule sorted-wrt-lower-sublist-still-sorted*)
subgoal by (*rule part*)
subgoal
using *IH1*(*1*) *mset-eq-length part*(*1*) *part*(*5*) *pre*(*2*) **by** *fastforce*
subgoal
by (*simp add*: *IH2*(*3*))
subgoal
by (*metis IH2*(*1*) *size-mset*)
done
qed

The second IH gives us the the upper list is sorted after the second recursive call

note *sorted-upper2* = *IH2*(*2*)

Finally, we have to show that the entire list is sorted after the second recursive call.

have *sorted-middle*: ‹*sorted-sublist-map R h xs''' lo hi*›
proof −
show *?thesis*
apply (*rule merge-sorted-map-partitions*[**where** *p*=*p*])
subgoal by (*rule trans*)
subgoal by (*rule still-partition2*)
subgoal by (*rule sorted-lower2*)
subgoal by (*rule sorted-upper2*)
subgoal using *pre*(*1*) **by** *auto*
subgoal by (*simp add*: *part*(*4*))
subgoal by (*simp add*: *part*(*5*))
subgoal by (*metis IH1*(*1*) *IH2*(*1*) *part*(*1*) *pre*(*2*) *size-mset*)
done
qed

show *?thesis*
proof (*intro quicksort-postI*)
show ‹*mset xs''' = mset xs*›
by (*simp add*: *IH1*(*1*) *IH2*(*1*) *part*(*1*))
next
show ‹*sorted-sublist-map R h xs''' lo hi*›
by (*rule sorted-middle*)
next
show ‹⋀*i*. *i* < *lo* ⟹ *xs'''* ! *i* = *xs* ! *i*›
using *IH1*(*3*) *IH2*(*3*) *part*(*4*) *part*(*6*) **by** *auto*
next
show ‹⋀*j*. *hi* < *j* ⟹ *j* < *length xs* ⟹ *xs'''* ! *j* = *xs* ! *j*›
by (*metis IH1*(*1*) *IH1*(*4*) *IH2*(*4*) *diff-le-self ifs*(*2*) *le-SucI less-le-trans nat-le-eq-or-lt not-less part*(*1*) *part*(*7*) *size-mset*)
qed

qed

We can now show the correctness of the abstract quicksort procedure, using the refinement framework and the above case lemmas.

lemma *quicksort-correct*:
assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
and *Pre*: ‹*lo0* ≤ *hi0*› ‹*hi0* < *length xs0*›
shows ‹*quicksort R h* (*lo0*,*hi0*,*xs0*) ≤ ⇓ *Id* (*SPEC*(λ*xs*. *quicksort-post R h lo0 hi0 xs0 xs*))›
proof −
have *wf*: ‹*wf* (*measure* (λ(*lo*, *hi*, *xs*). *Suc hi* − *lo*))›
by *auto*
define *pre* **where** ‹*pre* = (λ(*lo*,*hi*,*xs*). *quicksort-pre R h xs0 lo hi xs*)›
define *post* **where** ‹*post* = (λ(*lo*,*hi*,*xs*). *quicksort-post R h lo hi xs*)›
have *pre*: ‹*pre* (*lo0*,*hi0*,*xs0*)›
unfolding *quicksort-pre-def pre-def* **by** (*simp add*: *Pre*)

We first generalize the goal a over all states.

have ‹*WB-Sort.quicksort R h* (*lo0*,*hi0*,*xs0*) ≤ ⇓ *Id* (*SPEC* (*post* (*lo0*,*hi0*,*xs0*)))›
unfolding *quicksort-def prod.case*
apply (*rule RECT-rule*)
apply (*refine-mono*)
apply (*rule wf*)
apply (*rule pre*)

subgoal premises *IH* **for** *f x*
apply (*refine-vcg ASSERT-leI*)
unfolding *pre-def post-def*

subgoal — First premise (assertion) for partition
using *IH*(*2*) **by** (*simp add*: *quicksort-pre-def pre-def*)
subgoal — Second premise (assertion) for partition
using *IH*(*2*) **by** (*simp add*: *quicksort-pre-def pre-def*)
subgoal
using *IH*(*2*) **by** (*auto simp add*: *quicksort-pre-def pre-def dest*: *mset-eq-setD*)

Termination case: $p - (1::'c) \leq lo'$ and $hi' \leq p + (1::'c)$; directly show postcondition

subgoal unfolding *partition-spec-def* **by** (*auto dest*: *mset-eq-setD*)
subgoal — Postcondition (after partition)
apply −
using *IH*(*2*) **unfolding** *pre-def* **apply** (*simp*, *elim conjE*, *split prod.splits*)
using *trans lin* **apply** (*rule quicksort-correct-case1*) **by** *auto*

Case $p - (1::'c) \leq lo'$ and $hi' < p + (1::'c)$ (Only second recursive call)

subgoal
apply (*rule IH*(*1*)[*THEN order-trans*])

Show that the invariant holds for the second recursive call

subgoal
using *IH*(*2*) **unfolding** *pre-def* **apply** (*simp*, *elim conjE*, *split prod.splits*)
apply (*rule quicksort-correct-case2*) **by** *auto*

Wellfoundness (easy)

subgoal by (*auto simp add*: *quicksort-pre-def partition-spec-def*)

Show that the postcondition holds

subgoal
apply (*simp add*: *Misc.subset-Collect-conv post-def*, *intro allI impI*, *elim conjE*)
using *trans lin* **apply** (*rule quicksort-correct-case3*)
using *IH*(*2*) **unfolding** *pre-def* **by** *auto*
done

Case: At least the first recursive call

subgoal
apply (*rule IH*(*1*)[*THEN order-trans*])

Show that the precondition holds for the first recursive call

subgoal
using *IH*(*2*) **unfolding** *pre-def post-def* **apply** (*simp*, *elim conjE*, *split prod.splits*) **apply** *auto*
apply (*rule quicksort-correct-case4*) **by** *auto*

Wellfoundness for first recursive call (easy)

subgoal by (*auto simp add*: *quicksort-pre-def partition-spec-def*)

Simplify some refinement suff...

apply (*simp add*: *Misc.subset-Collect-conv ASSERT-leI*, *intro allI impI conjI*, *elim conjE*)
apply (*rule ASSERT-leI*)
apply (*simp-all add*: *Misc.subset-Collect-conv ASSERT-leI*)

subgoal unfolding *quicksort-post-def pre-def post-def* **by** (*auto dest*: *mset-eq-setD*)

Only the first recursive call: show postcondition

subgoal
using *trans lin* **apply** (*rule quicksort-correct-case5*)
using *IH*(*2*) **unfolding** *pre-def post-def* **by** *auto*

apply (*rule ASSERT-leI*)
subgoal unfolding *quicksort-post-def pre-def post-def* **by** (*auto dest*: *mset-eq-setD*)

Both recursive calls.

subgoal
apply (*rule IH*(*1*)[*THEN order-trans*])

Show precondition for second recursive call (after the first call)

subgoal
unfolding *pre-def post-def*
apply *auto*
apply (*rule quicksort-correct-case6*)
using *IH*(*2*) **unfolding** *pre-def post-def* **by** *auto*

Wellfoundedness for second recursive call (easy)

subgoal by (*auto simp add*: *quicksort-pre-def partition-spec-def*)

Show that the postcondition holds (after both recursive calls)

subgoal
apply (*simp add*: *Misc.subset-Collect-conv*, *intro allI impI*, *elim conjE*)
using *trans lin* **apply** (*rule quicksort-correct-case7*)
using *IH*(*2*) **unfolding** *pre-def post-def* **by** *auto*
done
done
done
done

Finally, apply the generalized lemma to show the thesis.

then show *?thesis* **unfolding** *post-def* **by** *auto*
qed

definition *partition-main-inv* :: ‹(*'b* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *nat* ⇒ *nat* ⇒ *'a list* ⇒ (*nat*×*nat*×*'a list*) ⇒ *bool*› **where**
‹*partition-main-inv R h lo hi xs0 p* ≡
case p of (*i*,*j*,*xs*) ⇒
j < *length xs* ∧ *j* ≤ *hi* ∧ *i* < *length xs* ∧ *lo* ≤ *i* ∧ *i* ≤ *j* ∧ *mset xs* = *mset xs0* ∧
(∀ *k*. *k* ≥ *lo* ∧ *k* < *i* ⟶ *R* (*h* (*xs*!*k*)) (*h* (*xs*!*hi*))) ∧ — All elements from *lo* to *i* − (*1*::*'c*) are smaller than the pivot
(∀ *k*. *k* ≥ *i* ∧ *k* < *j* ⟶ *R* (*h* (*xs*!*hi*)) (*h* (*xs*!*k*))) ∧ — All elements from *i* to *j* − (*1*::*'c*) are greater than the pivot
(∀ *k*. *k* < *lo* ⟶ *xs*!*k* = *xs0*!*k*) ∧ — Everything below *lo* is unchanged
(∀ *k*. *k* ≥ *j* ∧ *k* < *length xs* ⟶ *xs*!*k* = *xs0*!*k*) — All elements from *j* are unchanged (including everyting above *hi*)

›

The main part of the partition function. The pivot is assumed to be the last element. This is exactly the "Lomuto partition scheme" partition function from Wikipedia.

definition *partition-main* :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ nat ⇒ nat ⇒ 'a list ⇒ ('a list × nat) *nres*› **where**
‹*partition-main R h lo hi xs0* = *do* {
ASSERT(*hi* < *length xs0*);
pivot ← *RETURN* (*h* (*xs0* ! *hi*));
(*i*,*j*,*xs*) ← $WHILE_T^{partition\text{-}main\text{-}inv\ R\ h\ lo\ hi\ xs0}$ — We loop from *j* = *lo* to *j* = *hi* − (*1*::'*c*).
(λ(*i*,*j*,*xs*). *j* < *hi*)
(λ(*i*,*j*,*xs*). *do* {
ASSERT(*i* < *length xs* ∧ *j* < *length xs*);
if R (*h* (*xs*!*j*)) *pivot*
then RETURN (*i*+*1*, *j*+*1*, *swap xs i j*)
else RETURN (*i*, *j*+*1*, *xs*)
})
(*lo*, *lo*, *xs0*); — i and j are both initialized to lo
ASSERT(*i* < *length xs* ∧ *j* = *hi* ∧ *lo* ≤ *i* ∧ *hi* < *length xs* ∧ *mset xs* = *mset xs0*);
RETURN (*swap xs i hi*, *i*)
}›

lemma *partition-main-correct*:
assumes *bounds*: ‹*hi* < *length xs*› ‹*lo* ≤ *hi*› **and**
trans: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*partition-main R h lo hi xs* ≤ *SPEC*(λ(*xs'*, *p*). *mset xs* = *mset xs'* ∧
lo ≤ *p* ∧ *p* ≤ *hi* ∧ *isPartition-map R h xs' lo hi p* ∧ (∀ *i*. *i*<*lo* ⟶ *xs'*!*i*=*xs*!*i*) ∧ (∀ *i*. *hi*<*i*∧*i*<*length xs'* ⟶ *xs'*!*i*=*xs*!*i*))›
proof −
have *K*: ‹*b* ≤ *hi* − *Suc n* ⟹ *n* > *0* ⟹ *Suc n* ≤ *hi* ⟹ *Suc b* ≤ *hi* − *n*› **for** *b hi n*
by *auto*
have *L*: ‹~ *R* (*h x*) (*h y*) ⟹ *R* (*h y*) (*h x*)› **for** *x y* — Corollary of linearity
using *assms* **by** *blast*
have *M*: ‹*a* < *Suc b* ≡ *a* = *b* ∨ *a* < *b*› **for** *a b*
by *linarith*
have *N*: ‹(*a*::*nat*) ≤ *b* ≡ *a* = *b* ∨ *a* < *b*› **for** *a b*
by *arith*

show *?thesis*
unfolding *partition-main-def choose-pivot-def*
apply (*refine-vcg WHILEIT-rule*[**where** *R* = ‹*measure*(λ(*i*,*j*,*xs*). *hi*−*j*)›])
subgoal using *assms* **by** *blast* — We feed our assumption to the assertion
subgoal by *auto* — WF
subgoal — Invariant holds before the first iteration
unfolding *partition-main-inv-def*
using *assms* **apply** *simp* **by** *linarith*
subgoal unfolding *partition-main-inv-def* **by** *simp*
subgoal unfolding *partition-main-inv-def* **by** *simp*
subgoal
unfolding *partition-main-inv-def*
apply (*auto dest*: *mset-eq-length*)
done

subgoal unfolding *partition-main-inv-def* **by** (*auto dest*: *mset-eq-length*)
subgoal
unfolding *partition-main-inv-def* **apply** (*auto dest*: *mset-eq-length*)
by (*metis L M mset-eq-length nat-le-eq-or-lt*)

subgoal unfolding *partition-main-inv-def* **by** *simp* — assertions, etc
subgoal unfolding *partition-main-inv-def* **by** *simp*
subgoal unfolding *partition-main-inv-def* **by** (*auto dest*: *mset-eq-length*)
subgoal unfolding *partition-main-inv-def* **by** *simp*
subgoal unfolding *partition-main-inv-def* **by** (*auto dest*: *mset-eq-length*)
subgoal unfolding *partition-main-inv-def* **by** (*auto dest*: *mset-eq-length*)
subgoal unfolding *partition-main-inv-def* **by** (*auto dest*: *mset-eq-length*)
subgoal unfolding *partition-main-inv-def* **by** *simp*
subgoal unfolding *partition-main-inv-def* **by** *simp*

subgoal — After the last iteration, we have a partitioning! :-)
unfolding *partition-main-inv-def* **by** (*auto simp add*: *isPartition-wrt-def*)
subgoal — And the lower out-of-bounds parts of the list haven't been changed
unfolding *partition-main-inv-def* **by** *auto*
subgoal — And the upper out-of-bounds parts of the list haven't been changed
unfolding *partition-main-inv-def* **by** *auto*
done
qed

definition *partition-between* :: ‹('*b* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *nat* ⇒ *nat* ⇒ '*a list* ⇒ ('*a list* × *nat*) *nres*› **where**
‹*partition-between R h lo hi xs0* = *do* {
ASSERT(*hi* < *length xs0* ∧ *lo* ≤ *hi*);
k ← *choose-pivot R h xs0 lo hi*; — choice of pivot
ASSERT(*k* < *length xs0*);
xs ← *RETURN* (*swap xs0 k hi*); — move the pivot to the last position, before we start the actual loop
ASSERT(*length xs* = *length xs0*);
partition-main R h lo hi xs
}›

lemma *partition-between-correct*:
assumes ‹*hi* < *length xs*› **and** ‹*lo* ≤ *hi*› **and**
‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*partition-between R h lo hi xs* ≤ *SPEC*(*uncurry* (*partition-spec R h xs lo hi*))›
proof −
have *K*: ‹*b* ≤ *hi* − *Suc n* ⟹ *n* > *0* ⟹ *Suc n* ≤ *hi* ⟹ *Suc b* ≤ *hi* − *n*› **for** *b hi n*
by *auto*
show *?thesis*
unfolding *partition-between-def choose-pivot-def*
apply (*refine-vcg partition-main-correct*)
using *assms* **apply** (*auto dest*: *mset-eq-length simp add*: *partition-spec-def*)
by (*metis dual-order.strict-trans2 less-imp-not-eq2 mset-eq-length swap-nth*)
qed

We use the median of the first, the middle, and the last element.

definition *choose-pivot3* **where**
‹*choose-pivot3 R h xs lo* (*hi*::*nat*) = *do* {
ASSERT(*lo* < *length xs*);

```
    ASSERT(hi < length xs);
    let k' = (hi − lo) div 2;
    let k = lo + k';
    ASSERT(k < length xs);
    let start = h (xs ! lo);
    let mid = h (xs ! k);
    let end = h (xs ! hi);
    if (R start mid ∧ R mid end) ∨ (R end mid ∧ R mid start) then RETURN k
    else if (R start end ∧ R end mid) ∨ (R mid end ∧ R end start) then RETURN hi
    else RETURN lo
}›
```

— We only have to show that this procedure yields a valid index between *lo* and *hi*.

```
lemma choose-pivot3-choose-pivot:
  assumes ‹lo < length xs› ‹hi < length xs› ‹hi ≥ lo›
  shows ‹choose-pivot3 R h xs lo hi ≤ ⇓ Id (choose-pivot R h xs lo hi)›
  unfolding choose-pivot3-def choose-pivot-def
  using assms by (auto intro!: ASSERT-leI simp: Let-def)
```

The refined partion function: We use the above pivot function and fold instead of non-deterministic iteration.

```
definition partition-between-ref
  :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ nat ⇒ nat ⇒ 'a list ⇒ ('a list × nat) nres›
where
  ‹partition-between-ref R h lo hi xs0 = do {
    ASSERT(hi < length xs0 ∧ hi < length xs0 ∧ lo ≤ hi);
    k ← choose-pivot3 R h xs0 lo hi; — choice of pivot
    ASSERT(k < length xs0);
    xs ← RETURN (swap xs0 k hi); — move the pivot to the last position, before we start the actual
loop
    ASSERT(length xs = length xs0);
    partition-main R h lo hi xs
  }›


lemma partition-main-ref':
  ‹partition-main R h lo hi xs
    ≤ ⇓ ((λ a b c d. Id) a b c d) (partition-main R h lo hi xs)›
  by auto

lemma partition-between-ref-partition-between:
  ‹partition-between-ref R h lo hi xs ≤ (partition-between R h lo hi xs)›
proof −
  have swap: ‹(swap xs k hi, swap xs ka hi) ∈ Id› if ‹k = ka›
    for k ka
    using that by auto
  have [refine0]: ‹(h (xsa ! hi), h (xsaa ! hi)) ∈ Id›
    if ‹(xsa, xsaa) ∈ Id›
    for xsa xsaa
    using that by auto

  show ?thesis
    apply (subst (2) Down-id-eq[symmetric])
    unfolding partition-between-ref-def
      partition-between-def
      OP-def
```

apply (*refine-vcg choose-pivot3-choose-pivot swap partition-main-correct*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
by (*auto intro*: *Refine-Basic.Id-refine dest*: *mset-eq-length*)
qed

Technical lemma for sepref

lemma *partition-between-ref-partition-between'*:
‹(*uncurry* (*uncurry* (*partition-between-ref R h*)), *uncurry* (*uncurry* (*partition-between R h*))) ∈
nat-rel ×$_f$ *nat-rel* ×$_f$ ⟨*Id*⟩*list-rel* →$_f$ ⟨⟨*Id*⟩*list-rel* ×$_r$ *nat-rel*⟩*nres-rel*›
by (*intro frefI nres-relI*)
(*auto intro*: *partition-between-ref-partition-between*)

Example instantiation for pivot

definition *choose-pivot3-impl* **where**
‹*choose-pivot3-impl* = *choose-pivot3* (≤) *id*›

lemma *partition-between-ref-correct*:
assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
and *bounds*: ‹*hi* < *length xs*› ‹*lo* ≤ *hi*›
shows ‹*partition-between-ref R h lo hi xs* ≤ *SPEC* (*uncurry* (*partition-spec R h xs lo hi*))›
proof −
show *?thesis*
apply (*rule partition-between-ref-partition-between*[*THEN order-trans*])
using *bounds* **apply** (*rule partition-between-correct*[**where** *h*=*h*])
subgoal by (*rule trans*)
subgoal by (*rule lin*)
done
qed

term *quicksort*

Refined quicksort algorithm: We use the refined partition function.

definition *quicksort-ref* :: ‹- ⇒ - ⇒ *nat* × *nat* × *'a list* ⇒ *'a list nres*› **where**
‹*quicksort-ref R h* = (λ(*lo*,*hi*,*xs0*).
do {
RECT (λ*f* (*lo*,*hi*,*xs*). *do* {
ASSERT(*lo* ≤ *hi* ∧ *hi* < *length xs0* ∧ *mset xs* = *mset xs0*);
(*xs*, *p*) ← *partition-between-ref R h lo hi xs*; — This is the refined partition function. Note that we need the premises (trans,lin,bounds) here.
ASSERT(*mset xs* = *mset xs0* ∧ *p* ≥ *lo* ∧ *p* < *length xs0*);
xs ← (*if p*−*1*≤*lo then RETURN xs else f* (*lo*, *p*−*1*, *xs*));
ASSERT(*mset xs* = *mset xs0*);
if hi≤*p*+*1 then RETURN xs else f* (*p*+*1*, *hi*, *xs*)

```
  }) (lo,hi,xs0)
})›

lemma quicksort-ref-quicksort:
  assumes bounds: ‹hi < length xs› ‹lo ≤ hi› and
    trans: ‹⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)› and lin: ‹⋀x y. R (h x) (h y) ∨ R
(h y) (h x)›
  shows ‹quicksort-ref R h x0 ≤ ⇓ Id (quicksort R h x0)›
proof −
  have wf: ‹wf (measure (λ(lo, hi, xs). Suc hi − lo))›
    by auto
  have pre: ‹x0 = x0' ⟹ (x0, x0') ∈ Id ×r Id ×r ⟨Id⟩list-rel› for x0 x0' :: ‹nat × nat × 'b list›
    by auto
  have [refine0]: ‹(x1e = x1d) ⟹ (x1e,x1d) ∈ Id› for x1e x1d :: ‹'b list›
    by auto

  show ?thesis
    unfolding quicksort-def quicksort-ref-def
    apply (refine-vcg pre partition-between-ref-partition-between'[THEN fref-to-Down-curry2])
```

First assertion (premise for partition)

```
    subgoal
      by auto
```

First assertion (premise for partition)

```
    subgoal
      by auto
    subgoal
      by (auto dest: mset-eq-length)
    subgoal
      by (auto dest: mset-eq-length mset-eq-setD)
```

Correctness of the concrete partition function

```
    subgoal
      apply (simp, rule partition-between-ref-correct)
      subgoal by (rule trans)
      subgoal by (rule lin)
      subgoal by auto — first premise
      subgoal by auto — second premise
      done
    subgoal
      by (auto dest: mset-eq-length mset-eq-setD)
    subgoal by (auto simp: partition-spec-def isPartition-wrt-def)
    subgoal by (auto simp: partition-spec-def isPartition-wrt-def dest: mset-eq-length)
    subgoal
      by (auto dest: mset-eq-length mset-eq-setD)
    subgoal
      by (auto dest: mset-eq-length mset-eq-setD)
    subgoal
      by (auto dest: mset-eq-length mset-eq-setD)
    subgoal
      by (auto dest: mset-eq-length mset-eq-setD)

    by simp+
```

qed

— Sort the entire list
definition *full-quicksort* **where**
‹*full-quicksort R h xs* ≡ *if xs* = [] *then RETURN xs else quicksort R h* (*0*, *length xs* − *1*, *xs*)›

definition *full-quicksort-ref* **where**
‹*full-quicksort-ref R h xs* ≡
if List.null xs then RETURN xs
else quicksort-ref R h (*0*, *length xs* − *1*, *xs*)›

definition *full-quicksort-impl* :: ‹*nat list* ⇒ *nat list nres*› **where**
‹*full-quicksort-impl xs* = *full-quicksort-ref* (≤) *id xs*›

lemma *full-quicksort-ref-full-quicksort*:
assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹(*full-quicksort-ref R h*, *full-quicksort R h*) ∈
⟨*Id*⟩*list-rel* →*f* ⟨ ⟨*Id*⟩*list-rel*⟩*nres-rel*›
proof −
show *?thesis*
unfolding *full-quicksort-ref-def full-quicksort-def*
apply (*intro frefI nres-relI*)
apply (*auto intro*!: *quicksort-ref-quicksort*[*unfolded Down-id-eq*] *simp*: *List.null-def*)
subgoal by (*rule trans*)
subgoal using *lin* **by** *blast*
done
qed

lemma *sublist-entire*:
‹*sublist xs 0* (*length xs* − *1*) = *xs*›
by (*simp add*: *sublist-def*)

lemma *sorted-sublist-wrt-entire*:
assumes ‹*sorted-sublist-wrt R xs 0* (*length xs* − *1*)›
shows ‹*sorted-wrt R xs*›
proof −
have ‹*sorted-wrt R* (*sublist xs 0* (*length xs* − *1*))›
using *assms* **by** (*simp add*: *sorted-sublist-wrt-def*)
then show *?thesis*
by (*metis sublist-entire*)
qed

lemma *sorted-sublist-map-entire*:
assumes ‹*sorted-sublist-map R h xs 0* (*length xs* − *1*)›
shows ‹*sorted-wrt* (λ *x y*. *R* (*h x*) (*h y*)) *xs*›
proof −
show *?thesis*
using *assms* **by** (*rule sorted-sublist-wrt-entire*)
qed

Final correctness lemma

lemma *full-quicksort-correct-sorted*:
assumes

trans: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹*full-quicksort R h xs* $\leq \Downarrow$ *Id* (*SPEC*($\lambda xs'.$ *mset* $xs' =$ *mset xs* $\wedge$ *sorted-wrt* ($\lambda\ x\ y.\ R\ (h\ x)\ (h\ y)$) xs'))›
proof −
show *?thesis*
unfolding *full-quicksort-def*
apply (*refine-vcg*)
subgoal by *simp* — case xs=[]
subgoal by *simp* — case xs=[]

apply (*rule quicksort-correct*[*THEN order-trans*])
subgoal by (*rule trans*)
subgoal by (*rule lin*)
subgoal by *linarith*
subgoal by *simp*

apply (*simp add*: *Misc.subset-Collect-conv*, *intro allI impI conjI*)
subgoal
by (*auto simp add*: *quicksort-post-def*)
subgoal
apply (*rule sorted-sublist-map-entire*)
by (*auto simp add*: *quicksort-post-def dest*: *mset-eq-length*)
done
qed

lemma *full-quicksort-correct*:
assumes
trans: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and**
lin: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹*full-quicksort R h xs* $\leq \Downarrow$ *Id* (*SPEC*($\lambda xs'.$ *mset* $xs' =$ *mset xs*))›
by (*rule order-trans*[*OF full-quicksort-correct-sorted*])
(*use assms* **in** *auto*)

end
theory *Watched-Literals-Transition-System*
imports *More-Sepref.WB-More-Refinement CDCL.CDCL-W-Abstract-State*
CDCL.CDCL-W-Restart CDCL.Pragmatic-CDCL
begin

Chapter 2

Two-Watched Literals

2.1 Rule-based system

We define the calculus and map it to the pragmatic CDCL in order to inherit termination and correctness.
We initially inherited directly from CDCL, but this had two issues:

- First, there are inprocessing techniques we could not express (like subsumption-resolution, see the comments on PCDCL)...
- ... however, we tried to (had to) still express some of them (mostly about handling unit literals), leading to more complicated proofs. Additionally all of it was an afterthought (like the idea that clauses containing a true literal can be removed) and was never properly implemented (in that case, because there was no solution for clauses containing a false literal).

On a slightly less important note, but related, we support tautologies in our calculus, but this turned to be rather annoying in the generated code, because the length of clause does not fit in 32-bit native unsigned integers anymore (at least not with our encoding). We use 64-bit integer instead, but this lead to other work arounds (like position saving that actually save the position minus 2 to avoid overflow problems).
To overcome the issue, we decided to inherite not from CDCL, but from a different calculus devised exactly to express a more realistic calculus for SAT solvers.

2.1.1 Types and Transitions System

Types and accessing functions

datatype *'v twl-clause* =
  TWL-Clause (*watched*: *'v*) (*unwatched*: *'v*)

fun *clause* :: ‹*'a twl-clause* ⇒ *'a* :: {*plus*}› **where**
  ‹*clause* (*TWL-Clause W UW*) = *W* + *UW*›

abbreviation *clauses* :: ‹*'a* :: {*plus*} *twl-clause multiset* ⇒ *'a multiset*› **where**
  ‹*clauses C* ≡ *clause* ‘# *C*›

type-synonym *'v twl-cls* = ‹*'v clause twl-clause*›
type-synonym *'v twl-clss* = ‹*'v twl-cls multiset*›

type-synonym *'v clauses-to-update* = ‹(*'v literal* × *'v twl-cls*) *multiset*›
type-synonym *'v lit-queue* = ‹*'v literal multiset*›
type-synonym *'v twl-st* =
‹(*'v*, *'v clause*) *ann-lits* × *'v twl-clss* × *'v twl-clss* ×
'v clause option × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* ×
'v clauses-to-update × *'v lit-queue*›

fun *get-trail* :: ‹*'v twl-st* ⇒ (*'v*, *'v clause*) *ann-lit list*› **where**
‹*get-trail* (*M*, -, -, -, -, -, -, -) = *M*›

fun *clauses-to-update* :: ‹*'v twl-st* ⇒ (*'v literal* × *'v twl-cls*) *multiset*› **where**
‹*clauses-to-update* (-, -, -, -, -, -, -, -, -, -, *WS*, -) = *WS*›

fun *set-clauses-to-update* :: ‹(*'v literal* × *'v twl-cls*) *multiset* ⇒ *'v twl-st* ⇒ *'v twl-st*› **where**
‹*set-clauses-to-update WS* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, -, *Q*) =
(*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›

fun *literals-to-update* :: ‹*'v twl-st* ⇒ *'v lit-queue*› **where**
‹*literals-to-update* (-, -, -, -, -, -, -, -, -, -, -, *Q*) = *Q*›

fun *set-literals-to-update* :: ‹*'v lit-queue* ⇒ *'v twl-st* ⇒ *'v twl-st*› **where**
‹*set-literals-to-update Q* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, -) =
(*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›

fun *set-conflict* :: ‹*'v clause* ⇒ *'v twl-st* ⇒ *'v twl-st*› **where**
‹*set-conflict D* (*M*, *N*, *U*, -, *NE*, *UE*, *NS*, *US*, *WS*, *N0*, *U0*, *Q*) =
(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *WS*, *N0*, *U0*, *Q*)›

fun *get-conflict* :: ‹*'v twl-st* ⇒ *'v clause option*› **where**
‹*get-conflict* (*M*, *N*, *U*, *D*, *NE*, *UE*, *N0*, *U0*, *WS*, *Q*) = *D*›

fun *get-clauses* :: ‹*'v twl-st* ⇒ *'v twl-clss*› **where**
‹*get-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *N* + *U*›

fun *unit-clss* :: ‹*'v twl-st* ⇒ *'v clause multiset*› **where**
‹*unit-clss* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) = *NE* + *UE*›

fun *unit-init-clauses* :: ‹*'v twl-st* ⇒ *'v clauses*› **where**
‹*unit-init-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) = *NE*›

fun *subsumed-learned-clss* :: ‹*'v twl-st* ⇒ *'v clause multiset*› **where**
‹*subsumed-learned-clss* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) = *US*›

fun *subsumed-init-clauses* :: ‹*'v twl-st* ⇒ *'v clauses*› **where**
‹*subsumed-init-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *NS*›

fun *get-all-init-clss* :: ‹*'v twl-st* ⇒ *'v clause multiset*› **where**
‹*get-all-init-clss* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *clause* ‘# *N* + *NE* + *NS* +
N0›

fun *get-learned-clss* :: ‹*'v twl-st* ⇒ *'v twl-clss*› **where**
‹*get-learned-clss* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *U*›

fun *subsumed-learned-clauses* :: ‹*'v twl-st* ⇒ *'v clauses*› **where**
‹*subsumed-learned-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *US*›

fun *subsumed-clauses* :: ‹*'v twl-st ⇒ 'v clauses*› **where**
‹*subsumed-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *NS* + *US*›

fun *get-init-learned-clss* :: ‹*'v twl-st ⇒ 'v clauses*› **where**
‹*get-init-learned-clss* (-, *N*, *U*, -, -, *UE*, *NS*, *US*, -) = *UE*›

fun *get-all-learned-clss* :: ‹*'v twl-st ⇒ 'v clauses*› **where**
‹*get-all-learned-clss* (-, *N*, *U*, -, -, *UE*, *NS*, *US*, *N0*, *U0*, -) = *clause* ‘# *U* + *UE* + *US* + *U0*›

fun *get-all-clss* :: ‹*'v twl-st ⇒ 'v clause multiset*› **where**
‹*get-all-clss* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
clause ‘# *N* + *NE* + *NS* + *N0* + *clause* ‘# *U* + *UE* + *US* + *U0*›

fun *get-init-clauses0* :: ‹*'v twl-st ⇒ 'v clauses*› **where**
‹*get-init-clauses0* (-, *N*, *U*, -, -, *UE*, *NS*, *US*, *N0*, *U0*,-, -) = *N0*›

fun *get-learned-clauses0* :: ‹*'v twl-st ⇒ 'v clauses*› **where**
‹*get-learned-clauses0* (-, *N*, *U*, -, -, *UE*, *NS*, *US*, *N0*, *U0*,-, -) = *U0*›

fun *get-all-clauses0* :: ‹*'v twl-st ⇒ 'v clauses*› **where**
‹*get-all-clauses0* (-, *N*, *U*, -, -, *UE*, *NS*, *US*, *N0*, *U0*, -, -) = *N0* + *U0*›

fun *update-clause* **where**
‹*update-clause* (*TWL-Clause W UW*) *L L'* =
TWL-Clause (*add-mset L'* (*remove1-mset L W*)) (*add-mset L* (*remove1-mset L' UW*))›

When updating clause, we do it non-deterministically: in case of duplicate clause in the two sets, one of the two can be updated (and it does not matter), contrary to an if-condition. In later refinement, we know where the clause comes from and update it.

inductive *update-clauses* ::
‹*'a multiset twl-clause multiset* × *'a multiset twl-clause multiset ⇒*
'a multiset twl-clause ⇒ 'a ⇒ 'a ⇒
'a multiset twl-clause multiset × *'a multiset twl-clause multiset ⇒ bool*› **where**
‹*D* ∈# *N* ⟹ *update-clauses* (*N*, *U*) *D L L'* (*add-mset* (*update-clause D L L'*) (*remove1-mset D N*), *U*)›
| ‹*D* ∈# *U* ⟹ *update-clauses* (*N*, *U*) *D L L'* (*N*, *add-mset* (*update-clause D L L'*) (*remove1-mset D U*))›

inductive-cases *update-clausesE*: ‹*update-clauses* (*N*, *U*) *D L L'* (*N'*, *U'*)›

The Transition System

We ensure that there are always *2* watched literals and that there are different. All clauses containing a single literal are put in *NE* or *UE*.

inductive *cdcl-twl-cp* :: ‹*'v twl-st ⇒ 'v twl-st ⇒ bool*› **where**
pop:
‹*cdcl-twl-cp* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *add-mset L Q*)
(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#(*L*, *C*)|*C* ∈# *N* + *U*. *L* ∈# *watched C*#}, *Q*)› |
propagate:
‹*cdcl-twl-cp* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*)
(*Propagated L'* (*clause D*) # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* (−*L'*) *Q*)›
if
‹*watched D* = {#*L*, *L'*#}› **and** ‹*undefined-lit M L'*› **and** ‹∀ *L* ∈# *unwatched D*. −*L* ∈ *lits-of-l M*› |
conflict:
‹*cdcl-twl-cp* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*)

(*M*, *N*, *U*, *Some* (*clause D*), *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
if ‹*watched D* = {#*L*, *L*′#}› **and** ‹−*L*′ ∈ *lits-of-l M*› **and** ‹∀ *L* ∈# *unwatched D*. −*L* ∈ *lits-of-l M*› |
delete-from-working:
‹*cdcl-twl-cp* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*)
(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
if ‹*L*′ ∈# *clause D*› **and** ‹*L*′ ∈ *lits-of-l M*› |
update-clause:
‹*cdcl-twl-cp* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*)
(*M*, *N*′, *U*′, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
if ‹*watched D* = {#*L*, *L*′#}› **and** ‹−*L* ∈ *lits-of-l M*› **and** ‹*L*′ ∉ *lits-of-l M*› **and**
‹*K* ∈# *unwatched D*› **and** ‹*undefined-lit M K* ∨ *K* ∈ *lits-of-l M*› **and**
‹*update-clauses* (*N*, *U*) *D L K* (*N*′, *U*′)›
— The condition − *L* ∈ *lits-of-l M* is already implied by *valid* invariant.

inductive-cases *cdcl-twl-cpE*: ‹*cdcl-twl-cp S T*›

We do not care about the *literals-to-update* literals.

inductive *cdcl-twl-o* :: ‹′*v twl-st* ⇒ ′*v twl-st* ⇒ *bool*› **where**
decide:
‹*cdcl-twl-o* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})
(*Decided L* # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#−*L*#})›
if ‹*undefined-lit M L*› **and** ‹*atm-of L* ∈ *atms-of-mm* (*clause* ‘# *N* + *NE* + *NS* + *N0*)›
| *skip*:
‹*cdcl-twl-o* (*Propagated L C*′ # *M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})
(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
if ‹−*L* ∉# *D*› **and** ‹*D* ≠ {#}›
| *resolve*:
‹*cdcl-twl-o* (*Propagated L C* # *M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})
(*M*, *N*, *U*, *Some* (*cdcl*$_W$*-restart-mset.resolve-cls L D C*), *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
if ‹−*L* ∈# *D*› **and**
‹*get-maximum-level* (*Propagated L C* # *M*) (*remove1-mset* (−*L*) *D*) = *count-decided M*›
| *backtrack-unit-clause*:
‹*cdcl-twl-o* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})
(*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#−*L*#})›
if
‹*L* ∈# *D*› **and**
‹(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
‹*get-level M L* = *count-decided M*› **and**
‹*get-level M L* = *get-maximum-level M D*′› **and**
‹*get-maximum-level M* (*D*′ − {#*L*#}) ≡ *i*› **and**
‹*get-level M K* = *i* + 1›
‹*D*′ = {#*L*#}› **and**
‹*D*′ ⊆# *D*› **and**
‹*clause* ‘# (*N* + *U*) + *NE* + *UE* + *NS* + *US* + *N0* + *U0* ⊨*pm D*′›
| *backtrack-nonunit-clause*:
‹*cdcl-twl-o* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})
(*Propagated L D*′ # *M1*, *N*, *add-mset* (*TWL-Clause* {#*L*, *L*′#} (*D*′ − {#*L*, *L*′#})) *U*, *None*, *NE*, *UE*,
NS, *US*, *N0*, *U0*, {#}, {#−*L*#})›
if
‹*L* ∈# *D*› **and**
‹(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
‹*get-level M L* = *count-decided M*› **and**
‹*get-level M L* = *get-maximum-level M D*′› **and**
‹*get-maximum-level M* (*D*′ − {#*L*#}) ≡ *i*› **and**

‹*get-level M K = i + 1*›
‹$D' \neq \{\#L\#\}$› **and**
‹$D' \subseteq\# D$› **and**
‹*clause* '$\# (N + U) + NE + UE + NS + US + N0 + U0 \models pm\ D'$› **and**
‹$L \in\# D'$›
‹$L' \in\# D'$› **and** — L' is the new watched literal
‹*get-level M L' = i*›

inductive-cases *cdcl-twl-oE*: ‹*cdcl-twl-o S T*›

inductive *cdcl-twl-stgy* :: ‹'*v twl-st* ⇒ '*v twl-st* ⇒ *bool*› **for** *S* :: ‹'*v twl-st*› **where**
cp: ‹*cdcl-twl-cp S S'* ⟹ *cdcl-twl-stgy S S'*› |
other': ‹*cdcl-twl-o S S'* ⟹ *cdcl-twl-stgy S S'*›

inductive-cases *cdcl-twl-stgyE*: ‹*cdcl-twl-stgy S T*›

2.1.2 Definition of the Two-watched Literals Invariants

Definitions

The structural invariants states that there are at most two watched elements, that the watched literals are distinct, and that there are 2 watched literals if there are at least than two different literals in the full clauses.

primrec *struct-wf-twl-cls* :: ‹'*v multiset twl-clause* ⇒ *bool*› **where**
‹*struct-wf-twl-cls* (*TWL-Clause W UW*) ⟷
size W = 2 ∧ *distinct-mset* (*W + UW*)›

fun $pstate_W$*-of* :: ‹'*v twl-st* ⇒ '*v prag-st*› **where**
‹$pstate_W$*-of* (*M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *Q*) =
(*M*, *clause* '$\# N$, *clause* '$\# U$, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*)›

named-theorems *twl-st* ‹*Conversions simp rules*›

lemma [*twl-st*,*simp*]: ‹*pget-trail* ($pstate_W$*-of S'*) = *get-trail S'*›
by (*cases S'*) (*auto simp*: *trail.simps*)

lemma [*twl-st*,*simp*]: ‹*pget-conflict* ($pstate_W$*-of S'*) = *get-conflict S'*›
by (*cases S'*) (*auto simp*: *conflicting.simps*)

lemma [*twl-st*,*simp*]: ‹$cdcl_W$*-restart-mset.clauses* (*state-of* ($pstate_W$*-of S'*)) = *get-all-clss S'*›
by (*cases S'*) (*auto simp*: *clauses-def*)

TODO: could also be an abbreviation.

definition $state_W$*-of* :: ‹'*v twl-st* ⇒ '*v* $cdcl_W$*-restart-mset*› **where**
‹$state_W$*-of S* = *state-of* ($pstate_W$*-of S*)›

lemma *subsumed-clauses-simps*[*twl-st*,*simp*]:
‹*subsumed-init-clauses* (*set-clauses-to-update K S*) = *subsumed-init-clauses S*›
‹*subsumed-learned-clauses* (*set-clauses-to-update K S*) = *subsumed-learned-clauses S*›
‹*subsumed-clauses* (*set-clauses-to-update K S*) = *subsumed-clauses S*›
by (*cases S*; *auto*; *fail*)+

lemma [*twl-st*,*simp*]:
‹*trail* (*state-of* ($pstate_W$*-of S*)) = *get-trail S*›

‹*trail* (*state*$_W$ *-of S*) = *get-trail S*›
‹*conflicting* (*state-of* (*pstate*$_W$ *-of S*)) = *get-conflict S*›
‹*conflicting* (*state*$_W$ *-of S*) = *get-conflict S*›
‹*conflicting* (*state-of T*) = *pget-conflict T*›
by (*cases S*; *cases T*; *auto simp*: *state*$_W$ *-of-def*; *fail*)+

The invariant on the clauses is the following:

- the structure is correct (the watched part is of length exactly two).
- if we do not have to update the clause, then the invariant holds.

definition *twl-is-an-exception* :: ‹'*a multiset twl-clause* ⇒ '*a multiset* ⇒
('*b* × '*a multiset twl-clause*) *multiset* ⇒ *bool*›
where
‹*twl-is-an-exception C Q WS* ⟷
(∃ *L*. *L* ∈# *Q* ∧ *L* ∈# *watched C*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)›

definition *is-blit* :: ‹('*a*, '*b*) *ann-lits* ⇒ '*a clause* ⇒ '*a literal* ⇒ *bool*›**where**
[*simp*]: ‹*is-blit M D L* ⟷ (*L* ∈# *D* ∧ *L* ∈ *lits-of-l M*)›

definition *has-blit* :: ‹('*a*, '*b*) *ann-lits* ⇒ '*a clause* ⇒ '*a literal* ⇒ *bool*›**where**
‹*has-blit M D L'* ⟷ (∃ *L*. *is-blit M D L* ∧ *get-level M L* ≤ *get-level M L'*)›

This invariant state that watched literals are set at the end and are not swapped with an unwatched literal later.

fun *twl-lazy-update* :: ‹('*a*, '*b*) *ann-lits* ⇒ '*a twl-cls* ⇒ *bool*› **where**
‹*twl-lazy-update M* (*TWL-Clause W UW*) ⟷
(∀ *L*. *L* ∈# *W* ⟶ −*L* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*W*+*UW*) *L* ⟶
(∀ *K* ∈# *UW*. *get-level M L* ≥ *get-level M K* ∧ −*K* ∈ *lits-of-l M*))›

If one watched literals has been assigned to false (− *L* ∈ *lits-of-l M*) and the clause has not yet been updated (*L'* ∉ *lits-of-l M*: it should be removed either by updating *L*, propagating *L'*, or marking the conflict), then the literals *L* is of maximal level.

fun *watched-literals-false-of-max-level* :: ‹('*a*, '*b*) *ann-lits* ⇒ '*a twl-cls* ⇒ *bool*› **where**
‹*watched-literals-false-of-max-level M* (*TWL-Clause W UW*) ⟷
(∀ *L*. *L* ∈# *W* ⟶ −*L* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*W*+*UW*) *L* ⟶
get-level M L = *count-decided M*)›

This invariants talks about the enqueued literals:

- the working stack contains a single literal;
- the working stack and the *literals-to-update* literals are false with respect to the trail and there are no duplicates;
- and the latter condition holds even when *WS* = {#}.

fun *no-duplicate-queued* :: ‹'*v twl-st* ⇒ *bool*› **where**
‹*no-duplicate-queued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *C C'*. *C* ∈# *WS* ⟶ *C'* ∈# *WS* ⟶ *fst C* = *fst C'*) ∧
(∀ *C*. *C* ∈# *WS* ⟶ *add-mset* (*fst C*) *Q* ⊆# *uminus* ‘# *lit-of* ‘# *mset M*) ∧
Q ⊆# *uminus* ‘# *lit-of* ‘# *mset M*›

lemma *no-duplicate-queued-alt-def*:
  ‹*no-duplicate-queued S* =
   ((∀ *C C'*. *C* ∈# *clauses-to-update S* ⟶ *C'* ∈# *clauses-to-update S* ⟶ *fst C* = *fst C'*) ∧
    (∀ *C*. *C* ∈# *clauses-to-update S* ⟶
      add-mset (*fst C*) (*literals-to-update S*) ⊆# *uminus* ‘# *lit-of* ‘# *mset* (*get-trail S*)) ∧
    literals-to-update S ⊆# *uminus* ‘# *lit-of* ‘# *mset* (*get-trail S*))›
  by (*cases S*) *auto*

fun *distinct-queued* :: ‹'*v twl-st* ⇒ *bool*› **where**
‹*distinct-queued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
  distinct-mset Q ∧
  (∀ *L C*. *count WS* (*L*, *C*) ≤ *count* (*N* + *U*) *C*)›

These are the conditions to indicate that the 2-WL invariant does not hold and is not *literals-to-update*.

fun *clauses-to-update-prop* **where**
  ‹*clauses-to-update-prop Q M* (*L*, *C*) ⟷
     (*L* ∈# *watched C* ∧ −*L* ∈ *lits-of-l M* ∧ *L* ∉# *Q* ∧ ¬*has-blit M* (*clause C*) *L*)›
declare *clauses-to-update-prop.simps*[*simp del*]

This invariants talks about the enqueued literals:

- all clauses that should be updated are in *WS* and are repeated often enough in it.
- if *WS* = {#}, then there are no clauses to updated that is not enqueued;
- all clauses to updated are either in *WS* or *Q*.

  The first two conditions are written that way to please Isabelle.

fun *clauses-to-update-inv* :: ‹'*v twl-st* ⇒ *bool*› **where**
  ‹*clauses-to-update-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
    (∀ *L C*. ((*L*, *C*) ∈# *WS* ⟶ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*)) ∧
    (∀ *L*. *WS* = {#} ⟶ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} = {#}) ∧
    (∀ *L C*. *C* ∈# *N* + *U* ⟶ *L* ∈# *watched C* ⟶ −*L* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*clause C*) *L* ⟶
      (*L*, *C*) ∉# *WS* ⟶ *L* ∈# *Q*)›
| ‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) ⟷ *True*›

This is the invariant of the 2WL structure: if one watched literal is false, then all unwatched are false.

fun *twl-exception-inv* :: ‹'*v twl-st* ⇒ '*v twl-cls* ⇒ *bool*› **where**
  ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) *C* ⟷
    (∀ *L*. *L* ∈# *watched C* ⟶ −*L* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*clause C*) *L* ⟶
      L ∉# *Q* ⟶ (*L*, *C*) ∉# *WS* ⟶
      (∀ *K* ∈# *unwatched C*. −*K* ∈ *lits-of-l M*))›
| ‹*twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) *C* ⟷ *True*›

declare *twl-exception-inv.simps*[*simp del*]

fun *twl-st-exception-inv* :: ‹'*v twl-st* ⇒ *bool*› **where**
‹*twl-st-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
  (∀ *C* ∈# *N* + *U*. *twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) *C*)›

Candidats for propagation (i.e., the clause where only one literals is non assigned) are enqueued.

fun *propa-cands-enqueued* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*propa-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *L C*. *C* ∈# *N*+*U* ⟶ *L* ∈# *clause C* ⟶ *M* ⊨*as CNot* (*remove1-mset L* (*clause C*)) ⟶
undefined-lit M L ⟶
(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))›
| ‹*propa-cands-enqueued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷ *True*›

fun *confl-cands-enqueued* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *C* ∈# *N* + *U*. *M* ⊨*as CNot* (*clause C*) ⟶
(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))›
| ‹*confl-cands-enqueued* (*M*, *N*, *U*, *Some* -, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
True›

fun *propa-confl-cands-enqueued* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *C* ∈# *N* + *U*. ∀ *L* ∈# *clause C*. *M* ⊨*as CNot* (*clause C* − {#*L*#}) ⟶ *L* ∉ *lits-of-l M* ⟶
(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))›
| ‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *Some* -, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
True›

lemma *propa-confl-cands-enqueued-propa-confl-enqueued*:
assumes ‹∀ *C* ∈# *get-clauses S*. *struct-wf-twl-cls C*› **and** ‹*no-dup* (*get-trail S*)›
shows ‹*propa-confl-cands-enqueued S* ⟷ *propa-cands-enqueued S* ∧ *confl-cands-enqueued S*›
using *assms*
apply (*cases S*; *cases* ‹*get-conflict S*›)
apply (*auto dest*!: *multi-member-split simp*: *Decided-Propagated-in-iff-in-lits-of-l no-dup-consistentD*)
apply (*case-tac C*; *case-tac* ‹*watched C*›)
apply (*clarsimp-all simp*: *imp-conjR all-conj-distrib no-dup-consistentD*)
apply (*case-tac C*; *case-tac* ‹*watched C*›)
apply (*clarsimp-all simp*: *imp-conjR all-conj-distrib no-dup-consistentD*)
done

This invariant talk about the decomposition of the trail and the invariants that holds in these states.

fun *past-invs* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *M1 M2 K*. *M* = *M2* @ *Decided K* # *M1* ⟶ (
(∀ *C* ∈# *N* + *U*. *twl-lazy-update M1 C* ∧
watched-literals-false-of-max-level M1 C ∧
twl-exception-inv (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*) ∧
confl-cands-enqueued (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) ∧
propa-cands-enqueued (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) ∧
clauses-to-update-inv (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})))›
declare *past-invs.simps*[*simp del*]

fun *twl-st-inv* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *C* ∈# *N* + *U*. *struct-wf-twl-cls C*) ∧
(∀ *C* ∈# *N* + *U*. *D* = *None* ⟶ ¬*twl-is-an-exception C Q WS* ⟶ (*twl-lazy-update M C*)) ∧
(∀ *C* ∈# *N* + *U*. *D* = *None* ⟶ *watched-literals-false-of-max-level M C*)›

lemma *twl-st-inv-alt-def*:
‹*twl-st-inv S* ⟷
(∀ *C* ∈# *get-clauses S*. *struct-wf-twl-cls C*) ∧
(∀ *C* ∈# *get-clauses S*. *get-conflict S* = *None* ⟶

$\neg$*twl-is-an-exception C* (*literals-to-update S*) (*clauses-to-update S*) $\longrightarrow$
(*twl-lazy-update* (*get-trail S*) *C*)) $\wedge$
($\forall$ *C* $\in\#$ *get-clauses S*. *get-conflict S* = *None* $\longrightarrow$
watched-literals-false-of-max-level (*get-trail S*) *C*)›
by (*cases S*) (*auto simp*: *twl-st-inv.simps*)

literals-to-update literals are of maximum level and their negation is in the trail.

fun *valid-enqueued* :: ‹*'v twl-st* $\Rightarrow$ *bool*› **where**
‹*valid-enqueued* (*M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) $\longleftrightarrow$
($\forall$(*L*, *C*) $\in\#$ *WS*. *L* $\in\#$ *watched C* $\wedge$ *C* $\in\#$ *N* + *U* $\wedge$ $-L \in$ *lits-of-l M* $\wedge$
get-level M L = *count-decided M*) $\wedge$
($\forall$ *L* $\in\#$ *Q*. $-L \in$ *lits-of-l M* $\wedge$ *get-level M L* = *count-decided M*)›

Putting invariants together:

definition *twl-struct-invs* :: ‹*'v twl-st* $\Rightarrow$ *bool*› **where**
‹*twl-struct-invs S* $\longleftrightarrow$
(*twl-st-inv S* $\wedge$
valid-enqueued S $\wedge$
pcdcl-all-struct-invs (*pstate*$_W$*-of S*) $\wedge$
cdcl$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of S*) $\wedge$
twl-st-exception-inv S $\wedge$
no-duplicate-queued S $\wedge$
distinct-queued S $\wedge$
confl-cands-enqueued S $\wedge$
propa-cands-enqueued S $\wedge$
(*get-conflict S* $\neq$ *None* $\longrightarrow$ *clauses-to-update S* = {#} $\wedge$ *literals-to-update S* = {#}) $\wedge$
clauses-to-update-inv S $\wedge$
past-invs S)
›

definition *twl-stgy-invs* :: ‹*'v twl-st* $\Rightarrow$ *bool*› **where**
‹*twl-stgy-invs S* $\longleftrightarrow$
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant* (*state*$_W$*-of S*) $\wedge$
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0* (*state*$_W$*-of S*)›

Initial properties

lemma *twl-is-an-exception-add-mset-to-queue*: ‹*twl-is-an-exception C* (*add-mset L Q*) *WS* $\longleftrightarrow$
(*twl-is-an-exception C Q WS* $\vee$ (*L* $\in\#$ *watched C*))›
unfolding *twl-is-an-exception-def* **by** *auto*

lemma *twl-is-an-exception-add-mset-to-clauses-to-update*:
‹*twl-is-an-exception C Q* (*add-mset* (*L*, *D*) *WS*) $\longleftrightarrow$ (*twl-is-an-exception C Q WS* $\vee$ *C* = *D*)›
unfolding *twl-is-an-exception-def* **by** *auto*

lemma *twl-is-an-exception-empty*[*simp*]: ‹$\neg$*twl-is-an-exception C* {#} {#}›
unfolding *twl-is-an-exception-def* **by** *auto*

lemma *twl-inv-empty-trail*:
shows
‹*watched-literals-false-of-max-level* [] *C*› **and**
‹*twl-lazy-update* [] *C*›
by (*solves* ‹*cases C*; *auto*›)+

lemma *clauses-to-update-inv-cases*[*case-names WS-nempty WS-empty Q*]:
assumes

‹⋀L C. (L, C) ∈# WS ⟹ {#(L, C)| C ∈# N + U. clauses-to-update-prop Q M (L, C)#} ⊆# WS› **and**
‹⋀L. WS = {#} ⟹ {#(L, C)| C ∈# N + U. clauses-to-update-prop Q M (L, C)#} = {#}› **and**
‹⋀L C. C ∈# N + U ⟹ L ∈# watched C ⟹ −L ∈ lits-of-l M ⟹ ¬has-blit M (clause C) L ⟹
(L, C) ∉# WS ⟹ L ∈# Q›
shows
‹clauses-to-update-inv (M, N, U, None, NE, UE, NS, US, N0, U0, WS, Q)›
using *assms* **unfolding** *clauses-to-update-inv.simps* **by** *blast*

lemma
assumes ‹⋀C. C ∈# N + U ⟹ struct-wf-twl-cls C›
shows
twl-st-inv-empty-trail: ‹twl-st-inv ([], N, U, C, NE, UE, NS, US, N0, U0, WS, Q)›
by (*auto simp*: *assms twl-inv-empty-trail*)

lemma
shows
no-duplicate-queued-no-queued: ‹no-duplicate-queued (M, N, U, D, NE, UE, NS, US, N0, U0, {#}, {#})› **and**
no-distinct-queued-no-queued: ‹distinct-queued ([], N, U, D, NE, UE, NS, US, N0, U0, {#}, {#})›
by *auto*

lemma *twl-st-inv-add-mset-clauses-to-update*:
assumes ‹D ∈# N + U›
shows ‹twl-st-inv (M, N, U, None, NE, UE, NS, US, N0, U0, WS, Q)
⟷ twl-st-inv (M, N, U, None, NE, UE, NS, US, N0, U0, add-mset (L, D) WS, Q) ∧
(¬ twl-is-an-exception D Q WS ⟶twl-lazy-update M D)›
using *assms* **by** (*auto simp*: *twl-is-an-exception-add-mset-to-clauses-to-update*)

lemma *twl-st-simps*:
‹twl-st-inv (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) ⟷
(∀ C ∈# N + U. struct-wf-twl-cls C ∧
(D = None ⟶ (¬twl-is-an-exception C Q WS ⟶ twl-lazy-update M C) ∧
watched-literals-false-of-max-level M C))›
unfolding *twl-st-inv.simps* **by** *fast*

lemma *propa-cands-enqueued-unit-clause*:
‹propa-cands-enqueued (M, N, U, C, add-mset L NE, UE, NS, US, N0, U0, WS, Q) ⟷
propa-cands-enqueued (M, N, U, C, {#}, {#}, NS, US, N0, U0, WS, Q)›
‹propa-cands-enqueued (M, N, U, C, NE, add-mset L UE, NS, US, N0, U0, WS, Q) ⟷
propa-cands-enqueued (M, N, U, C, {#}, {#}, NS, US, N0, U0, WS, Q)›
by (*cases C*; *auto*)+

lemma *past-invs-enqueud*: ‹past-invs (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) ⟷
past-invs (M, N, U, D, NE, UE, NS, US, N0, U0, {#}, {#})›
unfolding *past-invs.simps* **by** *simp*

lemma *confl-cands-enqueued-unit-clause*:
‹confl-cands-enqueued (M, N, U, C, add-mset L NE, UE, NS, US, N0, U0, WS, Q) ⟷
confl-cands-enqueued (M, N, U, C, {#}, {#}, NS, US, N0, U0, WS, Q)›
‹confl-cands-enqueued (M, N, U, C, NE, add-mset L UE, NS, US, N0, U0, WS, Q) ⟷
confl-cands-enqueued (M, N, U, C, {#}, {#}, NS, US, N0, U0, WS, Q)›
by (*cases C*; *auto*)+

lemma *twl-inv-decomp*:

```
  assumes
    lazy: ‹twl-lazy-update M C› and
    decomp: ‹(Decided K # M1, M2) ∈ set (get-all-ann-decomposition M)› and
    n-d: ‹no-dup M›
  shows
    ‹twl-lazy-update M1 C›
proof −
  obtain W UW where C: ‹C = TWL-Clause W UW› by (cases C)
  obtain M3 where M: ‹M = M3 @ M2 @ Decided K # M1›
    using decomp by blast
  define M' where M': ‹M' = M3 @ M2 @ [Decided K]›
  have MM': ‹M = M' @ M1›
    by (auto simp: M M')
  have lev-M-M1: ‹get-level M L = get-level M1 L› if ‹L ∈ lits-of-l M1› for L
  proof −
    have LM: ‹L ∈ lits-of-l M›
      using that unfolding M by auto
    have ‹undefined-lit M' L›
      by (rule no-dup-append-in-atm-notin)
        (use that n-d in ‹auto simp: M M' defined-lit-map›)
    then show lev-L-M1: ‹get-level M L = get-level M1 L›
      using that n-d by (auto simp: M image-Un M')
  qed

  show ‹twl-lazy-update M1 C›
    unfolding C twl-lazy-update.simps
  proof (intro allI impI)
    fix L
    assume
      W: ‹L ∈# W› and
      uL: ‹− L ∈ lits-of-l M1› and
      L': ‹¬has-blit M1 (W+UW) L›

    then have lev-L-M1: ‹get-level M L = get-level M1 L›
      using uL n-d lev-M-M1[of ‹−L›] by auto

    have L'M: ‹¬has-blit M (W+UW) L›
    proof (rule ccontr)
      assume ‹¬ ?thesis›
      then obtain L' where
        b: ‹is-blit M (W+UW) L'› and
        lev-L'-L: ‹get-level M L' ≤ get-level M L›unfolding has-blit-def by auto
      then have L'M': ‹L' ∈ lits-of-l M'›
        using L' MM' W lev-L-M1 lev-M-M1 unfolding has-blit-def by auto
      moreover {
        have ‹atm-of L' ∈ atm-of ‘ lits-of-l M'›
          using L'M' by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
        moreover have ‹Decided K ∈set (dropWhile (λS. atm-of (lit-of S) ≠ atm-of K') M')›
          if ‹K' ∈ lits-of-l M'› for K'
          unfolding M' append-assoc[symmetric] by (rule last-in-set-dropWhile)
            (use that in ‹auto simp: lits-of-def M' MM'›)
        ultimately have ‹get-level M L' > count-decided M1›
          unfolding MM' by (force simp: filter-empty-conv get-level-def count-decided-def
            lits-of-def) }
      ultimately show False
        using lev-M-M1[of ‹−L›] uL count-decided-ge-get-level[of M1 ‹−L›] lev-L'-L by auto
```

```
    qed

    show ‹∀ K∈#UW. get-level M1 K ≤ get-level M1 L ∧ −K ∈ lits-of-l M1›
    proof clarify
      fix K''
      assume ‹K'' ∈# UW›
      then have
        lev-K'-L: ‹get-level M K'' ≤ get-level M L› and
        uK'-M: ‹−K'' ∈ lits-of-l M›
        using lazy W uL L'M unfolding C MM' by auto
      then have uK'-M1: ‹− K'' ∈ lits-of-l M1›
        using uK'-M unfolding  M apply (auto simp: get-level-append-if
            split: if-splits)
        using M' MM' n-d uL count-decided-ge-get-level[of M1 L]
        by (auto dest: defined-lit-no-dupD in-lits-of-l-defined-litD
            simp: get-level-cons-if atm-of-eq-atm-of
            split: if-splits)
      have ‹get-level M K'' = get-level M1 K''›
      proof (rule ccontr, cases ‹defined-lit M' K''›)
        case False
        moreover assume ‹get-level M K'' ≠ get-level M1 K''›
        ultimately show False unfolding MM' by auto
      next
        case True
        assume K'': ‹get-level M K'' ≠ get-level M1 K''›
        have ‹get-level M' K'' = 0›
        proof −
          have a1: ‹get-level M' K'' + count-decided M1 ≤ get-level M1 L›
            using lev-K'-L unfolding lev-L-M1 unfolding MM' get-level-skip-end[OF True] .
          then have ‹count-decided M1 ≤ get-level M1 L›
            by linarith
          then have ‹get-level M1 L = count-decided M1›
            using count-decided-ge-get-level le-antisym by blast
          then show ?thesis
            using a1 by linarith
        qed
        moreover have ‹Decided K ∈ set (dropWhile (λS. atm-of (lit-of S) ≠ atm-of K'') M')›
          unfolding M' append-assoc[symmetric] by (rule last-in-set-dropWhile
            (use True in ‹auto simp: lits-of-def M' MM' defined-lit-map›))
        ultimately show False
          by (auto simp: M' filter-empty-conv get-level-def)
      qed
      then show ‹get-level M1 K'' ≤ get-level M1 L ∧ −K'' ∈ lits-of-l M1›
        using lev-M-M1[OF uL] lev-K'-L uK'-M uK'-M1 by auto
    qed
  qed
qed

declare twl-st-inv.simps[simp del]

lemma has-blit-Cons[simp]:
  assumes blit: ‹has-blit M C L› and n-d: ‹no-dup (K # M)›
  shows ‹has-blit (K # M) C L›
proof −
  obtain L' where
    ‹is-blit M C L'› and
```

```
    ‹get-level M L' ≤ get-level M L›
    using blit unfolding has-blit-def by auto
  then have
    ‹is-blit (K # M) C L'› and
    ‹get-level (K # M) L' ≤ get-level (K # M) L›
    using n-d by (auto simp add: has-blit-def get-level-cons-if atm-of-eq-atm-of
      dest: in-lits-of-l-defined-litD)
  then show ?thesis
    unfolding has-blit-def by blast
qed


lemma is-blit-Cons:
  ‹is-blit (K # M) C L ⟷ (L = lit-of K ∧ lit-of K ∈# C) ∨ is-blit M C L›
  by (auto simp: has-blit-def)

lemma no-has-blit-propagate:
  ‹¬has-blit (Propagated L D # M) (W + UW) La ⟹
    undefined-lit M L ⟹ no-dup M ⟹ ¬has-blit M (W + UW) La›
  apply (auto simp: has-blit-def get-level-cons-if
    dest: in-lits-of-l-defined-litD
     split:  cong: if-cong)
  apply (smt atm-lit-of-set-lits-of-l count-decided-ge-get-level defined-lit-map image-eqI)
  by (smt atm-lit-of-set-lits-of-l count-decided-ge-get-level defined-lit-map image-eqI)

lemma no-has-blit-propagate':
  ‹¬has-blit (Propagated L D # M) (clause C) La ⟹
    undefined-lit M L ⟹ no-dup M ⟹ ¬has-blit M (clause C) La›
  using no-has-blit-propagate[of L D M ‹watched C› ‹unwatched C›]
  by (cases C) auto


lemma no-has-blit-decide:
  ‹¬has-blit (Decided L # M) (W + UW) La ⟹
    undefined-lit M L ⟹ no-dup M ⟹ ¬has-blit M (W + UW) La›
  apply (auto simp: has-blit-def get-level-cons-if
    dest: in-lits-of-l-defined-litD
     split:  cong: if-cong)
  apply (smt count-decided-ge-get-level defined-lit-map in-lits-of-l-defined-litD le-SucI)
  apply (smt count-decided-ge-get-level defined-lit-map in-lits-of-l-defined-litD le-SucI)
  done

lemma no-has-blit-decide':
  ‹¬has-blit (Decided L # M) (clause C) La ⟹
    undefined-lit M L ⟹ no-dup M ⟹ ¬has-blit M (clause C) La›
  using no-has-blit-decide[of L M ‹watched C› ‹unwatched C›]
  by (cases C) auto

lemma twl-lazy-update-Propagated:
  assumes
    W: ‹L ∈# W› and n-d: ‹no-dup (Propagated L D # M)› and
    lazy: ‹twl-lazy-update M (TWL-Clause W UW)›
  shows
    ‹twl-lazy-update (Propagated L D # M) (TWL-Clause W UW)›
  unfolding twl-lazy-update.simps
proof (intro conjI impI allI)
```

fix *La*
assume
La: ‹*La* ∈# *W*› **and**
uL-M: ‹− *La* ∈ *lits-of-l* (*Propagated L D* # *M*)› **and**
b: ‹¬ *has-blit* (*Propagated L D* # *M*) (*W* + *UW*) *La*›
have *b′*: ‹¬*has-blit M* (*W* + *UW*) *La*›
apply (*rule no-has-blit-propagate*[*OF b*])
using *assms* **by** *auto*

have ‹− *La* ∈ *lits-of-l M* ⟶ (∀ *K*∈#*UW*. *get-level M K* ≤ *get-level M La* ∧ − *K* ∈ *lits-of-l M*)›
using *lazy assms b′ uL-M La* **unfolding** *twl-lazy-update.simps*
by *blast*
then consider
‹∀ *K*∈#*UW*. *get-level M K* ≤ *get-level M La* ∧ −*K* ∈ *lits-of-l M*› **and** ‹*La* ≠ −*L*› |
‹*La* = −*L*›
using *b′ uL-M La*
by (*simp only*: *list.set*(*2*) *lits-of-insert insert-iff uminus-lit-swap*)
fastforce
then show ‹∀ *K*∈#*UW*. *get-level* (*Propagated L D* # *M*) *K* ≤ *get-level* (*Propagated L D* # *M*) *La* ∧
−*K* ∈ *lits-of-l* (*Propagated L D* # *M*)›
proof *cases*
case *1*
have [*simp*]: ‹*has-blit* (*Propagated L D* # *M*) (*W* + *UW*) *L*› **if** ‹*L* ∈# *W*+*UW*›
using *that* **unfolding** *has-blit-def* **apply** −
by (*rule exI*[*of* - *L*]) (*auto simp*: *get-level-cons-if atm-of-eq-atm-of*)
show *?thesis*
using *n-d b 1 b′ uL-M*
by (*auto simp*: *get-level-cons-if atm-of-eq-atm-of*
count-decided-ge-get-level Decided-Propagated-in-iff-in-lits-of-l
dest!: *multi-member-split*)
next
case *2*
have [*simp*]: ‹*has-blit* (*Propagated L D* # *M*) (*W* + *UW*) (−*L*)›
using *2 La W* **unfolding** *has-blit-def* **apply** −
by (*rule exI*[*of* - *L*])
(*auto simp*: *get-level-cons-if atm-of-eq-atm-of*)
show *?thesis*
using *2 b count-decided-ge-get-level*[*of* ‹*Propagated L D* # *M*›]
by (*auto simp*: *uminus-lit-swap split*: *if-splits*)
qed
qed

lemma *pair-in-image-Pair*:
‹(*La*, *C*) ∈ *Pair L* ‘ *D* ⟷ *La* = *L* ∧ *C* ∈ *D*›
by *auto*

lemma *image-Pair-subset-mset*:
‹*Pair L* ‘# *A* ⊆# *Pair L* ‘# *B* ⟷ *A* ⊆# *B*›
proof −
have [*simp*]: ‹*remove1-mset* (*L*, *x*) (*Pair L* ‘# *B*) = *Pair L* ‘# (*remove1-mset x B*)› **for** *x* :: ′*b* **and** *B*
proof −
have ‹(*L*, *x*) ∈# *Pair L* ‘# *B* ⟶ *x* ∈# *B*›
by *force*
then show *?thesis*
by (*metis* (*no-types*) *diff-single-trivial image-mset-remove1-mset-if*)
qed

show *?thesis*
by (*induction A arbitrary*: *B*) (*auto simp*: *insert-subset-eq-iff*)
qed

lemma *count-image-mset-Pair2*:
‹*count* {#(*L*, *x*). *L* ∈# *M x*#} (*L*, *C*) = (*if x* = *C then count* (*M x*) *L else* 0)›
proof −
have ‹*count* (*M C*) *L* = *count* {#*L*. *L*∈#*M C*#} *L*›
by *simp*
also have ‹... = *count* ((λ*L*. *Pair L C*) `# {#*L*. *L*∈#*M C*#}) ((λ*L*. *Pair L C*) *L*)›
by (*subst* (2) *count-image-mset-inj*) (*simp-all add*: *inj-on-def*)
finally have *C*: ‹*count* {#(*L*, *C*). *L* ∈# {#*L*. *L* ∈# *M C*#}#} (*L*, *C*) = *count* (*M C*) *L*› **..**

show *?thesis*
apply (*cases* ‹*x* ≠ *C*›)
apply (*auto simp*: *not-in-iff*[*symmetric*] *count-image-mset*; *fail*)[]
using *C* **by** *simp*
qed

lemma *lit-of-inj-on-no-dup*: ‹*no-dup M* ⟹ *inj-on* (λ*x*. − *lit-of x*) (*set M*)›
by (*induction M*) (*auto simp*: *no-dup-def*)

lemma
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of S*)› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
dist-q: ‹*distinct-queued S*› **and**
ws: ‹*clauses-to-update-inv S*›
shows *twl-cp-twl-st-exception-inv*: ‹*twl-st-exception-inv T*› **and**
twl-cp-clauses-to-update: ‹*clauses-to-update-inv T*›
using *cdcl twl twl-excep valid inv no-dup ws*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE N0 U0 L Q*)
case *1* **note** - = *this*(2)
then show *?case* **unfolding** *twl-st-inv.simps twl-is-an-exception-def*
by (*fastforce simp add*: *pair-in-image-Pair image-constant-conv uminus-lit-swap*
twl-exception-inv.simps)
case *2* **note** *twl* = *this*(1) **and** *ws* = *this*(6)
have *struct*: ‹*struct-wf-twl-cls C*› **if** ‹*C* ∈# *N* + *U*› **for** *C*
using *twl that* **by** (*simp add*: *twl-st-inv.simps*)
have *H*: ‹*count* (*watched C*) *L* ≤ *1*› **if** ‹*C* ∈# *N* + *U*› **for** *C L*
using *struct*[*OF that*] **by** (*cases C*) (*auto simp add*: *twl-st-inv.simps size-2-iff*)
have *sum-le-count*: ‹(∑ *x*∈#*N*+*U*. *count* {#(*L*, *x*). *L* ∈# *watched x*#} (*a*, *b*)) ≤ *count* (*N*+*U*) *b*›
for *a b*
apply (*subst* (2) *count-sum-mset-if-1-0*)
apply (*rule sum-mset-mono*)
using *H* **apply** (*auto simp*: *count-image-mset-Pair2*)
done
define *NU* **where** *NU*[*symmetric*]: ‹*NU* = *N* + *U*›
show *?case*
using *ws* **by** (*fastforce simp add*: *pair-in-image-Pair multiset-filter-mono2 image-Pair-subset-mset*
clauses-to-update-prop.simps NU filter-mset-empty-conv)

next
case (*propagate D L L′ M N U NE UE NS US N0 U0 WS Q*) **note** *watched* = *this*(*1*) **and** *undef* = *this*(*2*) **and**
unw = *this*(*3*)

case *1*
note *twl* = *this*(*1*) **and** *twl-excep* = *this*(*2*) **and** *valid* = *this*(*3*) **and** *inv* = *this*(*4*) **and**
no-dup = *this*(*5*) **and** *ws* = *this*(*6*)
have [*simp*]: ‹− *L′* ∉ *lits-of-l M*›
using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *blast*
have *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
then have *wf-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*simp add*: *twl-st-inv.simps*)
have ‹∀ *s*∈#*clause* ‘# *U*. ¬ *tautology s*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* **by** (*simp-all add*: $cdcl_W$*-restart-mset-state*
$state_W$*-of-def*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps* $state_W$*-of-def*)
have [*simp*]: ‹*L* ≠ *L′*›
using *wf-D watched* **by** (*cases D*) *auto*
have [*simp*]: ‹− *L* ∈ *lits-of-l M*›
using *valid* **by** *auto*
then have [*simp*]: ‹*L* ∉ *lits-of-l M*›
using *n-d no-dup-consistentD* **by** *blast*
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
have [*simp*]: ‹*has-blit* (*Propagated L′* (*add-mset L* (*add-mset L′ x2*)) # *M*)
(*add-mset L* (*add-mset L′ x2*)) *L*› **for** *x2*
unfolding *has-blit-def*
by (*rule exI*[*of* - *L′*])
(*use lev-L* **in** ‹*auto simp*: *get-level-cons-if*›)
have *HH*: ‹¬*clauses-to-update-prop* (*add-mset* (−*L′*) *Q*) (*Propagated L′* (*clause D*) # *M*) (*L*, *D*)›
using *watched* **unfolding** *clauses-to-update-prop.simps* **by** (*cases D*) (*auto simp*: *watched*)
have ‹*add-mset L Q* ⊆# {#− *lit-of x*. *x* ∈# *mset M*#}›
using *no-dup* **by** (*auto*)
moreover have ‹*distinct-mset* {#− *lit-of x*. *x* ∈# *mset M*#}›
by (*subst distinct-image-mset-inj*)
(*use n-d* **in** ‹*auto simp*: *lit-of-inj-on-no-dup distinct-map no-dup-def*›)
ultimately have [*simp*]: ‹*L* ∉# *Q*›
by (*metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add*)
have ‹¬*has-blit M* (*clause D*) *L*›
using *watched undef unw n-d* **by** (*cases D*)
(*auto simp*: *has-blit-def Decided-Propagated-in-iff-in-lits-of-l dest*: *no-dup-consistentD*)
then have *w-q-p-D*: ‹*clauses-to-update-prop Q M* (*L*, *D*)›
by (*auto simp*: *clauses-to-update-prop.simps watched*)
have ‹*Pair L* ‘# {#*C* ∈# *add-mset D NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*, *D*) *WS*›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
then have *IH*: ‹*Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
using *w-q-p-D* **by** *auto*
have *IH-Q*: ‹∀ *La C*. *C* ∈# *add-mset D NU* ⟶ *La* ∈# *watched C* ⟶ − *La* ∈ *lits-of-l M* ⟶
¬ *has-blit M* (*clause C*) *La* ⟶ (*La*, *C*) ∉# *add-mset* (*L*, *D*) *WS* ⟶ *La* ∈# *Q*›

```
  using ws no-dup unfolding clauses-to-update-inv.simps NU
  by (auto simp: all-conj-distrib)

show ?case
  unfolding Ball-def twl-st-exception-inv.simps twl-exception-inv.simps
proof (intro allI conjI impI)
  fix C J K
  assume C: ‹C ∈# N + U› and
    watched-C: ‹J ∈# watched C› and
    J: ‹– J ∈ lits-of-l (Propagated L' (clause D) # M)› and
    J': ‹¬ has-blit (Propagated L' (clause D) # M) (clause C) J› and
    J-notin: ‹J ∉# add-mset (– L') Q› and
    C-WS: ‹(J, C) ∉# WS› and
    ‹K ∈# unwatched C›
  moreover have ‹¬ has-blit M (clause C) J›
    using no-has-blit-propagate'[OF J'] n-d undef by fast
  ultimately have ‹– K ∈ lits-of-l (Propagated L' (clause D) # M)› if ‹C ≠ D›
    using twl-excep that by (auto simp add: uminus-lit-swap twl-exception-inv.simps)

  moreover have CD: False if ‹C = D›
    using J J' watched-C watched that J-notin
    by (cases D)  (auto simp: add-mset-eq-add-mset)
  ultimately show ‹– K ∈ lits-of-l (Propagated L' (clause D) # M)›
    by blast
qed
case 2
show ?case
proof (induction rule: clauses-to-update-inv-cases)
  case (WS-nempty L'' C)
  then have [simp]: ‹L'' = L›
    using ws no-dup unfolding clauses-to-update-inv.simps NU by (auto simp: all-conj-distrib)

  have *: ‹Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#} ⊇#
    Pair L `# {#C ∈# NU.
      clauses-to-update-prop (add-mset (– L') Q) (Propagated L' (clause D) # M) (L'', C)#}›
    using undef n-d
    unfolding image-Pair-subset-mset multiset-filter-mono2 clauses-to-update-prop.simps
    by (auto dest!: no-has-blit-propagate')
  show ?case
    using subset-mset.dual-order.trans[OF IH *]  HH
    unfolding NU ‹L'' = L›
    by simp
next
  case (WS-empty K)
  then show ?case
    using IH IH-Q watched undef n-d unfolding NU
    by (cases D) (auto simp: filter-mset-empty-conv
        clauses-to-update-prop.simps watched add-mset-eq-add-mset
        dest!: no-has-blit-propagate')
next
  case (Q LC' C)
  then show ?case
     using watched 1.prems(6) HH Q.hyps HH IH-Q undef n-d
     apply (cases D)
     apply (cases C)
     apply (auto simp: add-mset-eq-add-mset NU)
```

```
    by (metis HH Q.IH(2) Q.IH(3) Q.hyps clauses-to-update-prop.simps insert-iff
      no-has-blit-propagate' set-mset-add-mset-insert)
  qed
next
  case (conflict D L L' M N U NE UE N0 U0 WS Q)
  case 1
  note twl = this(5)
  show ?case by (auto simp: twl-st-inv.simps twl-exception-inv.simps)

  case 2
  show ?case
    by (auto simp: twl-st-inv.simps twl-exception-inv.simps)
next
  case (delete-from-working L' D M N U NE UE NS US N0 U0 L WS Q) note watched = this(1) and
L' = this(2)

  case 1 note twl = this(1) and twl-excep = this(2) and valid = this(3) and inv = this(4) and
    no-dup = this(5) and ws = this(6)
  have n-d: ‹no-dup M›
    using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps state_W-of-def)
  have D-N-U: ‹D ∈# N + U›
    using valid by auto
  then have wf-D: ‹struct-wf-twl-cls D›
    using twl by (simp add: twl-st-inv.simps)
  obtain NU where NU: ‹N + U = add-mset D NU›
    by (metis D-N-U insert-DiffM)
  have D-N-U: ‹D ∈# N + U› and lev-L: ‹get-level M L = count-decided M›
    using valid by auto
  have [simp]: ‹has-blit M (clause D) L›
    unfolding has-blit-def
    by (rule exI[of - L'])
      (use watched L' lev-L in ‹auto simp: count-decided-ge-get-level›)
  have [simp]: ‹¬clauses-to-update-prop Q M (L, D)›
    using L' by (auto simp: clauses-to-update-prop.simps watched)
  have IH-WS: ‹Pair L `# {#C ∈# N + U. clauses-to-update-prop Q M (L, C)#} ⊆# add-mset (L,
D) WS›
    using ws by (auto simp del: filter-union-mset simp: NU)
  then have IH-WS-NU: ‹Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#} ⊆#
    add-mset (L, D) WS›
    using ws by (auto simp del: filter-union-mset simp: NU)

  have IH-WS': ‹Pair L `# {#C ∈# N + U. clauses-to-update-prop Q M (L, C)#} ⊆# WS›
    by (rule subset-add-mset-notin-subset-mset[OF IH-WS]) auto
  have IH-Q: ‹∀ La C. C ∈# add-mset D NU ⟶ La ∈# watched C ⟶ - La ∈ lits-of-l M ⟶
    ¬has-blit M (clause C) La ⟶ (La, C) ∉# add-mset (L, D) WS ⟶ La ∈# Q›
    using ws no-dup unfolding clauses-to-update-inv.simps NU
    by (auto simp: all-conj-distrib)

  show ?case
    unfolding Ball-def twl-st-exception-inv.simps twl-exception-inv.simps
  proof (intro allI conjI impI)
    fix C J K
    assume C: ‹C ∈# N + U› and
      watched-C: ‹J ∈# watched C› and
      J: ‹- J ∈ lits-of-l M› and
```

J': ‹¬*has-blit M* (*clause C*) *J*› **and**
J-notin: ‹*J* ∉# *Q*› **and**
C-WS: ‹(*J*, *C*) ∉# *WS*› **and**
‹*K* ∈# *unwatched C*›
then have ‹− *K* ∈ *lits-of-l M*› **if** ‹*C* ≠ *D*›
using *twl-excep that* **by** (*simp add*: *uminus-lit-swap twl-exception-inv.simps*)

moreover {
from *n-d* **have** *False* **if** ‹ − *L'* ∈ *lits-of-l M*› ‹*L'* ∈ *lits-of-l M*›
using *that consistent-interp-def distinct-consistent-interp* **by** *blast*
then have *CD*: *False* **if** ‹*C* = *D*›
using *J J' watched-C watched L' C-WS IH-Q J-notin* ‹¬ *clauses-to-update-prop Q M* (*L*, *D*)› *that*
apply (*auto simp*: *add-mset-eq-add-mset*)
by (*metis C-WS J-notin* ‹¬ *clauses-to-update-prop Q M* (*L*, *D*)› *clauses-to-update-prop.simps that*)
}
ultimately show ‹− *K* ∈ *lits-of-l M*›
by *blast*
qed

case *2*
show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty K C*) **note** *KC* = *this*
have *LK*: ‹*L* = *K*›
using *no-dup KC* **by** *auto*
from *subset-add-mset-notin-subset-mset*[*OF IH-WS*]
have *1*: ‹*Pair K* '# {#*C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
using *L' LK* ‹*has-blit M* (*clause D*) *L*›
by (*auto simp del*: *filter-union-mset simp*: *pair-in-image-Pair watched add-mset-eq-add-mset all-conj-distrib clauses-to-update-prop.simps*)
show *?case*
by (*metis* (*no-types*, *lifting*) *1 LK*)
next
case (*WS-empty K*) **note** [*simp*] = *this*(*1*)
have [*simp*]: ‹¬*clauses-to-update-prop Q M* (*K*, *D*)›
using *IH-Q WS-empty.IH watched* ‹*has-blit M* (*clause D*) *L*›
using *IH-WS' IH-Q watched* **by** (*auto simp*: *add-mset-eq-add-mset NU filter-mset-empty-conv all-conj-distrib clauses-to-update-prop.simps*)
show *?case*
using *IH-WS' IH-Q watched* **by** (*auto simp*: *add-mset-eq-add-mset NU filter-mset-empty-conv all-conj-distrib clauses-to-update-prop.simps*)
next
case (*Q K C*)
then show *?case*
using ‹¬ *clauses-to-update-prop Q M* (*L*, *D*)› *ws*
unfolding *clauses-to-update-inv.simps*(*1*) *clauses-to-update-prop.simps member-add-mset is-blit-def*
by *blast*
qed
next
case (*update-clause D L L' M K N U N' U' NE UE NS US N0 U0 WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*)
and *L'* = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N'U'* = *this*(*6*)

case *1* **note** *twl* = *this*(*1*) **and** *twl-excep* = *this*(*2*) **and** *valid* = *this*(*3*) **and** *inv* = *this*(*4*) **and** *no-dup* = *this*(*5*) **and** *ws* = *this*(*6*)
obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
then have *struct-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *L'-UWD*: ‹*L* ∉# *remove1-mset L' UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD L* ≥ *1*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
then have ‹*count* (*clause D*) *L* ≥ *2*›
using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have *L'-L'-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD K* ≥ *2*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
then have ‹*count* (*clause D*) *K* ≥ *2*›
using *D L'* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N'* + *U'*› **for** *C*
using *C N'U' that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *trail.simps* $state_W$*-of-def*)
then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def distinct-consistent-interp* **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
have *L-M*: ‹*L* ∉ *lits-of-l M*›
using *n-d uL* **by** (*fastforce dest*!: *distinct-consistent-interp simp*: *consistent-interp-def lits-of-def uminus-lit-swap*)
have *w-max-D*: ‹*watched-literals-false-of-max-level M D*›

using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *lev-L'*: ‹*get-level M L' = count-decided M*›
if ‹− *L' ∈ lits-of-l M*› ‹¬*has-blit M* (*clause D*) *L'*›
using *L-M w-max-D D watched L' uL that* **by** *auto*
have *D-ne-D*: ‹*D ≠ update-clause D L K*›
using *D add-remove-WD* **by** *auto*
have *N'U'*: ‹*N' + U' = add-mset ?D* (*remove1-mset D* (*N + U*))›
using *N'U' D-N-U* **by** (*auto elim*!: *update-clausesE*)
define *NU* **where** ‹*NU = remove1-mset D* (*N + U*)›
then have *NU*: ‹*N + U = add-mset D NU*›
using *D-N-U* **by** *auto*
have *watched-D*: ‹*watched ?D* = {#*K*, *L'*#}›
using *D add-remove-WD watched* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps* $state_W$*-of-def*)
have *D-N-U*: ‹*D ∈# N + U*› **and** *lev-L*: ‹*get-level M L = count-decided M*›
using *valid* **by** *auto*
have ‹*has-blit* (*Propagated L' C # M*)
(*add-mset L* (*add-mset L' x2*)) *L*› **for** *C x2*
unfolding *has-blit-def*
by (*rule exI*[*of* - *L'*])
(*use lev-L* **in** ‹*auto simp*: *count-decided-ge-get-level get-level-cons-if*›)
then have *HH*: ‹¬*clauses-to-update-prop* (*add-mset* (−*L'*) *Q*) (*Propagated L'* (*clause D*) *# M*) (*L*, *D*)›
using *watched* **unfolding** *clauses-to-update-prop.simps* **by** (*cases D*) (*auto simp*: *watched*)
have ‹*add-mset L Q ⊆# {#− lit-of x. x ∈# mset M#}*›
using *no-dup* **by** (*auto*)
moreover have ‹*distinct-mset {#− lit-of x. x ∈# mset M#}*›
by (*subst distinct-image-mset-inj*)
(*use n-d* **in** ‹*auto simp*: *lit-of-inj-on-no-dup distinct-map no-dup-def*›)
ultimately have *LQ*: ‹*L ∉# Q*›
by (*metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add*)
have *w-q-p-D*: ‹¬*has-blit M* (*clause D*) *L ⟹ clauses-to-update-prop Q M* (*L*, *D*)›
using *watched uL L'* **by** (*cases D*) (*auto simp*: *LQ clauses-to-update-prop.simps*)
have ‹*Pair L `# {#C ∈# add-mset D NU. clauses-to-update-prop Q M* (*L*, *C*)#} *⊆# add-mset* (*L*, *D*) *WS*›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
then have *IH*: ‹¬*has-blit M* (*clause D*) *L ⟹ Pair L `# {#C ∈# NU. clauses-to-update-prop Q M* (*L*, *C*)#} *⊆# WS*›
using *w-q-p-D* **by** *auto*
have *IH-Q*: ‹⋀*La C. C ∈# add-mset D NU ⟹ La ∈# watched C ⟹ − La ∈ lits-of-l M ⟹*
¬*has-blit M* (*clause C*) *La ⟹* (*La*, *C*) *∉# add-mset* (*L*, *D*) *WS ⟹ La ∈# Q*›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
have *blit-clss-to-upd*: ‹*has-blit M* (*clause D*) *L ⟹* ¬ *clauses-to-update-prop Q M* (*L*, *D*)›
by (*auto simp*: *clauses-to-update-prop.simps*)
have
‹*Pair L `# {#C ∈# N + U. clauses-to-update-prop Q M* (*L*, *C*)#} *⊆# add-mset* (*L*, *D*) *WS*›
using *ws* **by** (*auto simp del*: *filter-union-mset*)
moreover have ‹*has-blit M* (*clause D*) *L ⟹*
(*L*, *D*) *∉# Pair L `# {#C ∈# NU. clauses-to-update-prop Q M* (*L*, *C*)#}›
by (*auto simp*: *clauses-to-update-prop.simps*)
ultimately have *Q-M-L-WS*:
‹*Pair L `# {#C ∈# NU. clauses-to-update-prop Q M* (*L*, *C*)#} *⊆# WS*›

```
    by (auto simp del: filter-union-mset simp: NU w-q-p-D blit-clss-to-upd
      intro: subset-add-mset-notin-subset-mset split: if-splits)
  have L-ne-L': ‹L ≠ L'›
    using struct-D D watched by auto
  have clss-upd-D[simp]: ‹clause ?D = clause D›
    using D K watched by auto
  show ?case
    unfolding Ball-def twl-st-exception-inv.simps twl-exception-inv.simps
  proof (intro allI conjI impI)
    fix C J K''
    assume C: ‹C ∈# N' + U'› and
      watched-C: ‹J∈# watched C› and
      J: ‹- J ∈ lits-of-l M› and
      J': ‹¬has-blit M (clause C) J› and
      J-notin: ‹J ∉# Q› and
      C-WS: ‹(J, C) ∉# WS› and
      K'': ‹K'' ∈# unwatched C›
    then have ‹- K'' ∈ lits-of-l M› if ‹C ≠ D› ‹C ≠ ?D›
      using twl-excep that *[OF - C] N'U' by (simp add: uminus-lit-swap twl-exception-inv.simps)
    moreover have ‹- K'' ∈ lits-of-l M› if CD: ‹C = D›
    proof (rule ccontr)
      assume uK''-M: ‹- K'' ∉ lits-of-l M›
      have ‹Pair L `# {#C ∈# N + U. clauses-to-update-prop Q M (L, C)#} ⊆# add-mset (L, D)
WS›
        using ws by (auto simp: all-conj-distrib
          simp del: filter-union-mset)
      show False
      proof cases
        assume [simp]: ‹J = L›
        have w-q-p-L: ‹clauses-to-update-prop Q M (L, C)›
          unfolding clauses-to-update-prop.simps watched-C J J' K'' uK''-M
          apply (auto simp add: add-mset-eq-add-mset conj-disj-distribR ex-disj-distrib)
          using watched watched-C CD J J' J-notin K'' uK''-M uL L' L-M
          by (auto simp: clauses-to-update-prop.simps add-mset-eq-add-mset)
        then have ‹Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#} ⊆# WS›
          using ws by (auto simp: all-conj-distrib NU CD simp del: filter-union-mset)
        moreover have ‹(L, C) ∈# Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#}›
          using C w-q-p-L D-ne-D by (auto simp: pair-in-image-Pair N'U' NU CD)
        ultimately have ‹(L, C) ∈# WS›
          by blast
        then show ‹False›
          using C-WS by simp
      next
        assume ‹J ≠ L›
        then have ‹clauses-to-update-prop Q M (L, C)›
          unfolding clauses-to-update-prop.simps watched-C J J' K'' uK''-M
          apply (auto simp add: add-mset-eq-add-mset conj-disj-distribR ex-disj-distrib)
          using watched watched-C CD J J' J-notin K'' uK''-M uL L' L-M
            apply (auto simp: clauses-to-update-prop.simps add-mset-eq-add-mset)
          using C-WS D-N-U clauses-to-update-prop.simps ws by auto
        then show ‹False›
          using C-WS D-N-U J J' J-notin ‹J ≠ L› that watched-C ws by auto
      qed
    qed
    moreover {
      assume CD: ‹C = ?D›
```

```
    have JL[simp]: ‹J = L'›
      using CD J J' watched-C watched L' D uK-M undef
      by (auto simp: add-mset-eq-add-mset)
    have ‹K'' ≠ K›
      using K'' uK-M uL D L'-L'-UWD unfolding CD
      by (cases D) auto
    have K''-unwatched-L: ‹K'' ∈# remove1-mset K (unwatched D) ∨ K'' = L›
      using K'' unfolding CD by (cases D) auto
    have ‹clause C = clause D›
      using D K watched unfolding CD by auto
    then have blit: ‹¬ has-blit M (clause D) L'›
      using J' unfolding CD by simp
    have False if ‹- L' ∈ lits-of-l M› ‹L' ∈ lits-of-l M›
      using n-d that consistent-interp-def distinct-consistent-interp by blast
    have H: ‹⋀x La xa. x ∈# N + U ⟹
        La ∈# watched x ⟹ - La ∈ lits-of-l M ⟹
        ¬has-blit M (clause x) La ⟹ La ∉# Q ⟹ (La, x) ∉# add-mset (L, D) WS ⟹
        xa ∈# unwatched x ⟹ - xa ∈ lits-of-l M›
      using twl-excep[unfolded twl-st-exception-inv.simps Ball-def twl-exception-inv.simps]
      unfolding has-blit-def is-blit-def
      by blast
    have LL': ‹L ≠ L'›
      using struct-D watched by (cases D) auto
    have L'D-WS: ‹(L', D) ∉# WS›
      using no-dup LL' by (auto dest: multi-member-split)
    have ‹xa ∈# unwatched D ⟹ - xa ∈ lits-of-l M›
      if ‹- L' ∈ lits-of-l M› and ‹L' ∉# Q› and ‹¬ has-blit M (clause D) L'› for xa
      by (rule H[of D L'])
        (use D-N-U watched LL' that L'D-WS K'' that in ‹auto simp: add-mset-eq-add-mset L-M›)
    consider
      (unwatched-unqueued) ‹K'' ∈# remove1-mset K (unwatched D)› |
      (KL) ‹K'' = L›
      using K''-unwatched-L by blast
    then have ‹- K'' ∈ lits-of-l M›
    proof cases
      case KL
      then show ?thesis
        using uL by simp
    next
      case unwatched-unqueued
      moreover have ‹L' ∉# Q›
        using JL J-notin by blast
      ultimately show ?thesis
        using blit H[of D L'] D-N-U watched LL' L'D-WS K'' J J'
        by (auto simp: add-mset-eq-add-mset L-M dest: in-diffD)
    qed
    }
  ultimately show ‹- K'' ∈ lits-of-l M›
    by blast
qed

case 2
show ?case
proof (induction rule: clauses-to-update-inv-cases)
  case (WS-nempty K'' C) note KC = this(1)
  have LK: ‹L = K''›
```

```
    using no-dup KC by auto
  have [simp]: ‹¬clauses-to-update-prop Q M (K'', update-clause D K'' K)›
    using watched uK-M struct-D
    by (cases D) (auto simp: clauses-to-update-prop.simps add-mset-eq-add-mset LK)
  have 1: ‹Pair L `# {#C ∈# N' + U'. clauses-to-update-prop Q M (L, C)#} ⊆#
    Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#}›
    unfolding image-Pair-subset-mset LK
    using LK N'U' by (auto simp del: filter-union-mset simp: pair-in-image-Pair watched NU
      add-mset-eq-add-mset all-conj-distrib)
  then show ‹Pair K'' `# {#C ∈# N' + U'. clauses-to-update-prop Q M (K'', C)#} ⊆# WS›
    using Q-M-L-WS unfolding LK by auto
next
  case (WS-empty K'')
  then show ?case
    using IH IH-Q uL uK-M L-M watched L-ne-L' unfolding N'U' NU
    by (force simp: filter-mset-empty-conv clauses-to-update-prop.simps
      add-mset-eq-add-mset watched-D all-conj-distrib)
next
  case (Q K' C) note C = this(1) and uK'-M = this(2) and uK''-M = this(3) and KC-WS =
this(4)
    and watched-C = this(5)
  have ?case if CD: ‹C ≠ D› ‹C ≠ ?D›
    using IH-Q[of C K'] CD watched uK-M L' L-ne-L' L-M uK'-M uK''-M
      Q unfolding N'U' NU
    by auto
  moreover have ?case if CD: ‹C = D›
  proof −
    consider
      (KL) ‹K' = L› |
      (K'L') ‹K' = L'›
      using watched watched-C CD by (auto simp: add-mset-eq-add-mset)
    then show ?thesis
    proof cases
      case KL note [simp] = this
      have ‹(L, C) ∈# Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#}›
        using CD C w-q-p-D uK''-M unfolding NU N'U' by (auto simp: pair-in-image-Pair D-ne-D)
      then have ‹(L, C) ∈# WS›
        using Q-M-L-WS by blast
      then have False using KC-WS unfolding CD by simp
      then show ?thesis by fast
    next
      case K'L' note [simp] = this
      show ?thesis
        by (rule IH-Q[of C]) (use CD watched-C uK'-M uK''-M KC-WS L-ne-L' in auto)
    qed
  qed
  moreover {
    have ‹(L', D) ∉# WS›
      using no-dup L-ne-L' by (auto simp: all-conj-distrib)
    then have ?case if CD: ‹C = ?D›
      using IH-Q[of D L] IH-Q[of D L'] CD watched watched-D watched-C watched uK-M L'
        L-ne-L' L-M uK'-M uK''-M D-ne-D C unfolding NU N'U'
      by (auto simp: add-mset-eq-add-mset all-conj-distrib imp-conjR)
  }
  ultimately show ?case
    by blast
```

```
  qed
qed

declare state_W-of-def[simp]

lemma twl-cp-twl-inv:
  assumes
    cdcl: ‹cdcl-twl-cp S T› and
    twl: ‹twl-st-inv S› and
    valid: ‹valid-enqueued S› and
    inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)› and
    twl-excep: ‹twl-st-exception-inv S› and
    no-dup: ‹no-duplicate-queued S› and
    wq: ‹clauses-to-update-inv S›
  shows ‹twl-st-inv T›
  using cdcl twl valid inv twl-excep no-dup wq
proof (induction rule: cdcl-twl-cp.induct)
  case (pop M N U NE UE NS US L Q) note inv = this(1)
  then show ?case unfolding twl-st-inv.simps twl-is-an-exception-def
    by (fastforce simp add: pair-in-image-Pair)
next
  case (propagate D L L' M N U NE UE NS US N0 U0 WS Q) note watched = this(1) and undef =
this(2) and
   unw = this(3) and twl = this(4) and valid = this(5) and inv = this(6) and exception = this(7)
  have uL'-M[simp]: ‹- L' ∉ lits-of-l M›
    using Decided-Propagated-in-iff-in-lits-of-l propagate.hyps(2) by blast
  have D-N-U: ‹D ∈# N + U› and lev-L: ‹get-level M L = count-decided M›
    using valid by auto
  then have wf-D: ‹struct-wf-twl-cls D›
    using twl by (auto simp add: twl-st-inv.simps)
  have [simp]: ‹- L ∈ lits-of-l M›
    using valid by auto
  have n-d: ‹no-dup M›
    using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps)
  show ?case unfolding twl-st-simps Ball-def
  proof (intro allI conjI impI)
    fix C
    assume C: ‹C ∈# N + U›
    show ‹struct-wf-twl-cls C›
      using twl C by (auto simp: twl-st-inv.simps)[]
    have watched-max: ‹watched-literals-false-of-max-level M C›
      using twl C by (auto simp: twl-st-inv.simps)
    then show ‹watched-literals-false-of-max-level (Propagated L' (clause D) # M) C›
      using undef n-d
      by (cases C) (auto simp: get-level-cons-if dest!: no-has-blit-propagate')

    assume excep: ‹¬twl-is-an-exception C (add-mset (- L') Q) WS›
    have excep-C: ‹¬ twl-is-an-exception C Q (add-mset (L, D) WS)› if ‹C ≠ D›
      using excep that by (auto simp add: twl-is-an-exception-def)
    then have ‹twl-lazy-update M C› if ‹C ≠ D›
      using twl C D-N-U that by (cases ‹C = D›) (auto simp add: twl-st-inv.simps)
    then show ‹twl-lazy-update (Propagated L' (clause D) # M) C›
      using twl C excep uL'-M twl undef n-d uL'-M unw watched-max
      apply (cases C)
      apply (auto simp: get-level-cons-if count-decided-ge-get-level
```

```
      twl-is-an-exception-add-mset-to-queue atm-of-eq-atm-of
      dest!: no-has-blit-propagate' no-has-blit-propagate)
    apply (metis twl-clause.sel(2) uL'-M unw)
    apply (metis twl-clause.sel(2) uL'-M unw)
    apply (metis twl-clause.sel(2) uL'-M unw)
    apply (metis twl-clause.sel(2) uL'-M unw)
    done
 qed
next
 case (conflict D L L' M N U NE UE NS US N0 U0 WS Q) note twl = this(4)
 then show ?case
   by (auto simp: twl-st-inv.simps)
next
 case (delete-from-working L' D M N U NE UE NS US N0 U0 L WS Q) note watched = this(1) and
L' = this(2) and
 twl = this(3) and valid = this(4) and inv = this(5) and tauto = this(6)
 show ?case unfolding twl-st-simps Ball-def
 proof (intro allI conjI impI)
   fix C
   assume C: ‹C ∈# N + U›
   show ‹struct-wf-twl-cls C›
     using twl C by (auto simp: twl-st-inv.simps)[]
   show ‹watched-literals-false-of-max-level M C›
     using twl C by (auto simp: twl-st-inv.simps)

   assume excep: ‹¬twl-is-an-exception C Q WS›
   have ‹get-level M L = count-decided M› and L: ‹−L ∈ lits-of-l M› and D: ‹D ∈# N + U›
     using valid by auto
   have ‹watched-literals-false-of-max-level M D›
     using twl D by (auto simp: twl-st-inv.simps)
   have ‹no-dup M›
     using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
       cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (simp add: trail.simps)
   then have [simp]: ‹− L' ∉ lits-of-l M›
     using L' consistent-interp-def distinct-consistent-interp by blast
   have ‹¬ twl-is-an-exception C Q (add-mset (L, D) WS)› if ‹C ≠ D›
     using excep that by (auto simp add: twl-is-an-exception-def)
   have twl-D: ‹twl-lazy-update M D›
     using twl C excep twl watched L' ‹watched-literals-false-of-max-level M D›
     by (cases D)
       (auto simp: get-level-cons-if count-decided-ge-get-level has-blit-def
         twl-is-an-exception-add-mset-to-queue atm-of-eq-atm-of count-decided-ge-get-level
         dest!: no-has-blit-propagate' no-has-blit-propagate)
   have twl-C: ‹twl-lazy-update M C› if ‹C ≠ D›
     using twl C excep that by (auto simp add: twl-st-inv.simps
         twl-is-an-exception-add-mset-to-clauses-to-update)

   show ‹twl-lazy-update M C›
     using twl-C twl-D by blast
 qed
next
 case (update-clause D L L' M K N U N' U' NE UE NS US N0 U0 WS Q) note watched = this(1)
and uL = this(2)
   and L' = this(3) and K = this(4) and undef = this(5) and N'U' = this(6) and twl = this(7)
and
   valid = this(8) and inv = this(9) and twl-excep = this(10) and
```

no-dup = *this*(*11*) **and** *wq* = *this*(*12*)
obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
then have *struct-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *L′-UWD*: ‹*L* ∉# *remove1-mset L′ UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD L* ≥ *1*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *L* ≥ *2*›
using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have *L′-L′-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD K* ≥ *2*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *K* ≥ *2*›
using *D L′* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N′* + *U′*› **for** *C*
using *C N′U′ that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
distinct-consistent-interp **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
have *cls-D-D*: ‹*clause ?D* = *clause D*›
by (*cases D*) (*use watched K* **in** *auto*)

have *L-M*: ‹*L* ∉ *lits-of-l M*›
using *n-d uL* **by** (*fastforce dest*!: *distinct-consistent-interp*
simp: *consistent-interp-def lits-of-def uminus-lit-swap*)
have *w-max-D*: ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)

show *?case* **unfolding** *twl-st-simps Ball-def*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N'* + *U'*›
moreover have ‹*L* ≠ *L'*›
using *struct-D watched* **by** (*auto simp*: *D dest*: *multi-member-split*)
ultimately have *struct-D'*: ‹*struct-wf-twl-cls ?D*›
using *L K struct-D watched* **by** (*auto simp*: *D L'-UWD L'-L'-UWD dest*: *in-diffD*)

have *struct-C*: ‹*struct-wf-twl-cls C*› **if** ‹*C* ≠ *?D*›
using *twl C that N'U'* **by** (*fastforce simp*: *twl-st-inv.simps elim*!: *update-clausesE*
split: *if-splits dest*: *in-diffD*)
show ‹*struct-wf-twl-cls C*›
using *struct-D' struct-C* **by** *blast*

have *H*: ‹⋀*C*. *C* ∈# *N*+*U* ⟹ ¬ *twl-is-an-exception C Q WS* ⟹ *C* ≠ *D* ⟹
twl-lazy-update M C›
using *twl*
by (*auto simp add*: *twl-st-inv.simps twl-is-an-exception-add-mset-to-clauses-to-update*)
have ‹*watched-literals-false-of-max-level M C*› **if** ‹*C* ≠ *?D*›
using *twl C that N'U'* **by** (*fastforce simp*: *twl-st-inv.simps elim*!: *update-clausesE*
dest: *in-diffD*)
moreover have ‹*watched-literals-false-of-max-level M ?D*›
using *w-max-D D watched L' uK-M distinct-consistent-interp*[*OF n-d*] *uL K*
apply (*cases D*)
apply (*simp-all add*: *add-mset-eq-add-mset consistent-interp-def*)
by (*metis add-mset-eq-add-mset*)
ultimately show ‹*watched-literals-false-of-max-level M C*›
by *blast*

assume *excep*: ‹¬*twl-is-an-exception C Q WS*›
have ‹*get-level M L* = *count-decided M*› **and** *L*: ‹−*L* ∈ *lits-of-l M*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*›
using *valid* **by** *auto*

have *excep-WS*: ‹¬ *twl-is-an-exception C Q WS*›
using *excep C* **by** (*force simp*: *twl-is-an-exception-def*)
have *excep-inv-D*: ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*)
WS, *Q*) *D*›
using *twl-excep D-N-U* **unfolding** *twl-st-exception-inv.simps*
by *blast*
then have ‹¬ *has-blit M* (*clause D*) *L* ⟹
L ∉# *Q* ⟹ (*L*, *D*) ∉# *add-mset* (*L*, *D*) *WS* ⟹ (∀ *K*∈#*unwatched D*. − *K* ∈ *lits-of-l M*)›
using *watched L*
unfolding *twl-exception-inv.simps*
apply *auto*
done
have *NU-WS*: ‹*Pair L* ‘# {#*C* ∈# *N*+*U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*,
D) *WS*›
using *wq* **by** *auto*
have ‹*distinct-mset* {#− *lit-of x*. *x* ∈# *mset M*#}›
by (*subst distinct-image-mset-inj*)
(*use n-d* **in** ‹*auto simp*: *lit-of-inj-on-no-dup distinct-map no-dup-def*›)
moreover have ‹*add-mset L Q* ⊆# {#− *lit-of x*. *x* ∈# *mset M*#}›
using *no-dup* **by** *auto*

ultimately have *LQ*[*simp*]: ‹*L* ∉# *Q*›
  by (*metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add*)

have ‹*twl-lazy-update M C*› **if** *CD*: ‹*C* = *D*›
  unfolding *twl-lazy-update.simps CD D*
proof (*intro conjI impI allI*)
  fix *K'*
  assume ‹*K'* ∈# *WD*› ‹− *K'* ∈ *lits-of-l M*›‹¬ *has-blit M* (*WD* + *UWD*) *K'*›
  have *C-D'*: ‹*C* ≠ *update-clause D L K*›
    using *D add-remove-WD that* **by** *auto*

  have *H*: ‹¬ *has-blit M* (*add-mset L* (*add-mset L' UWD*)) *L'* ⟹
     has-blit M (*add-mset L* (*add-mset L' UWD*)) *L* ⟹ *False*›
    using ‹− *K'* ∈ *lits-of-l M*› ‹*K'* ∈# *WD*› ‹¬ *has-blit M* (*WD* + *UWD*) *K'*›
      lev-L w-max-D
    using *L-M* **by** (*auto simp*: *has-blit-def D*)
  obtain *NU* **where** *NU*: ‹*N*+*U* = *add-mset D NU*›
    using *multi-member-split*[*OF D-N-U*] **by** *auto*
  have ‹*C* ∈# *remove1-mset D* (*N* + *U*)›
    using *C C-D' N'U'* **unfolding** *NU*
    apply (*auto simp*: *update-clauses.simps NU*[*symmetric*])
    using *C* **by** *auto*
  then obtain *NU'* **where** ‹*N*+*U* = *add-mset C* (*add-mset D NU'*)›
    using *NU multi-member-split* **by** *force*
  moreover have ‹*clauses-to-update-prop Q M* (*L*, *D*)›
    using *watched uL* ‹¬ *has-blit M* (*WD* + *UWD*) *K'*› ‹*K'* ∈# *WD*› *LQ*
    by (*auto simp*: *clauses-to-update-prop.simps D dest*: *H*)
  ultimately have ‹(*L*, *D*) ∈# *WS*›
    using *NU-WS* **by** (*auto simp*: *CD split*: *if-splits*)
  then have *False*
    using *excep* **unfolding** *CD*
    by (*auto simp*: *twl-is-an-exception-def*)
  then show ‹∀ *K*∈#*UWD*. *get-level M K* ≤ *get-level M K'* ∧ − *K* ∈ *lits-of-l M*›
    by *fast*
qed

moreover have ‹*twl-lazy-update M C*› **if** ‹*C* ≠ *?D*› ‹*C* ≠ *D*›
  using *H*[*of C*] *that excep-WS* ∗ *C*
  by (*auto simp add*: *twl-st-inv.simps*)[]
moreover {
  have *D'*: ‹*?D* = *TWL-Clause* {#*K*, *L'*#} (*add-mset L* (*remove1-mset K UWD*))› **and**
    mset-D': ‹{#*K*, *L'*#} + *add-mset L* (*remove1-mset K UWD*) = *clause D*›
    using *D watched cls-D-D* **by** *auto*
  have *lev-L'*: ‹*get-level M L'* = *count-decided M*› **if** ‹− *L'* ∈ *lits-of-l M* › **and**
    ‹¬ *has-blit M* (*clause D*) *L'*›
    using *L-M w-max-D D watched L' uL that*
    by *simp*
  have ‹∀ *C*. *C* ∈# *WS* ⟶ *fst C* = *L*›
    using *no-dup*
    using *watched uL L' undef D*
    by (*auto simp del*: *set-mset-union simp*:)
  then have ‹(*L'*, *TWL-Clause* {#*L*, *L'*#} *UWD*) ∉# *WS*›
    using *wq multi-member-split*[*OF D-N-U*] *struct-D*
    using *watched uL L' undef D*
    by *auto*
  then have ‹− *L'* ∈ *lits-of-l M* ⟹ ¬ *has-blit M* (*add-mset L* (*add-mset L' UWD*)) *L'* ⟹

```
        L' ∈# Q ›
    using wq multi-member-split[OF D-N-U] struct-D
    using watched uL L' undef D
    by (auto simp del: set-mset-union simp: )
  then have
    H: ‹– L' ∈ lits-of-l M ⟹ ¬ has-blit M (add-mset L (add-mset L' UWD)) L' ⟹
      False› if ‹C = ?D›
    using excep multi-member-split[OF D-N-U] struct-D
    using watched uL L' undef D that
    by (auto simp del: set-mset-union simp: twl-is-an-exception-def)

  have in-remove1-mset: ‹K' ∈# remove1-mset K UWD ⟷ K' ≠ K ∧ K' ∈# UWD› for K'
    using struct-D L'-L'-UWD by (auto simp: D in-remove1-mset-neq dest: in-diffD)
  have ‹twl-lazy-update M ?D› if ‹C = ?D›
    using watched uL L' undef D w-max-D H
    unfolding twl-lazy-update.simps D' mset-D' that
    by (auto simp: uK-M D add-mset-eq-add-mset lev-L count-decided-ge-get-level
        in-remove1-mset twl-is-an-exception-def)
  }
 ultimately show ‹twl-lazy-update M C›
  by blast
qed
qed

lemma twl-cp-no-duplicate-queued:
  assumes
    cdcl: ‹cdcl-twl-cp S T› and
    no-dup: ‹no-duplicate-queued S›
  shows ‹no-duplicate-queued T›
  using cdcl no-dup
proof (induction rule: cdcl-twl-cp.induct)
  case (pop M N U NE UE NS US N0 U0 L Q)
  then show ?case
    by (auto simp: image-Un image-image subset-mset.less-imp-le
        dest: mset-subset-eq-insertD)
qed auto

lemma distinct-mset-Pair: ‹distinct-mset (Pair L '# C) ⟷ distinct-mset C›
  by (induction C) auto

lemma distinct-image-mset-clause:
  ‹distinct-mset (clause '# C) ⟹ distinct-mset C›
  by (induction C) auto

lemma twl-cp-distinct-queued:
  assumes
    cdcl: ‹cdcl-twl-cp S T› and
    twl: ‹twl-st-inv S› and
    valid: ‹valid-enqueued S› and
    inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)› and
    no-dup: ‹no-duplicate-queued S› and
    dist: ‹distinct-queued S›
  shows ‹distinct-queued T›
  using cdcl twl valid inv no-dup dist
proof (induction rule: cdcl-twl-cp.induct)
  case (pop M N U NE UE NS US N0 U0 L Q) note c-dist = this(4) and dist = this(5)
```

```
  show ?case
    using dist by (auto simp: distinct-mset-Pair count-image-mset-Pair simp del: image-mset-union)
next
  case (propagate D L L' M N U NE UE NS US N0 U0 WS Q) note watched = this(1) and undef =
this(2) and
    twl = this(4) and valid = this(5)  and inv = this(6) and no-dup = this(7)
    and dist = this(8)
  have ‹L' ∉ lits-of-l M›
    using Decided-Propagated-in-iff-in-lits-of-l propagate.hyps(2) by auto
  then have ‹-L' ∉# Q›
    using no-dup by (fastforce simp: lits-of-def dest!: mset-subset-eqD)
  then show ?case
    using dist by (auto simp: all-conj-distrib split: if-splits dest!: Suc-leD)
next
  case (conflict D L L' M N U NE UE N0 U0 WS Q) note dist = this(8)
  then show ?case
    by auto
next
  case (delete-from-working D L L' M N U NE UE NS US N0 U0 WS Q) note dist = this(7)
  show ?case using dist by (auto simp: all-conj-distrib split: if-splits dest!: Suc-leD)
next
  case (update-clause D L L' M K N U N' U' NE UE NS US N0 U0 WS Q) note watched = this(1)
and uL = this(2) and
    L' = this(3) and K = this(4) and undef = this(5) and N'U' = this(6) and twl = this(7) and
    valid = this(8) and inv = this(9) and no-dup = this(10) and dist = this(11)

  show ?case
    unfolding distinct-queued.simps
  proof (intro conjI allI)
    show ‹distinct-mset Q›
      using dist N'U' by (auto simp: all-conj-distrib split: if-splits intro: le-SucI)

    fix K'' C
    have LD: ‹Suc (count WS (L, D)) ≤ count N D + count U D›
      using dist N'U' by (auto split: if-splits)
    have LC: ‹count WS (La, Ca) ≤ count N Ca + count U Ca›
      if ‹(La , Ca) ≠ (L, D)› for Ca La
      using dist N'U' by (force simp: all-conj-distrib split: if-splits intro: le-SucI)
    show ‹count WS (K'', C) ≤ count (N' + U') C›
    proof (cases ‹K'' ≠ L›)
      case True
      then have ‹count WS (K'', C) = 0›
      using no-dup by auto
      then show ?thesis by arith
    next
      case False
      then show ?thesis
        apply (cases ‹C = D›)
        using LD N'U' apply (auto simp: all-conj-distrib elim!: update-clausesE intro: le-SucI;
            fail)
        using LC[of L C] N'U' by (auto simp: all-conj-distrib elim!: update-clausesE intro: le-SucI)
    qed
  qed
qed

lemma twl-cp-valid:
```

assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of S*)› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
dist: ‹*distinct-queued S*›
shows ‹*valid-enqueued T*›
using *cdcl twl valid inv no-dup dist*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE NS US N0 U0 L Q*) **note** *valid* = *this*(*2*)
then show *?case*
by (*auto simp del*: *filter-union-mset*)
next
case (*propagate D L L′ M N U NE UE NS US N0 U0 WS Q*) **note** *watched* = *this*(*1*) **and** *twl* = *this*(*4*) **and**
valid = *this*(*5*) **and** *inv* = *this*(*6*) **and** *no-taut* = *this*(*7*)
show *?case*
using *valid* **by** (*auto dest*: *mset-subset-eq-insertD simp*: *get-level-cons-if*)
next
case (*conflict D L L′ M N U NE UE NS US N0 U0 WS Q*) **note** *valid* = *this*(*5*)
then show *?case*
by *auto*
next
case (*delete-from-working D L L′ M N U NE UE NS US N0 U0 WS Q*) **note** *watched* = *this*(*1*) **and** *L′* = *this*(*2*) **and**
twl = *this*(*3*) **and** *valid* = *this*(*4*) **and** *inv* = *this*(*5*)
show *?case* **unfolding** *twl-st-simps Ball-def*
using *valid* **by** (*auto dest*: *mset-subset-eq-insertD*)
next
case (*update-clause D L L′ M K N U N′ U′ NE UE NS US N0 U0 WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*) **and**
L′ = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N′U′* = *this*(*6*) **and** *twl* = *this*(*7*) **and**
valid = *this*(*8*) **and** *inv* = *this*(*9*) **and** *no-dup* = *this*(*10*) **and** *dist* = *this*(*11*)
show *?case*
unfolding *valid-enqueued.simps Ball-def*
proof (*intro allI impI conjI*)
fix *L* :: ‹*'a literal*›
assume *L*: ‹*L* ∈# *Q*›
then show ‹−*L* ∈ *lits-of-l M*›
using *valid* **by** *auto*
show ‹*get-level M L* = *count-decided M*›
using *L valid* **by** *auto*
next
fix *KC* :: ‹*'a literal* × *'a twl-cls*›
assume *LC-WS*: ‹*KC* ∈# *WS*›
obtain *K″ C* **where** *LC*: ‹*KC* = (*K″*, *C*)› **by** (*cases KC*)
have ‹*K″* ∈# *watched C*›
using *LC-WS valid LC* **by** *auto*
have *C-ne-D*: ‹*case KC of* (*L*, *C*) ⇒ *L* ∈# *watched C* ∧ *C* ∈# *N′* + *U′* ∧ − *L* ∈ *lits-of-l M* ∧
get-level M L = *count-decided M*› **if** ‹*C* ≠ *D*›
by (*cases* ‹*C* = *D*›)
(*use valid LC LC-WS N′U′ that* **in** ‹*auto simp*: *in-remove1-mset-neq elim*!: *update-clausesE*›)
have *K″-L*: ‹*K″* = *L*›
using *no-dup LC-WS LC* **by** *auto*
have ‹*Suc* (*count WS* (*L*, *D*)) ≤ *count N D* + *count U D*›

```
    using dist by (auto simp: all-conj-distrib split: if-splits)
  then have D-DN-U: ‹D ∈# remove1-mset D (N+U)› if [simp]: ‹C = D›
    using LC-WS unfolding count-greater-zero-iff[symmetric]
    by (auto simp del: count-greater-zero-iff simp: LC K''-L)
  have D-D-N: ‹D ∈# remove1-mset D N› if ‹D ∈# N› and ‹D ∉# U› and [simp]: ‹C = D›
  proof −
    have ‹D ∈# remove1-mset D (U + N)›
      using D-DN-U by (simp add: union-commute)
    then have ‹D ∈# U + remove1-mset D N›
      using that(1) by (metis (no-types) add-mset-remove-trivial insert-DiffM
          union-mset-add-mset-right)
    then show ‹D ∈# remove1-mset D N›
      using that(2) by (meson union-iff)
  qed
  have D-D-U: ‹D ∈# remove1-mset D U› if ‹D ∈# U› and ‹D ∉# N› and [simp]: ‹C = D›
  proof −
    have ‹D ∈# remove1-mset D (U + N)›
      using D-DN-U by (simp add: union-commute)
    then have ‹D ∈# N + remove1-mset D U›
      using D-DN-U that(1) by fastforce
    then show ‹D ∈# remove1-mset D U›
      using that(2) by (meson union-iff)
  qed
  have CD: ‹case KC of (L, C) ⇒ L ∈# watched C ∧ C ∈# N' + U' ∧ − L ∈ lits-of-l M ∧
    get-level M L = count-decided M› if ‹C = D›
    by (use valid LC-WS N'U' in ‹auto simp: LC D-D-N that in-remove1-mset-neq
      dest!: D-D-U elim!: update-clausesE›)
  show ‹case KC of (L, C) ⇒ L ∈# watched C ∧ C ∈# N' + U' ∧ − L ∈ lits-of-l M ∧
    get-level M L = count-decided M›
    using CD C-ne-D by blast
 qed
qed


lemma twl-cp-propa-cands-enqueued:
 assumes
  cdcl: ‹cdcl-twl-cp S T› and
  twl: ‹twl-st-inv S› and
  valid: ‹valid-enqueued S› and
  inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)› and
  twl-excep: ‹twl-st-exception-inv S› and
  no-dup: ‹no-duplicate-queued S› and
  cands: ‹propa-cands-enqueued S› and
  ws: ‹clauses-to-update-inv S›
 shows ‹propa-cands-enqueued T›
 using cdcl twl valid inv twl-excep no-dup cands ws
proof (induction rule: cdcl-twl-cp.induct)
 case (pop M N U NE UE NS US N0 U0 L Q) note inv = this(1) and valid = this(2) and cands =
this(6)
 show ?case unfolding propa-cands-enqueued.simps
 proof (intro allI conjI impI)
  fix C K
  assume C: ‹C ∈# N + U› and
   ‹K ∈# clause C› and
   ‹M ⊨as CNot (remove1-mset K (clause C))› and
   ‹undefined-lit M K›
```

```
    then have ‹(∃ L'. L' ∈# watched C ∧ L' ∈# add-mset L Q)›
      using cands by auto
    then show
      ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨
        (∃ La. (La, C) ∈# Pair L `# {#C ∈# N + U. L ∈# watched C#})›
      using C by auto
  qed
next
  case (propagate D L L' M N U NE UE NS US N0 U0 WS Q) note watched = this(1) and undef =
this(2) and
    false = this(3) and
    twl = this(4) and valid = this(5) and inv = this(6) and excep = this(7)
    and no-dup = this(8) and cands = this(9) and to-upd = this(10)
  have uL'-M: ‹- L' ∉ lits-of-l M›
    using Decided-Propagated-in-iff-in-lits-of-l propagate.hyps(2) by blast
  have D-N-U: ‹D ∈# N + U›
    using valid by auto
  then have wf-D: ‹struct-wf-twl-cls D›
    using twl by (simp add: twl-st-inv.simps)
  show ?case unfolding propa-cands-enqueued.simps
  proof (intro allI conjI impI)
    fix C K
    assume C: ‹C ∈# N + U› and
      K: ‹K ∈# clause C› and
      L'-M-C: ‹Propagated L' (clause D) # M ⊨as CNot (remove1-mset K (clause C))› and
      undef-K: ‹undefined-lit (Propagated L' (clause D) # M) K›
    then have wf-C: ‹struct-wf-twl-cls C›
      using twl by (simp add: twl-st-inv.simps)
    have undef-K-M: ‹undefined-lit M K›
      using undef-K by (simp add: Decided-Propagated-in-iff-in-lits-of-l)
    consider
      (no-L') ‹M ⊨as CNot (remove1-mset K (clause C))› |
      (L') ‹-L' ∈# remove1-mset K (clause C)›
      using L'-M-C ‹- L' ∉ lits-of-l M›
      by (metis insertE list.simps(15) lit-of.simps(2) lits-of-insert
          true-annots-CNot-lit-of-notin-skip true-annots-true-cls-def-iff-negation-in-model)
    then show ‹(∃ L'a. L'a ∈# watched C ∧ L'a ∈# add-mset (- L') Q) ∨ (∃ L. (L, C) ∈# WS)›
    proof cases
      case no-L'
      then have ‹(∃ L'. L' ∈# watched C ∧ L' ∈#  Q) ∨ (∃ La. (La, C) ∈# add-mset (L, D) WS)›
        using cands C K undef-K-M by auto
      moreover {
        have ‹K = L'› if ‹C = D›
          by (metis ‹- L' ∉ lits-of-l M› add-mset-add-single clause.simps in-CNot-implies-uminus(2)
              in-remove1-mset-neq multi-member-this no-L' that twl-clause.exhaust twl-clause.sel(1)
              union-iff watched)
        then have False if ‹C = D›
          using undef-K by (simp add: Decided-Propagated-in-iff-in-lits-of-l that)
      }
      ultimately show ?thesis by auto
    next
      case L'
      have ?thesis if ‹L' ∈# watched C›
      proof -
        have ‹K = L'›
          using that L'-M-C ‹- L' ∉ lits-of-l M› L' undef
```

by (*metis clause.simps in-CNot-implies-uminus*(2) *in-lits-of-l-defined-litD*
in-remove1-mset-neq insert-iff list.simps(15) *lits-of-insert*
twl-clause.exhaust-sel uminus-not-id' uminus-of-uminus-id union-iff)
then have *False*
using *Decided-Propagated-in-iff-in-lits-of-l undef-K* **by** *force*
then show *?thesis*
by *fastforce*
qed

moreover have *?thesis* **if** *L'-C*: ‹*L'* ∉# *watched C*›
proof (*rule ccontr*, *clarsimp*)
assume
Q: ‹∀ *L'a*. *L'a* ∈# *watched C* ⟶ *L'a* ≠ − *L'* ∧ *L'a* ∉# *Q*› **and**
WS: ‹∀ *L*. (*L*, *C*) ∉# *WS*›
then have ‹¬ *twl-is-an-exception C* (*add-mset* (− *L'*) *Q*) *WS*›
by (*auto simp*: *twl-is-an-exception-def*)
moreover have
‹*twl-st-inv* (*Propagated L'* (*clause D*) # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*,
add-mset (− *L'*) *Q*)›
using *twl-cp-twl-inv*[*OF - twl valid inv excep no-dup to-upd*]
cdcl-twl-cp.propagate[*OF propagate*(1−3)] **by** *fast*
ultimately have ‹*twl-lazy-update* (*Propagated L'* (*clause D*) # *M*) *C*›
using *C* **by** (*auto simp*: *twl-st-inv.simps*)

have *CD*: ‹*C* ≠ *D*›
using *that watched* **by** *auto*
have *struct*: ‹*struct-wf-twl-cls C*›
using *twl C* **by** (*simp add*: *twl-st-inv.simps*)
obtain *a b W UW* **where**
C-W-UW: ‹*C* = *TWL-Clause W UW*› **and**
W: ‹*W* = {#*a*, *b*#}›
using *struct* **by** (*cases C*, *auto simp*: *size-2-iff*)
have *ua-or-ub*: ‹−*a* ∈ *lits-of-l M* ∨ −*b* ∈ *lits-of-l M*›
using *L'-M-C C-W-UW W* ‹∀ *L'a*. *L'a* ∈# *watched C* ⟶ *L'a* ≠ − *L'* ∧ *L'a* ∉# *Q*›
apply (*cases* ‹*K* = *a*›) **by** *fastforce*+

have ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps*)
then have [*dest*]: *False* **if** ‹*a* ∈ *lits-of-l M*› **and** ‹−*a* ∈ *lits-of-l M*› **for** *a*
using *consistent-interp-def distinct-consistent-interp that*(1) *that*(2) **by** *blast*
have *uab*: ‹*a* ∉ *lits-of-l M*› **if** ‹−*b* ∈ *lits-of-l M*›
using *L'-M-C C-W-UW W that undef-K-M uL'-M*
by (*cases* ‹*K* = *a*›) (*fastforce simp*: *Decided-Propagated-in-iff-in-lits-of-l*
simp del: *uL'-M*)+
have *uba*: ‹*b* ∉ *lits-of-l M*› **if** ‹−*a* ∈ *lits-of-l M*›
using *L'-M-C C-W-UW W that undef-K-M uL'-M*
by (*cases* ‹*K* = *b*›) (*fastforce simp*: *Decided-Propagated-in-iff-in-lits-of-l*
add-mset-commute[*of a b*])+
have [*simp*]: ‹−*a* ≠ *L'*› ‹−*b* ≠ *L'*›
using *Q W C-W-UW* **by** *fastforce*+
have *H'*: ‹∀ *La L'*. *watched C* = {#*La*, *L'*#} ⟶ − *La* ∈ *lits-of-l M* ⟶
¬*has-blit M* (*clause C*) *La* ⟶ *L'* ∉ *lits-of-l M* ⟶
(∀ *K*∈#*unwatched C*. − *K* ∈ *lits-of-l M*)›
using *excep C CD Q W WS uab uba* **by** (*auto simp*: *twl-exception-inv.simps simp del*:
set-mset-union

```
        dest: multi-member-split)
      moreover have ‹watched C = {#La, L''#} ⟶ − La ∈ lits-of-l M ⟶ ¬has-blit M (clause C)
La› for La L''
        using in-CNot-implies-uminus[OF - L'-M-C]  wf-C L' uL'-M undef-K-M undef uab uba
        unfolding C-W-UW has-blit-def apply −
        apply (cases ‹La = K›)
         apply (auto simp: has-blit-def Decided-Propagated-in-iff-in-lits-of-l W
           add-mset-eq-add-mset in-remove1-mset-neq)
        apply (metis ‹⋀a. ⟦a ∈ lits-of-l M; − a ∈ lits-of-l M⟧ ⟹ False› add-mset-remove-trivial
           defined-lit-uminus in-lits-of-l-defined-litD in-remove1-mset-neq undef)
        apply (metis ‹⋀a. ⟦a ∈ lits-of-l M; − a ∈ lits-of-l M⟧ ⟹ False› add-mset-remove-trivial
           defined-lit-uminus in-lits-of-l-defined-litD in-remove1-mset-neq undef)
        done
      ultimately have ‹∀ K∈#unwatched C. − K ∈ lits-of-l M›
        using uab uba W C-W-UW ua-or-ub wf-C unfolding C-W-UW
        by (auto simp: add-mset-eq-add-mset )
      then show False
        by (metis Decided-Propagated-in-iff-in-lits-of-l L' uminus-lit-swap
           Q clause.simps in-diffD propagate.hyps(2) twl-clause.collapse union-iff)
    qed

    ultimately show ?thesis by fast
  qed
 qed
next
 case (conflict D L L' M N U NE UE NS US N0 U0 WS Q) note cands = this(10)
 then show ?case
  by auto
next
 case (delete-from-working L' D M N U NE UE NS US N0 U0 L WS Q) note watched = this(1) and
L' = this(2)
  and twl = this(3) and valid = this(4) and inv = this(5) and cands = this(8) and ws = this(9)
 have n-d: ‹no-dup M›
  using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
   cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (simp add: trail.simps)
 show ?case unfolding propa-cands-enqueued.simps
 proof (intro allI conjI impI)
  fix C K
  assume C: ‹C ∈# N + U› and
   K: ‹K ∈# clause C› and
   L'-M-C: ‹M ⊨as CNot (remove1-mset K (clause C))› and
   undef-K: ‹undefined-lit M K›
  then have ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ La. La = L ∧ C = D ∨ (La, C) ∈# WS)›
   using cands by auto
  moreover have False if [simp]: ‹C = D›
   using L' L'-M-C undef-K watched
   using Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def distinct-consistent-interp
    local.K n-d K
   by (cases D)
    (auto 5 5 simp: true-annots-true-cls-def-iff-negation-in-model add-mset-eq-add-mset
      dest: in-lits-of-l-defined-litD no-dup-consistentD dest!: multi-member-split)
  ultimately show ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ L. (L, C) ∈# WS)›
   by auto
 qed
next
 case (update-clause D L L' M K N U N' U' NE UE NS US N0 U0 WS Q) note watched = this(1)
```

and *uL* = *this*(*2*)
  and *L′* = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N′U′* = *this*(*6*) **and** *twl* = *this*(*7*)
and
  valid = *this*(*8*) **and** *inv* = *this*(*9*) **and** *twl-excep* = *this*(*10*) **and** *no-dup* = *this*(*11*) **and**
  cands = *this*(*12*) **and** *ws* = *this*(*13*)
 obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
 have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
  using *valid* **by** *auto*
 then have *struct-D*: ‹*struct-wf-twl-cls D*›
  using *twl* **by** (*auto simp*: *twl-st-inv.simps*)
 have *L′-UWD*: ‹*L* ∉# *remove1-mset L′ UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
 proof (*rule ccontr*)
  assume ‹¬ *?thesis*›
  then have ‹*count UWD L* ≥ *1*›
   by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
     split: *if-splits*)
  then have ‹*count* (*clause D*) *L* ≥ *2*›
   using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
     split: *if-splits*)
  moreover have ‹*distinct-mset* (*clause D*)›
   using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
  ultimately show *False*
   unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
 qed
 have *L′-L′-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
 proof (*rule ccontr*)
  assume ‹¬ *?thesis*›
  then have ‹*count UWD K* ≥ *2*›
   by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
     split: *if-splits*)
  then have ‹*count* (*clause D*) *K* ≥ *2*›
   using *D L′* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
     split: *if-splits*)
  moreover have ‹*distinct-mset* (*clause D*)›
   using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
  ultimately show *False*
   unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
 qed
 have ‹*watched-literals-false-of-max-level M D*›
  using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
 let *?D* = ‹*update-clause D L K*›
 have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N′* + *U′*› **for** *C*
  using *C N′U′ that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
 have *n-d*: ‹*no-dup M*›
  using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
   cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
 then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
  using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
   distinct-consistent-interp **by** *blast*
 have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
  using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
 have *D-N-U*: ‹*D* ∈# *N* + *U*›
  using *N′U′ D uK-M uL D-N-U* **by** (*auto simp*: *add-mset-remove-trivial-iff split*: *if-splits*)
 have *D-ne-D*: ‹*D* ≠ *update-clause D L K*›
  using *D add-remove-WD* **by** *auto*

have *L-M*: ‹*L* ∉ *lits-of-l M*›
using *n-d uL* **by** (*fastforce dest*!: *distinct-consistent-interp*
simp: *consistent-interp-def lits-of-def uminus-lit-swap*)
have *w-max-D*: ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)

have *clause-D*: ‹*clause ?D* = *clause D*›
using *D K watched* **by** *auto*
show *?case* **unfolding** *propa-cands-enqueued.simps*
proof (*intro allI conjI impI*)
fix *C K2*
assume *C*: ‹*C* ∈# *N'* + *U'*› **and**
K: ‹*K2* ∈# *clause C*› **and**
L'-M-C: ‹*M* ⊨*as CNot* (*remove1-mset K2* (*clause C*))› **and**
undef-K: ‹*undefined-lit M K2*›
then have ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *La*. (*La*, *C*) ∈# *WS*)› **if** ‹*C* ≠ *?D*› ‹*C* ≠ *D*›
using *cands* ∗[*OF that*(1) *C*] *that*(2) **by** *auto*
moreover have ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)› **if** [*simp*]: ‹*C* = *?D*›
proof (*rule ccontr*)
have ‹*K* ∉ *lits-of-l M*›
by (*metis D Decided-Propagated-in-iff-in-lits-of-l L'-M-C add-diff-cancel-left'*
clause.simps clause-D in-diffD in-remove1-mset-neq that
true-annots-true-cls-def-iff-negation-in-model twl-clause.sel(2) *uK-M undef-K*
update-clause.hyps(4))
moreover have ‹∀ *L*∈#*remove1-mset K2* (*clause ?D*). *defined-lit M L*›
using *L'-M-C* **unfolding** *true-annots-true-cls-def-iff-negation-in-model*
by (*auto simp*: *clause-D Decided-Propagated-in-iff-in-lits-of-l*)
ultimately have [*simp*]: ‹*K2* = *K*›
using *undef undef-K K* **unfolding** *that clause-D*
by (*metis D clause.simps in-remove1-mset-neq twl-clause.sel*(2) *union-iff*
update-clause.hyps(4))

have *uL'-M*: ‹− *L'* ∈ *lits-of-l M*›
using *D watched L'-M-C* **by** *auto*
have [*simp*]: ‹*L* ≠ *L'*› ‹*L'* ≠ *L*›
using *struct-D D watched* **by** *auto*

assume ‹¬ ((∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))›
then have [*simp*]: ‹*L'* ∉# *Q*› **and** *L'-C-WS*: ‹(*L'*, *C*) ∉# *WS*›
using *watched D* **by** *auto*
have ‹*C* ∈# *add-mset* (*L*, *TWL-Clause WD UWD*) *WS* ⟶
C' ∈# *add-mset* (*L*, *TWL-Clause WD UWD*) *WS* ⟶
fst C = *fst C'*› **for** *C C'*
using *no-dup* **unfolding** *D no-duplicate-queued.simps*
by *blast*
from *this*[*of* ‹(*L*, *TWL-Clause WD UWD*)› ‹(*L'*, *TWL-Clause* {#*L*, *L'*#} *UWD*)›]
have *notin*: ‹*False*› **if** ‹(*L'*, *TWL-Clause* {#*L*, *L'*#} *UWD*) ∈# *WS*›
using *struct-D watched that* **unfolding** *D*
by *auto*
have ‹*?D* ≠ *D*›
using *C D watched L K uK-M uL* **by** *auto*
then have *excep*: ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*) *D*›
using *twl-excep* ∗[*of D*] *D-N-U* **by** (*auto simp*: *twl-st-inv.simps*)
moreover have ‹*D* = *TWL-Clause* {#*L*, *L'*#} *UWD* ⟹

```
        WD = {#L, L'#} ⟹
        ∀ L∈#remove1-mset K UWD.
          − L ∈ lits-of-l M ⟹
        ¬has-blit M (add-mset L (add-mset L' UWD)) L'›
      using uL uL'-M n-d ‹K ∉ lits-of-l M› unfolding has-blit-def
      apply (auto dest:no-dup-consistentD simp: in-remove1-mset-neq Ball-def)
      by (metis in-remove1-mset-neq no-dup-consistentD)
    ultimately have ‹∀ K ∈# unwatched D. −K ∈ lits-of-l M›
      using D watched L'-M-C L'-C-WS
      by (auto simp: add-mset-eq-add-mset uL'-M L-M uL twl-exception-inv.simps
          true-annots-true-cls-def-iff-negation-in-model dest: in-diffD notin)
    then show False
      using uK-M update-clause.hyps(4) by blast
  qed
  moreover have ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ L. (L, C) ∈# WS)› if [simp]: ‹C = D›
    unfolding that
  proof −
    have n-d: ‹no-dup M›
      using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
        cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps)
    obtain NU where NU: ‹N + U = add-mset D NU›
      by (metis D-N-U insert-DiffM)
    have N'U': ‹N' + U' = add-mset ?D (remove1-mset D (N + U))›
      using N'U' D-N-U by (auto elim!: update-clausesE)

    have ‹add-mset L Q ⊆# {#− lit-of x. x ∈# mset M#}›
      using no-dup by (auto)
    moreover have ‹distinct-mset {#− lit-of x. x ∈# mset M#}›
      by (subst distinct-image-mset-inj)
        (use n-d in ‹auto simp: lit-of-inj-on-no-dup distinct-map no-dup-def›)
    ultimately have [simp]: ‹L ∉# Q›
      by (metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add)
    have ‹has-blit M (clause D) L ⟹ False›
      by (smt K L'-M-C has-blit-def in-lits-of-l-defined-litD insert-DiffM insert-iff
          is-blit-def n-d no-dup-consistentD set-mset-add-mset-insert that
          true-annots-true-cls-def-iff-negation-in-model undef-K)
    then have w-q-p-D: ‹clauses-to-update-prop Q M (L, D)›
      by (auto simp: clauses-to-update-prop.simps watched)
        (use uL undef L' in ‹auto simp: Decided-Propagated-in-iff-in-lits-of-l›)
    have ‹Pair L `# {#C ∈# add-mset D NU. clauses-to-update-prop Q M (L, C)#} ⊆#
      add-mset (L, D) WS›
      using ws no-dup unfolding clauses-to-update-inv.simps NU
      by (auto simp: all-conj-distrib)
    then have IH: ‹Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#} ⊆# WS›
      using w-q-p-D by auto
    moreover have ‹(L, D) ∈# Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#}›
      using C D-ne-D w-q-p-D unfolding NU N'U' by (auto simp: pair-in-image-Pair)
    ultimately show ‹(∃ L'. L' ∈# watched D ∧ L' ∈# Q) ∨ (∃ L. (L, D) ∈# WS)›
      by blast
  qed
  ultimately show ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ L. (L, C) ∈# WS)›
    by auto
 qed
qed
```

lemma *twl-cp-confl-cands-enqueued*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of S*)› **and**
excep: ‹*twl-st-exception-inv S*› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
cands: ‹*confl-cands-enqueued S*› **and**
ws: ‹*clauses-to-update-inv S*›
shows
‹*confl-cands-enqueued T*›
using *cdcl*
proof (*induction rule*: *cdcl-twl-cp.cases*)
case (*pop M N U NE UE NS US N0 U0 L Q*) **note** *S* = *this*(*1*) **and** *T* = *this*(*2*)
show *?case* **unfolding** *confl-cands-enqueued.simps Ball-def S T*
proof (*intro allI conjI impI*)
fix *C K*
assume *C*: ‹*C* ∈# *N* + *U*› **and**
‹*M* ⊨*as CNot* (*clause C*)›
then have ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *add-mset L Q*)›
using *cands S* **by** *auto*
then show
‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨
(∃ *La*. (*La*, *C*) ∈# *Pair L* ‘# {#*C* ∈# *N* + *U*. *L* ∈# *watched C*#})›
using *C* **by** *auto*
qed
next
case (*propagate D L L' M N U NE UE NS US N0 U0 WS Q*) **note** *S* = *this*(*1*) **and** *T* = *this*(*2*)
and *watched* = *this*(*3*)
and *undef* = *this*(*4*)
have *uL'-M*: ‹− *L'* ∉ *lits-of-l M*›
using *Decided-Propagated-in-iff-in-lits-of-l undef* **by** *blast*
have *D-N-U*: ‹*D* ∈# *N* + *U*›
using *valid S* **by** *auto*
then have *wf-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*simp add*: *twl-st-inv.simps S*)
show *?case* **unfolding** *confl-cands-enqueued.simps Ball-def S T*
proof (*intro allI conjI impI*)
fix *C K*
assume *C*: ‹*C* ∈# *N* + *U*› **and**
L'-M-C: ‹*Propagated L'* (*clause D*) # *M* ⊨*as CNot* (*clause C*)›
consider
(*no-L'*) ‹*M* ⊨*as CNot* (*clause C*)›
| (*L'*) ‹−*L'* ∈# *clause C*›
using *L'-M-C* ‹− *L'* ∉ *lits-of-l M*›
by (*metis insertE list.simps*(*15*) *lit-of.simps*(*2*) *lits-of-insert*
true-annots-CNot-lit-of-notin-skip true-annots-true-cls-def-iff-negation-in-model)
then show ‹(∃ *L'a*. *L'a* ∈# *watched C* ∧ *L'a* ∈# *add-mset* (− *L'*) *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)›
proof *cases*
case *no-L'*
then have ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *La*. (*La*, *C*) ∈# *add-mset* (*L*, *D*) *WS*)›
using *cands C* **by** (*auto simp*: *S*)
moreover {
have ‹*C* ≠ *D*›
by (*metis* ‹− *L'* ∉ *lits-of-l M*› *add-mset-add-single clause.simps in-CNot-implies-uminus*(*2*)

```
      multi-member-this no-L' twl-clause.exhaust twl-clause.sel(1)
      union-iff watched)
  }
  ultimately show ?thesis by auto
next
  case L'
  have L'-C: ‹L' ∉# watched C›
    using L'-M-C ‹- L' ∉ lits-of-l M›
    by (metis (no-types, opaque-lifting) Decided-Propagated-in-iff-in-lits-of-l L' clause.simps
      in-CNot-implies-uminus(2) insertE list.simps(15) lits-of-insert twl-clause.exhaust-sel
      uminus-not-id' uminus-of-uminus-id undef union-iff)
  moreover have ?thesis
  proof (rule ccontr, clarsimp)
    assume
      Q: ‹∀ L'a. L'a ∈# watched C ⟶ L'a ≠ - L' ∧ L'a ∉# Q› and
      WS: ‹∀ L. (L, C) ∉# WS›
    then have ‹¬ twl-is-an-exception C (add-mset (- L') Q) WS›
      by (auto simp: twl-is-an-exception-def)
    moreover have
      ‹twl-st-inv (Propagated L' (clause D) # M, N, U, None, NE, UE, NS, US, N0, U0, WS,
        add-mset (- L') Q)›
      using twl-cp-twl-inv[OF - twl valid inv excep no-dup ws] cdcl unfolding S T by fast
    ultimately have ‹twl-lazy-update (Propagated L' (clause D) # M) C›
      using C by (auto simp: twl-st-inv.simps)

    have struct: ‹struct-wf-twl-cls C›
      using twl C by (simp add: twl-st-inv.simps S)
    have CD: ‹C ≠ D›
      using L'-C watched by auto
    have struct: ‹struct-wf-twl-cls C›
      using twl C by (simp add: twl-st-inv.simps S)
    obtain a b W UW where
      C-W-UW: ‹C = TWL-Clause W UW› and
      W: ‹W = {#a, b#}›
      using struct by (cases C) (auto simp: size-2-iff)
    have ua-ub: ‹-a ∈ lits-of-l M ∨ -b ∈ lits-of-l M›
      using L'-M-C C-W-UW W ‹∀ L'a. L'a ∈# watched C ⟶ L'a ≠ - L' ∧ L'a ∉# Q›
      by (cases ‹K = a›) fastforce+

    have ‹no-dup M›
      using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
        cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (simp add: trail.simps S)
    then have [dest]: False if ‹a ∈ lits-of-l M› and ‹-a ∈ lits-of-l M› for a
      using consistent-interp-def distinct-consistent-interp that(1) that(2) by blast
    have uab: ‹a ∉ lits-of-l M› if ‹-b ∈ lits-of-l M›
      using L'-M-C C-W-UW W that uL'-M by (cases ‹K = a›) auto
    have uba: ‹b ∉ lits-of-l M› if ‹-a ∈ lits-of-l M›
      using L'-M-C C-W-UW W that uL'-M by (cases ‹K = b›) auto
    have [simp]: ‹-a ≠ L'› ‹-b ≠ L'›
      using ‹∀ L'a. L'a ∈# watched C ⟶ L'a ≠ - L' ∧ L'a ∉# Q› W C-W-UW
      by fastforce+
    have H': ‹∀ La L'. watched C = {#La, L'#} ⟶ - La ∈ lits-of-l M ⟶ L' ∉ lits-of-l M ⟶
      ¬ has-blit M (clause C) La ⟶(∀ K∈#unwatched C. - K ∈ lits-of-l M)›
      using excep C CD Q W WS uab uba
      by (auto simp: twl-exception-inv.simps S dest: multi-member-split)
    moreover have ‹¬ has-blit M (clause C) a› ‹¬ has-blit M (clause C) b›
```

using *multi-member-split*[*OF C*]
using *watched L′ undef L′-M-C*
unfolding *has-blit-def*
by (*metis* (*no-types*, *lifting*) *Clausal-Logic.uminus-lit-swap*
‹⋀*a*. ⟦*a* ∈ *lits-of-l M*; − *a* ∈ *lits-of-l M*⟧ ⟹ *False*› *in-CNot-implies-uminus*(*2*)
in-lits-of-l-defined-litD insert-iff is-blit-def list.set(*2*) *lits-of-insert uL′-M*)+
ultimately have ‹∀ *K*∈#*unwatched C*. − *K* ∈ *lits-of-l M*›
using *uab uba W C-W-UW ua-ub struct*
by (*auto simp*: *add-mset-eq-add-mset*)
then show *False*
by (*metis Decided-Propagated-in-iff-in-lits-of-l L′ uminus-lit-swap*
Q clause.simps undef twl-clause.collapse union-iff)
qed
ultimately show *?thesis* **by** *fast*
qed
qed
next
case (*conflict D L L′ M N U NE UE NS US N0 U0 WS Q*)
then show *?case*
by *auto*
next
case (*delete-from-working L′ D M N U NE UE NS US N0 U0 L WS Q*) **note** *S* = *this*(*1*) **and** *T* = *this*(*2*) **and**
watched = *this*(*3*) **and** *L′* = *this*(*4*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps S*)
show *?case* **unfolding** *confl-cands-enqueued.simps Ball-def S T*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N* + *U*› **and**
L′-M-C: ‹*M* ⊨*as CNot* (*clause C*)›
then have ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *Q*) ∨ (∃ *La*. *La* = *L* ∧ *C* = *D* ∨ (*La*, *C*) ∈# *WS*)›
using *cands S* **by** *auto*
moreover have *False* **if** [*simp*]: ‹*C* = *D*›
using *L′-M-C watched L′ n-d* **by** (*cases D*) (*auto dest*!: *distinct-consistent-interp*
simp: *consistent-interp-def dest*!: *multi-member-split*)
ultimately show ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)›
by *auto*
qed
next
case (*update-clause D L L′ M K N U N′ U′ NE UE NS US N0 U0 WS Q*) **note** *S* = *this*(*1*) **and** *T* = *this*(*2*) **and**
watched = *this*(*3*) **and** *uL* = *this*(*4*) **and** *L′* = *this*(*5*) **and** *K* = *this*(*6*) **and** *undef* = *this*(*7*) **and**
N′U′ = *this*(*8*)
obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid S* **by** *auto*
then have *struct-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp*: *twl-st-inv.simps S*)
have *L′-UWD*: ‹*L* ∉# *remove1-mset L′ UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD L* ≥ *1*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]

split: *if-splits*)
then have ‹*count* (*clause D*) *L* ≥ *2*›
using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have *L′-L′-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD K* ≥ *2*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *K* ≥ *2*›
using *D L′* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps S*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N′* + *U′*› **for** *C*
using *C N′U′ that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps S*)
then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
distinct-consistent-interp **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
have *D-N-U*: ‹*D* ∈# *N* + *U*›
using *N′U′ D uK-M uL D-N-U* **by** (*auto simp*: *add-mset-remove-trivial-iff split*: *if-splits*)

have *D-ne-D*: ‹*D* ≠ *update-clause D L K*›
using *D add-remove-WD* **by** *auto*

have *L-M*: ‹*L* ∉ *lits-of-l M*›
using *n-d uL* **by** (*fastforce dest*!: *distinct-consistent-interp*
simp: *consistent-interp-def lits-of-def uminus-lit-swap*)
have *w-max-D*: ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps S*)

have *clause-D*: ‹*clause ?D* = *clause D*›
using *D K watched* **by** *auto*

show *?case* **unfolding** *confl-cands-enqueued.simps Ball-def S T*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N′* + *U′*› **and**
L′-M-C: ‹*M* ⊨*as CNot* (*clause C*)›

```
    then have ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ La. (La, C) ∈# WS)› if ‹C ≠ ?D› ‹C ≠ D›
      using cands *[OF that(1) C] that(2) S by auto
    moreover have ‹C ≠ ?D›
      by (metis D L'-M-C add-diff-cancel-left' clause.simps clause-D in-diffD
          true-annots-true-cls-def-iff-negation-in-model twl-clause.sel(2) uK-M K)
    moreover have ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ La. (La, C) ∈# WS)› if [simp]: ‹C =
D›
      unfolding that
    proof −
      obtain NU where NU: ‹N + U = add-mset D NU›
        by (metis D-N-U insert-DiffM)
      have N'U': ‹N' + U' = add-mset ?D (remove1-mset D (N + U))›
        using N'U' D-N-U by (auto elim!: update-clausesE)

      have ‹add-mset L Q ⊆# {#− lit-of x. x ∈# mset M#}›
        using no-dup by (auto simp: S)
      moreover have ‹distinct-mset {#− lit-of x. x ∈# mset M#}›
        by (subst distinct-image-mset-inj)
          (use n-d in ‹auto simp: lit-of-inj-on-no-dup distinct-map no-dup-def›)
      ultimately have [simp]: ‹L ∉# Q›
        by (metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add)

      have ‹has-blit M (clause D) L ⟹ False›
        by (smt K L'-M-C has-blit-def in-lits-of-l-defined-litD insert-DiffM insert-iff
            is-blit-def n-d no-dup-consistentD set-mset-add-mset-insert that
            true-annots-true-cls-def-iff-negation-in-model)
      then have w-q-p-D: ‹clauses-to-update-prop Q M (L, D)›
        by (auto simp: clauses-to-update-prop.simps watched)
          (use uL undef L' in ‹auto simp: Decided-Propagated-in-iff-in-lits-of-l›)
      have ‹Pair L `# {#C ∈# add-mset D NU. clauses-to-update-prop Q M (L, C)#} ⊆#
        add-mset (L, D) WS›
        using ws no-dup unfolding clauses-to-update-inv.simps NU S
        by (auto simp: all-conj-distrib)
      then have IH: ‹Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#} ⊆# WS›
        using w-q-p-D by auto
      moreover have ‹(L, D) ∈# Pair L `# {#C ∈# NU. clauses-to-update-prop Q M (L, C)#}›
        using C D-ne-D w-q-p-D unfolding NU N'U' by (auto simp: pair-in-image-Pair)
      ultimately show ‹(∃ L'. L' ∈# watched D ∧ L' ∈# Q) ∨ (∃ L. (L, D) ∈# WS)›
        by blast
    qed
    ultimately show ‹(∃ L'. L' ∈# watched C ∧ L' ∈# Q) ∨ (∃ L. (L, C) ∈# WS)›
      by auto
  qed
qed

lemma twl-cp-past-invs:
  assumes
    cdcl: ‹cdcl-twl-cp S T› and
    twl: ‹twl-st-inv S› and
    valid: ‹valid-enqueued S› and
    inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)› and
    twl-excep: ‹twl-st-exception-inv S› and
    no-dup: ‹no-duplicate-queued S› and
    past-invs: ‹past-invs S›
  shows ‹past-invs T›
  using cdcl twl valid inv twl-excep no-dup past-invs
```

proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE NS US N0 U0 L Q*) **note** *past-invs* = *this*(*6*)
then show *?case*
by (*subst past-invs-enqueud*, *subst* (*asm*) *past-invs-enqueud*)
next
case (*propagate D L L′ M N U NE UE NS US N0 U0 WS Q*) **note** *watched* = *this*(*1*) **and** *twl* = *this*(*4*) **and**
valid = *this*(*5*) **and** *inv* = *this*(*6*) **and** *past-invs* = *this*(*9*)
have [*simp*]: ‹− *L′* ∉ *lits-of-l M*›
using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *blast*
have *D-N-U*: ‹*D* ∈# *N* + *U*›
using *valid* **by** *auto*
then have *wf-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*simp add*: *twl-st-inv.simps*)
show *?case* **unfolding** *past-invs.simps Ball-def*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N* + *U*›

fix *M1 M2* :: ‹('*a*, '*a clause*) *ann-lits*› **and** *K*
assume ‹*Propagated L′* (*clause D*) # *M* = *M2* @ *Decided K* # *M1*›
then have *M*: ‹*M* = *tl M2* @ *Decided K* # *M1*›
by (*meson cdcl*$_W$*-restart-mset.propagated-cons-eq-append-decide-cons*)
then show
‹*twl-lazy-update M1 C*› **and**
‹*watched-literals-false-of-max-level M1 C*› **and**
‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *C past-invs* **by** (*auto simp add*: *past-invs.simps*)
next
fix *M1 M2* :: ‹('*a*, '*a clause*) *ann-lits*› **and** *K*
assume ‹*Propagated L′* (*clause D*) # *M* = *M2* @ *Decided K* # *M1*›
then have *M*: ‹*M* = *tl M2* @ *Decided K* # *M1*›
by (*meson cdcl*$_W$*-restart-mset.propagated-cons-eq-append-decide-cons*)
then show ‹*confl-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
‹*clauses-to-update-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *past-invs* **by** (*auto simp add*: *past-invs.simps*)
qed
next
case (*conflict D L L′ M N U NE UE NS US N0 U0 WS Q*) **note** *twl* = *this*(*9*)
then show *?case*
by (*auto simp*: *past-invs.simps*)
next
case (*delete-from-working L′ D M N U NE UE NS US N0 U0 L WS Q*) **note** *watched* = *this*(*1*) **and** *L′* = *this*(*2*) **and**
twl = *this*(*3*) **and** *valid* = *this*(*4*) **and** *inv* = *this*(*5*) **and** *past-invs* = *this*(*8*)
show *?case* **unfolding** *past-invs.simps Ball-def*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N* + *U*›

fix *M1 M2* :: ‹('*a*, '*a clause*) *ann-lits*› **and** *K*
assume ‹*M* = *M2* @ *Decided K* # *M1*›
then show ‹*twl-lazy-update M1 C*› **and**
‹*watched-literals-false-of-max-level M1 C*› **and**
‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›

```
    using C past-invs by (auto simp add: past-invs.simps)
  next
    fix M1 M2 :: ‹('a, 'a clause) ann-lits› and K
    assume ‹M = M2 @ Decided K # M1›
    then show ‹confl-cands-enqueued (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})› and
      ‹propa-cands-enqueued (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})› and
      ‹clauses-to-update-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
      using past-invs by (auto simp add: past-invs.simps)
  qed
next
  case (update-clause D L L' M K N U N' U' NE UE NS US N0 U0 WS Q) note watched = this(1)
and uL = this(2)
    and L' = this(3) and K = this(4) and undef = this(5) and N'U' = this(6) and twl = this(7)
and
    valid = this(8) and inv = this(9) and twl-excep = this(10) and no-dup = this(11) and
    past-invs = this(12)
  obtain WD UWD where D: ‹D = TWL-Clause WD UWD› by (cases D)
  have L: ‹L ∈# watched D› and D-N-U: ‹D ∈# N + U› and lev-L: ‹get-level M L = count-decided
M›
    using valid by auto
  then have struct-D: ‹struct-wf-twl-cls D›
    using twl by (auto simp: twl-st-inv.simps)
  have L'-UWD: ‹L ∉# remove1-mset L' UWD› if ‹L ∈# WD› for L
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then have ‹count UWD L ≥ 1›
      by (auto simp del: count-greater-zero-iff simp: count-greater-zero-iff[symmetric]
          split: if-splits)
    then have ‹count (clause D) L ≥ 2›
      using D that by (auto simp del: count-greater-zero-iff simp: count-greater-zero-iff[symmetric]
          split: if-splits)
    moreover have ‹distinct-mset (clause D)›
      using struct-D D by (auto simp: distinct-mset-union)
    ultimately show False
      unfolding distinct-mset-count-less-1 by (metis Suc-1 not-less-eq-eq)
  qed
  have L'-L'-UWD: ‹K ∉# remove1-mset K UWD›
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then have ‹count UWD K ≥ 2›
      by (auto simp del: count-greater-zero-iff simp: count-greater-zero-iff[symmetric]
          split: if-splits)
    then have ‹count (clause D) K ≥ 2›
      using D L' by (auto simp del: count-greater-zero-iff simp: count-greater-zero-iff[symmetric]
          split: if-splits)
    moreover have ‹distinct-mset (clause D)›
      using struct-D D by (auto simp: distinct-mset-union)
    ultimately show False
      unfolding distinct-mset-count-less-1 by (metis Suc-1 not-less-eq-eq)
  qed
  have ‹watched-literals-false-of-max-level M D›
    using D-N-U twl by (auto simp: twl-st-inv.simps)
  let ?D = ‹update-clause D L K›
  have ∗: ‹C ∈# N + U› if ‹C ≠ ?D› and C: ‹C ∈# N' + U'› for C
    using C N'U' that by (auto elim!: update-clausesE dest: in-diffD)
  have n-d: ‹no-dup M›
```

```
    using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps)
  then have uK-M: ‹- K ∉ lits-of-l M›
    using undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def
      distinct-consistent-interp by blast
  have add-remove-WD: ‹add-mset K (remove1-mset L WD) ≠ WD›
    using uK-M uL by (auto simp: add-mset-remove-trivial-iff trivial-add-mset-remove-iff)
  have cls-D-D: ‹clause ?D = clause D›
    by (cases D) (use watched K in auto)

  have L-M: ‹L ∉ lits-of-l M›
    using n-d uL by (fastforce dest!: distinct-consistent-interp
       simp: consistent-interp-def lits-of-def uminus-lit-swap)
  have w-max-D: ‹watched-literals-false-of-max-level M D›
    using D-N-U twl by (auto simp: twl-st-inv.simps)

  show ?case unfolding past-invs.simps Ball-def
  proof (intro allI conjI impI)
    fix C
    assume C: ‹C ∈# N' + U'›

    fix M1 M2 :: ‹('a, 'a clause) ann-lits› and K'
    assume M: ‹M = M2 @ Decided K' # M1›

    have lev-L-M1: ‹get-level M1 L = 0›
      using lev-L n-d unfolding M
      apply (auto simp: get-level-append-if get-level-cons-if
          atm-of-notin-get-level-eq-0 split: if-splits dest: defined-lit-no-dupD)
      using atm-of-notin-get-level-eq-0 defined-lit-no-dupD(1) apply blast
      apply (simp add: defined-lit-map)
      by (metis Suc-count-decided-gt-get-level add-Suc-right not-add-less2)

    have ‹twl-lazy-update M1 D›
      using past-invs D-N-U unfolding past-invs.simps M twl-lazy-update.simps C
      by fast
    then have
      lazy-L': ‹- L' ∈ lits-of-l M1 ⟹ ¬ has-blit M1 (add-mset L (add-mset L' UWD)) L' ⟹
          (∀ K∈#UWD. get-level M1 K ≤ get-level M1 L' ∧ - K ∈ lits-of-l M1)›
      using watched unfolding D twl-lazy-update.simps
      by (simp-all add: all-conj-distrib)
    have excep-inv: ‹twl-exception-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#}) C› if
‹C ≠ ?D›
      using ∗ C past-invs that M by (auto simp add: past-invs.simps)
    then have ‹twl-exception-inv (M1, N', U', None, NE, UE, NS, US, N0, U0, {#}, {#}) C› if ‹C
≠ ?D›
      using N'U' that by (auto simp add: twl-st-inv.simps twl-exception-inv.simps)
    moreover have ‹twl-lazy-update M1 C› ‹watched-literals-false-of-max-level M1 C›
      if ‹C ≠ ?D›
      using ∗ C twl past-invs M N'U' that
      by (auto simp add: past-invs.simps twl-exception-inv.simps)
    moreover {
      have ‹twl-lazy-update M1 ?D›
        using D watched uK-M K lazy-L'
        by (auto simp add: M add-mset-eq-add-mset twl-exception-inv.simps lev-L-M1
            all-conj-distrib add-mset-commute dest!: multi-member-split[of K])
    }
```

moreover have ‹*watched-literals-false-of-max-level M1 ?D*›
using *D watched uK-M K lazy-L′*
by (*auto simp add*: *M add-mset-eq-add-mset twl-exception-inv.simps lev-L-M1*
all-conj-distrib add-mset-commute dest!: *multi-member-split*[*of K*])
moreover have ‹*twl-exception-inv* (*M1*, *N′*, *U′*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *?D*›
using *D watched uK-M K lazy-L′*
by (*auto simp add*: *M add-mset-eq-add-mset twl-exception-inv.simps lev-L-M1*
all-conj-distrib add-mset-commute dest!: *multi-member-split*[*of K*])
ultimately show ‹*twl-lazy-update M1 C*› ‹*watched-literals-false-of-max-level M1 C*›
‹*twl-exception-inv* (*M1*, *N′*, *U′*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
by *blast*+
next
have [*dest*!]: ‹*C* ∈# *N′* ⟹ *C* ∈# *N* ∨ *C* = *?D*› ‹*C* ∈# *U′* ⟹ *C* ∈# *U* ∨ *C* = *?D*› **for** *C*
using *N′U′* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
fix *M1 M2* :: ‹('*a*, '*a clause*) *ann-lits*› **and** *K′*
assume *M*: ‹*M* = *M2* @ *Decided K′* # *M1*›
then have ‹*confl-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
w-q: ‹*clauses-to-update-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *past-invs* **by** (*auto simp add*: *past-invs.simps*)
moreover have ‹¬*M1* ⊨*as CNot* (*clause ?D*)›
using *K uK-M* **unfolding** *true-annots-true-cls-def-iff-negation-in-model cls-D-D M*
by (*cases D*) *auto*
moreover {
have *lev-L-M*: ‹*get-level M L* = *count-decided M*› **and** *uL-M*: ‹−*L* ∈ *lits-of-l M*›
using *valid* **by** *auto*
have ‹−*L* ∉ *lits-of-l M1*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*undefined-lit* (*M2* @ [*Decided K′*]) *L*›
using *uL-M n-d* **unfolding** *M*
by (*auto simp*: *lits-of-def uminus-lit-swap no-dup-def defined-lit-map*
dest: *mk-disjoint-insert*)
then show *False*
using *lev-L-M count-decided-ge-get-level*[*of M1 L*]
by (*auto simp*: *lits-of-def uminus-lit-swap M*)
qed
then have ‹¬*M1* ⊨*as CNot* (*remove1-mset K″* (*clause ?D*))› **for** *K″*
using *K uK-M watched D* **unfolding** *M* **by** (*cases* ‹*K″* = *L*›) *auto* **}**
ultimately show ‹*confl-cands-enqueued* (*M1*, *N′*, *U′*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
and
‹*propa-cands-enqueued* (*M1*, *N′*, *U′*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
by (*auto simp add*: *twl-st-inv.simps split*: *if-splits*)
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
then have *NU-remove*: ‹*NU* = *remove1-mset D* (*N* + *U*)›
by *auto*
have ‹*N′* + *U′* = *add-mset ?D* (*remove1-mset D* (*N* + *U*))›
using *N′U′ D-N-U* **by** (*auto elim*!: *update-clausesE*)
then have *N′U′*: ‹*N′*+*U′* = *add-mset ?D NU*›
unfolding *NU-remove* **.**
have *watched-D*: ‹*watched ?D* = {#*K*, *L′*#}›
using *D add-remove-WD watched* **by** *auto*

have ‹*twl-lazy-update M1 D*›
using *past-invs D-N-U* **unfolding** *past-invs.simps M twl-lazy-update.simps*

```
    by fast
  then have
    lazy-L': ‹− L' ∈ lits-of-l M1 ⟹ ¬ has-blit M1 (add-mset L (add-mset L' UWD)) L' ⟹
        (∀K∈#UWD. get-level M1 K ≤ get-level M1 L' ∧ − K ∈ lits-of-l M1)›
    using watched unfolding D twl-lazy-update.simps
    by (simp-all add: all-conj-distrib)
  have uL'-M1: ‹has-blit M1 (clause (update-clause D L K)) L'› if ‹− L' ∈ lits-of-l M1›
  proof −
    show ?thesis
      using K uK-M lazy-L' that D watched unfolding cls-D-D
      by (force simp: M dest!: multi-member-split[of K UWD])
  qed
  show ‹clauses-to-update-inv (M1, N', U', None, NE, UE, NS, US, N0, U0, {#}, {#})›
  proof (induction rule: clauses-to-update-inv-cases)
    case (WS-nempty L C)
    then show ?case by simp
  next
    case (WS-empty K'')
    have uK-M1: ‹− K ∉ lits-of-l M1›
      using uK-M unfolding M by auto
    have ‹¬clauses-to-update-prop {#} M1 (K'', ?D)›
      using uK-M1 uL'-M1 by (auto simp: clauses-to-update-prop.simps watched-D
          add-mset-eq-add-mset)
    then show ?case
      using w-q unfolding clauses-to-update-inv.simps N'U' NU
      by (auto split: if-splits simp: all-conj-distrib watched-D add-mset-eq-add-mset)
  next
    case (Q J C)
    moreover have ‹− K ∉ lits-of-l M1›
      using uK-M unfolding M by auto
    moreover have ‹clauses-to-update-prop {#} M1 (L', D)› if ‹− L' ∈ lits-of-l M1›
      using watched that uL'-M1 Q.hyps calculation(1,2,3,6) cls-D-D
        insert-DiffM w-q watched-D by auto
    ultimately show ?case
      using w-q watched-D unfolding clauses-to-update-inv.simps N'U' NU
      by (fastforce split: if-splits simp: all-conj-distrib add-mset-eq-add-mset)
  qed
 qed
qed
```

2.1.3 Invariants and the Transition System

Conflict and propagate

```
fun literals-to-update-measure :: ‹'v twl-st ⇒ nat list› where
 ‹literals-to-update-measure S = [size (literals-to-update S), size (clauses-to-update S)]›

lemma twl-cp-propagate-or-conflict:
 assumes
   cdcl: ‹cdcl-twl-cp S T› and
   twl: ‹twl-st-inv S› and
   valid: ‹valid-enqueued S› and
   inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)›
 shows
   ‹cdcl-propagate (pstate_W-of S) (pstate_W-of T) ∨
   cdcl-conflict (pstate_W-of S) (pstate_W-of T) ∨
```

```
    (pstate_W-of S = pstate_W-of T ∧ (literals-to-update-measure T, literals-to-update-measure S) ∈
      lexn less-than 2)›
  using cdcl twl valid inv
proof (induction rule: cdcl-twl-cp.induct)
  case (pop M N U N0 U0 L Q)
  then show ?case by (simp add: lexn2-conv)
next
  case (propagate D L L' M N U NE UE NS US N0 U0 WS Q) note watched = this(1) and undef =
this(2) and
    no-upd = this(3) and twl = this(4) and valid = this(5) and inv = this(6)
  let ?S = ‹pstate_W-of (M, N, U, None, NE, UE, NS, US, N0, U0, add-mset (L, D) WS, Q)›
  let ?T = ‹pstate_W-of (Propagated L' (clause D) # M, N, U, None, NE, UE, NS, US, N0, U0, WS,
    add-mset (- L') Q)›
  have ‹∀ s∈#clause `# U. ¬ tautology s›
    using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
    cdcl_W-restart-mset.distinct-cdcl_W-state-def by (simp-all add: cdcl_W-restart-mset-state)
  have D-N-U: ‹D ∈# N + U›
    using valid by auto
  have ‹cdcl-propagate ?S ?T›
    unfolding pstate_W-of.simps
    apply (rule cdcl-propagate.intros[of L' ‹clause D›])
     using watched apply (cases D, simp add: clauses-def; fail)
    using no-upd watched valid apply (cases D;
        simp add: trail.simps true-annots-true-cls-def-iff-negation-in-model; fail)
    using undef apply (simp add: trail.simps)
    apply (use ‹D ∈# N+U› in ‹auto simp add: clauses-def state_W-of-def›; fail)
    done
  then show ?case by blast
next
  case (conflict D L L' M N U NE UE NS US N0 U0 WS Q) note watched = this(1) and defined =
this(2)
    and no-upd = this(3) and twl = this(3) and valid = this(5) and inv = this(6)
  let ?S = ‹pstate_W-of (M, N, U, None, NE, UE, NS, US, N0, U0, add-mset (L, D) WS, Q)›
  let ?T = ‹pstate_W-of (M, N, U, Some (clause D), NE, UE, NS, US, N0, U0, {#}, {#})›
  have D-N-U: ‹D ∈# N + U›
    using valid by auto
  have ‹distinct-mset (clause D)›
    using inv valid ‹D ∈# N + U› unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
    cdcl_W-restart-mset.distinct-cdcl_W-state-def distinct-mset-set-def
    by (auto simp: cdcl_W-restart-mset-state)
  then have ‹L ≠ L'›
    using watched by (cases D) simp
  have ‹M ⊨as CNot (unwatched D)›
    using no-upd by (auto simp: true-annots-true-cls-def-iff-negation-in-model)
  have ‹cdcl-conflict ?S ?T›
    unfolding pstate_W-of.simps
    apply (rule cdcl-conflict.intros[of - ‹clause D›])
    using watched defined valid ‹M ⊨as CNot (unwatched D)›
    apply (cases D; auto simp add: clauses-def
        trail.simps twl-st-inv.simps; fail)
    apply (use ‹D ∈# N+U› in ‹auto simp add: clauses-def state_W-of-def›; fail)
    done
  then show ?case by fast
next
  case (delete-from-working D L L' M N U NE UE NS US N0 U0 WS Q)
  then show ?case by (simp add: lexn2-conv)
```

next
case (*update-clause D L L′ M K N U N′ U′ NE UE NS US N0 U0 WS Q*) **note** *unwatched* = *this*(*4*)
and
valid = *this*(*8*)
have ‹*D* ∈# *N* + *U*›
using *valid* **by** *auto*
have [*simp*]: ‹*clause* (*update-clause D L K*) = *clause D*›
using *valid unwatched* **by** (*cases D*) (*auto simp*: *diff-union-swap2*[*symmetric*]
simp del: *diff-union-swap2*)
have ‹*pstate*$_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*) =
pstate$_W$*-of* (*M*, *N′*, *U′*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹(*literals-to-update-measure* (*M*, *N′*, *U′*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*),
literals-to-update-measure (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*))
∈ *lexn less-than 2*›
using *update-clause* ‹*D* ∈# *N* + *U*› **by** (*cases* ‹*D* ∈# *N*›)
(*fastforce simp*: *image-mset-remove1-mset-if elim*!: *update-clausesE*
simp add: *lexn2-conv*)+
then show *?case* **by** *fast*
qed

lemma *cdcl-twl-o-cdcl*$_W$*-o*:
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of S*)›
shows ‹*pcdcl-tcore* (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)›
using *cdcl twl valid inv*
proof (*induction rule*: *cdcl-twl-o.induct*)
case (*decide M L N NE NS N0 U UE US U0*) **note** *undef* = *this*(*1*) **and** *atm* = *this*(*2*)
have ‹*cdcl-decide* (*pstate*$_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))
(*pstate*$_W$*-of* (*Decided L* # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#−*L*#}))›
unfolding *pstate*$_W$*-of.simps*
apply (*rule cdcl-decide.intros*)
using *undef* **apply** (*simp add*: *trail.simps*; *fail*)
using *atm* **apply** (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
done
then show *?case*
by (*blast dest*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-o.intros pcdcl-core.intros pcdcl-tcore.intros*)
next
case (*skip L D C′ M N U NE UE N0 U0 NS US*) **note** *LD* = *this*(*1*) **and** *D* = *this*(*2*)
show *?case*
apply (*rule pcdcl-tcore.intros*(*1*), *rule pcdcl-core.intros*(*4*))
unfolding *pstate*$_W$*-of.simps*
apply (*rule cdcl-skip.intros*)
using *LD* **apply** (*simp*; *fail*)
using *D* **apply** (*simp*; *fail*)
done
next
case (*resolve L D C M N U NE UE NS US N0 U0*) **note** *LD* = *this*(*1*) **and** *lev* = *this*(*2*) **and** *inv*
= *this*(*5*)
have ‹∀ *La mark a b*. *a* @ *Propagated La mark* # *b* = *Propagated L C* # *M* ⟶
b ⊨*as CNot* (*remove1-mset La mark*) ∧ *La* ∈# *mark*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
by (*auto simp*: *trail.simps*)

then have *LC*: ‹*L* ∈# *C*›
by *blast*
show *?case*
apply (*rule pcdcl-tcore.intros*(*1*), *rule pcdcl-core.intros*(*5*))
unfolding $pstate_W$*-of.simps*
apply (*rule cdcl-resolve.intros*)
using *LD* **apply** (*simp*; *fail*)
using *lev* **apply** (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
using *LC* **apply** (*simp add*: *trail.simps*; *fail*)
done
next
case (*backtrack-unit-clause L D K M1 M2 M D′ i N U NE UE NS US N0 U0*) **note** *L-D* = *this*(*1*)
and
decomp = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *max-D′-L* = *this*(*4*) **and** *lev-D* = *this*(*5*) **and**
lev-K = *this*(*6*) **and** *D′-D* = *this*(*8*) **and** *NU-D′* = *this*(*9*) **and** *inv* = *this*(*12*) **and**
D′[*simp*] = *this*(*7*)
have *D*: ‹*D* = *add-mset L* (*remove1-mset L D*)›
using *L-D* **by** *auto*
show *?case*
apply (*rule pcdcl-tcore.intros*(*4*))
unfolding $pstate_W$*-of.simps*
apply (*subst D*)
apply (*rule cdcl-backtrack-unit.intros*)
using *backtrack-unit-clause* **by** *auto*
next
case (*backtrack-nonunit-clause L D K M1 M2 M D′ i N U NE UE NS US N0 U0 L′*) **note** *LD* = *this*(*1*) **and**
decomp = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *max-lev* = *this*(*4*) **and** *i* = *this*(*5*) **and** *lev-K* = *this*(*6*)
and *D′-D* = *this*(*8*) **and** *NU-D′* = *this*(*9*) **and** *L-D′* = *this*(*10*) **and** *L′* = *this*(*11−12*) **and**
inv = *this*(*15*)
let *?S* = ‹$state_W$*-of* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?T* = ‹$state_W$*-of* (*Propagated L D* # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#*L*#})›
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*)
have ‹*undefined-lit M1 L*›
apply (*rule* $cdcl_W$*-restart-mset.backtrack-lit-skiped*[*of ?S - K - M2 i*])
subgoal
using *lev-L inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
subgoal using *decomp* **by** (*simp add*: *trail.simps*; *fail*)
subgoal using *lev-L inv*
unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
subgoal using *lev-K* **by** (*simp add*: *trail.simps*; *fail*)
done
obtain *M3* **where** *M3*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** (*blast dest*!: *get-all-ann-decomposition-exists-prepend*)

have ‹*undefined-lit* (*M3* @ *M2*) *K*›
using *n-d* **unfolding** *M3* **by** (*auto simp*: *lits-of-def*)
then have *count-M1*: ‹*count-decided M1* = *i*›
using *lev-K* **unfolding** *M3* **by** (*auto simp*: *image-Un*)

have ‹$L \neq L'$›
using L' *lev-L lev-K count-decided-ge-get-level*[*of M K*] L' **by** *auto*
then have *D*: ‹*add-mset L* (*add-mset L'* (D' − {#*L*, *L'*#})) = D'›
using L' *L-D'*
by (*metis add-mset-diff-bothsides diff-single-eq-union insert-noteq-member mset-add*)
then have *D4*: ‹({#*L*, *L'*#} + (D' − {#*L*, *L'*#})) = *add-mset L* (*add-mset L'* (D' − {#*L*, *L'*#}))›
by *auto*
have D': ‹*remove1-mset L D'* = *add-mset L'* (D' − {#*L*, *L'*#})›
by (*subst D*[*symmetric*]) *auto*
have D'': ‹*D* = *add-mset L* (*remove1-mset L D*)›
using *L-D' D'-D* **by** *auto*
show *?case*
apply (*subst D*[*symmetric*])
apply (*subst* D'')
apply (*rule pcdcl-tcore.intros*(*1*), *rule pcdcl-core.intros*(*6*))
unfolding $pstate_W$*-of.simps image-mset-add-mset clause.simps D4*
apply (*rule cdcl-backtrack.intros*[*of K M1 M2 - - - i*])
subgoal using *decomp* **by** (*simp add*: *trail.simps*)
subgoal using *lev-L* **by** (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
subgoal using *max-lev L-D'* **by** (*simp add*: $cdcl_W$*-restart-mset-state get-maximum-level-add-mset D*)
subgoal using *i* **by** (*simp add*: $cdcl_W$*-restart-mset-state* D')
subgoal using *lev-K i* **unfolding** D' **by** (*simp add*: *trail.simps*)
subgoal using *D'-D* **by** (*metis* D' *mset-le-subtract*)
subgoal using *NU-D' L-D'* **by** (*simp add*: *mset-le-subtract clauses-def ac-simps D*)
done
qed

lemma $cdcl$*-twl-cp-*$cdcl_W$*-stgy*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹
pcdcl-tcore-stgy ($pstate_W$*-of S*) ($pstate_W$*-of T*) ∨
(*state*$_W$*-of S* = *state*$_W$*-of T* ∧ *state*$_W$*-of S* = *state*$_W$*-of T* ∧ (*literals-to-update-measure T*, *literals-to-update-measure S*)
∈ *lexn less-than 2*)›
by (*auto dest*!: *twl-cp-propagate-or-conflict*
$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.conflict'*
$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.propagate'*
intro: *pcdcl-core.intros pcdcl-core-stgy.intros pcdcl-tcore-stgy.intros*
simp: *twl-struct-invs-def pcdcl-all-struct-invs-def*)

lemma *cdcl-twl-cp-conflict*:
‹*cdcl-twl-cp S T* ⟹ *get-conflict T* ≠ *None* ⟶
clauses-to-update T = {#} ∧ *literals-to-update T* = {#}›
by (*induction rule*: *cdcl-twl-cp.induct*) *auto*

lemma *cdcl-twl-cp-twl-struct-invs*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹ *twl-struct-invs T*›
supply [*simp*] = *pcdcl-all-struct-invs-def*
apply (*subst twl-struct-invs-def*)
apply (*intro conjI*)
subgoal by (*rule twl-cp-twl-inv*; *auto simp add*: *twl-struct-invs-def twl-cp-twl-inv*)
subgoal by (*simp add*: *twl-cp-valid twl-struct-invs-def*)
subgoal by (*metis pcdcl-all-struct-invs-def pcdcl-core.intros*(*1*) *pcdcl-core.intros*(*2*)
pcdcl-tcore.simps pcdcl-tcore-pcdcl-all-struct-invs $state_W$*-of-def twl-cp-propagate-or-conflict*
twl-struct-invs-def)
subgoal

by (*metis pcdcl-all-struct-invs-def pcdcl-core-stgy.intros*(*1*) *pcdcl-core-stgy.intros*(*2*)
pcdcl-core-stgy-no-smaller-propa state$_W$*-of-def twl-cp-propagate-or-conflict twl-struct-invs-def*)
subgoal by (*rule twl-cp-twl-st-exception-inv*; *auto simp add*: *twl-struct-invs-def*; *fail*)
subgoal by (*use twl-struct-invs-def twl-cp-no-duplicate-queued* **in** *blast*)
subgoal by (*rule twl-cp-distinct-queued*; *auto simp add*: *twl-struct-invs-def*)
subgoal by (*rule twl-cp-confl-cands-enqueued*; *auto simp add*: *twl-struct-invs-def*; *fail*)
subgoal by (*rule twl-cp-propa-cands-enqueued*; *auto simp add*: *twl-struct-invs-def*; *fail*)
subgoal by (*simp add*: *cdcl-twl-cp-conflict*; *fail*)
subgoal by (*simp add*: *twl-struct-invs-def twl-cp-clauses-to-update*; *fail*)
subgoal by (*simp add*: *twl-cp-past-invs twl-struct-invs-def*; *fail*)
done

lemma (**in** *conflict-driven-clause-learning*$_W$) *cdcl*$_W$*-restart-conflict-non-zero-unless-level-0*:
assumes
‹*cdcl*$_W$*-restart S T*›
‹*cdcl*$_W$*-stgy-invariant S*› **and**
‹*conflict-non-zero-unless-level-0 S*›
shows ‹*conflict-non-zero-unless-level-0 T*›
proof −
have [*dest*]: ‹*local.backtrack-lvl S* = *0* ⟹ *count-decided* (*tl* (*trail S*)) = *0*›
by (*cases* ‹*trail S*›)
(*auto split*: *if-splits simp del*: *state-simp*)
have ‹{#} ∈# *clauses S* ⟹ *count-decided* (*trail S*) = *0*›
using *assms*(*2*)
by (*fastforce simp*: *cdcl*$_W$*-stgy-invariant-def no-smaller-confl-def count-decided-0-iff is-decided-def*
dest!: *multi-member-split split-list*)
with *assms* **show** *?thesis*
by (*induction rule*: *cdcl*$_W$*-restart-all-induct*)
(*auto simp add*: *conflict-non-zero-unless-level-0-def state-prop*
no-false-clause-def propagate.simps remove1-mset-empty-iff
conflict.simps cdcl$_W$*-o.simps*
decide.simps resolve.simps
backtrack.simps cdcl$_W$*-bj.simps*
skip.simps)
qed

lemma *cdcl-twl-cp-twl-stgy-invs*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹ *twl-stgy-invs S* ⟹ *twl-stgy-invs T*›
using *pcdcl-stgy-stgy-invariant*[*of* ‹*pstate*$_W$*-of S*› ‹*pstate*$_W$*-of S*›]
unfolding *twl-stgy-invs-def*
apply (*intro conjI impI*)
apply (*metis cdcl-twl-cp-cdcl*$_W$*-stgy pcdcl-tcore-stgy-pcdcl-stgy*′ *rtranclp-pcdcl-stgy-stgy-invariant state*$_W$*-of-def twl-struct-invs-def*)
by (*metis cdcl*$_W$*-restart-mset.cdcl*$_W$*-restart-conflict-non-zero-unless-level-0*
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-cdcl*$_W$*-restart pcdcl-all-struct-invs-def pcdcl-core-stgy.simps*
pcdcl-core-stgy-is-cdcl-stgy state$_W$*-of-def twl-cp-propagate-or-conflict twl-struct-invs-def*)

The other rules

lemma
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-struct-invs S*›
shows

```
    cdcl-twl-o-twl-st-inv: ‹twl-st-inv T› and
    cdcl-twl-o-past-invs: ‹past-invs T›
  using cdcl twl
proof (induction rule: cdcl-twl-o.induct)
  case (decide M K N NE NS N0 U UE US U0) note undef = this(1) and atm = this(2)

  case 1 note invs = this(1)
  let ?S = ‹(M, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
  have inv: ‹twl-st-inv ?S› and excep: ‹twl-st-exception-inv ?S› and past: ‹past-invs ?S› and
    w-q: ‹clauses-to-update-inv ?S›
    using invs unfolding twl-struct-invs-def by blast+
  have n-d: ‹no-dup M›
    using invs unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def pcdcl-all-struct-invs-def
   by (simp add: cdcl_W-restart-mset-state state_W-of-def)
  have n-d': ‹no-dup (Decided K # M)›
    using defined-lit-map n-d undef by auto
  have propa-cands: ‹propa-cands-enqueued ?S› and
    confl-cands: ‹confl-cands-enqueued ?S›
    using invs unfolding twl-struct-invs-def by blast+

  show ?case
    unfolding twl-st-inv.simps Ball-def
  proof (intro conjI allI impI)
    fix C :: ‹'a twl-cls›
    assume C: ‹C ∈# N + U›
    show struct: ‹struct-wf-twl-cls C›
      using inv C by (auto simp: twl-st-inv.simps)

    have watched: ‹watched-literals-false-of-max-level M C› and
      lazy: ‹twl-lazy-update M C›
      using C inv by (auto simp: twl-st-inv.simps)

    obtain W UW where C-W: ‹C = TWL-Clause W UW›
      by (cases C)

    have H: False if
      W: ‹L ∈# W› and
      uL: ‹- L ∈ lits-of-l (Decided K # M)› and
      L': ‹¬has-blit (Decided K # M) (W + UW) L› and
      False: ‹-L ≠ K› for L
    proof -
      have H: ‹- L ∈ lits-of-l M ⟹ ¬ has-blit M (W + UW) L ⟹ get-level M L = count-decided M
›
        using watched W unfolding C-W
        by auto
      obtain L' where W': ‹W = {#L, L'#}›
        using struct W size-2-iff[of W] unfolding C-W
        by (auto simp: add-mset-eq-single add-mset-eq-add-mset dest!: multi-member-split)
      have no-has-blit: ‹¬has-blit M (W + UW) L›
        using no-has-blit-decide'[of K M C] L' n-d C-W W undef by auto
      then have ‹∀K ∈# UW. -K ∈ lits-of-l M›
        using uL L' False excep C W C-W L' W n-d undef
        by (auto simp: twl-exception-inv.simps all-conj-distrib
            dest!: multi-member-split[of - N])
      then have M-CNot-C: ‹M ⊨as CNot (remove1-mset L' (clause C))›
```

```
    using uL False W' unfolding true-annots-true-cls-def-iff-negation-in-model
    by (auto simp: C-W W)
  moreover have L'-C: ‹L' ∈# clause C›
    unfolding C-W W' by auto
  ultimately have ‹defined-lit M L'›
    using propa-cands C by auto

  then have ‹-L' ∈ lits-of-l M›
    using L' W' False uL C-W L'-C H no-has-blit
    apply (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
    by (metis C-W L'-C no-has-blit clause.simps
        count-decided-ge-get-level has-blit-def is-blit-def)
  then have ‹M ⊨as CNot (clause C)›
    using M-CNot-C W' unfolding true-annots-true-cls-def-iff-negation-in-model
    by (auto simp: C-W)
  then show False
    using confl-cands C by auto
qed

show ‹watched-literals-false-of-max-level (Decided K # M) C›
  unfolding C-W watched-literals-false-of-max-level.simps
proof (intro allI impI)
  fix L
  assume
    W: ‹L ∈# W› and
    uL: ‹- L ∈ lits-of-l (Decided K # M)› and
    L': ‹¬has-blit (Decided K # M) (W + UW) L›
  then have ‹-L = K›
    using H[OF W uL L'] by fast
  then show ‹get-level (Decided K # M) L = count-decided (Decided K # M)›
    by auto
qed

{
  assume exception: ‹¬ twl-is-an-exception C {#-K#} {#}›
  have ‹twl-lazy-update M C›
    using C inv by (auto simp: twl-st-inv.simps)
  have lev-le-Suc: ‹get-level M Ka ≤ Suc (count-decided M)› for Ka
    using count-decided-ge-get-level le-Suc-eq by blast
  show ‹twl-lazy-update (Decided K # M) C›
    unfolding C-W twl-lazy-update.simps Ball-def
  proof (intro allI impI)
    fix L K' :: ‹'a literal›
    assume
      W: ‹L ∈# W› and
      uL: ‹- L ∈ lits-of-l (Decided K # M)› and
      L': ‹¬has-blit (Decided K # M) (W + UW) L› and
      K': ‹K' ∈# UW›
    then have ‹-L = K›
      using H[OF W uL L'] by fast
    then have False
      using exception W
      by (auto simp: C-W twl-is-an-exception-def)
    then show ‹get-level (Decided K # M) K' ≤ get-level (Decided K # M) L ∧
        -K' ∈ lits-of-l (Decided K # M)›
      by fast
```

```
    qed
  }
 qed

 case 2
 show ?case
  unfolding past-invs.simps Ball-def
 proof (intro allI impI conjI)
  fix M1 M2 K' C
  assume ‹Decided K # M = M2 @ Decided K' # M1› and C: ‹C ∈# N + U›
  then have M: ‹M = tl M2 @ Decided K' # M1 ∨ M = M1›
   by (cases M2) auto
  have IH: ‹∀ M1 M2 K. M = M2 @ Decided K # M1 ⟶
     twl-lazy-update M1 C ∧ watched-literals-false-of-max-level M1 C ∧
     twl-exception-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#}) C›
   using past C unfolding past-invs.simps by blast

  have ‹twl-lazy-update M C›
   using inv C unfolding twl-st-inv.simps by auto
  then show ‹twl-lazy-update M1 C›
   using IH M by blast

  have ‹watched-literals-false-of-max-level M C›
   using inv C unfolding twl-st-inv.simps by auto
  then show ‹watched-literals-false-of-max-level M1 C›
   using IH M by blast

  have ‹twl-exception-inv (M, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#}) C›
   using excep inv C unfolding twl-st-inv.simps by auto
  then show ‹twl-exception-inv (M1, N, U, None, NE, UE, NS, US,N0, U0, {#}, {#}) C›
   using IH M by blast
 next
  fix M1 M2 :: ‹('a, 'a clause) ann-lits› and K'
  assume ‹Decided K # M = M2 @ Decided K' # M1›
  then have M: ‹M = tl M2 @ Decided K' # M1 ∨ M = M1›
   by (cases M2) auto
  then show ‹confl-cands-enqueued (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})› and
   ‹propa-cands-enqueued (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})› and
   ‹clauses-to-update-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
   using confl-cands past propa-cands w-q unfolding past-invs.simps by blast+
 qed

next
 case (skip L D C' M N U NE UE NS US)
 case 1
 then show ?case
  by (auto simp: twl-st-inv.simps twl-struct-invs-def)
 case 2
 then show ?case
  by (auto simp: past-invs.simps twl-struct-invs-def)
next
 case (resolve L D C M N U NE UE NS US)
 case 1
 then show ?case
  by (auto simp: twl-st-inv.simps twl-struct-invs-def)
 case 2
```

then show *?case*
by (*auto simp*: *past-invs.simps twl-struct-invs-def*)
next
case (*backtrack-unit-clause K′ D K M1 M2 M D′ i N U NE UE NS US N0 U0*) **note** *decomp* = *this*(*2*)
and
lev = *this*(*3*−*5*)

case *1* **note** *invs* = *this*(*1*)
let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?T* = ‹(*Propagated K′* {#*K′*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#− *K′*#})›
let *?M1* = ‹*Propagated K′* {#*K′*#} # *M1*›
have *bt-twl*: ‹*cdcl-twl-o ?S ?T*›
using *cdcl-twl-o.backtrack-unit-clause*[*OF backtrack-unit-clause.hyps*] .
then have ‹*pcdcl-tcore* (*pstate*$_W$ *-of ?S*) (*pstate*$_W$ *-of ?T*)›
by (*rule cdcl-twl-o-cdcl*$_W$*-o*) (*use invs* **in** ‹*simp-all add*: *twl-struct-invs-def pcdcl-all-struct-invs-def*›)
from *pcdcl-tcore-pcdcl-all-struct-invs*[*OF this*]
have *struct-inv-T*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$ *-of ?T*)›
using *invs* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
have *inv*: ‹*twl-st-inv ?S*› **and** *w-q*: ‹*clauses-to-update-inv ?S*› **and** *past*: ‹*past-invs ?S*›
using *invs* **unfolding** *twl-struct-invs-def* **by** *blast*+
have *n-d*: ‹*no-dup M*›
using *invs* **unfolding** *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*simp add*: *cdcl*$_W$*-restart-mset-state*)
have *n-d′*: ‹*no-dup ?M1*›
using *struct-inv-T* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps*)

have *propa-cands*: ‹*propa-cands-enqueued ?S*› **and**
confl-cands: ‹*confl-cands-enqueued ?S*›
using *invs* **unfolding** *twl-struct-invs-def* **by** *blast*+

have *excep*: ‹*twl-st-exception-inv ?S*›
using *invs* **unfolding** *twl-struct-invs-def* **by** *fast*

obtain *M3* **where** *M*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** *blast*
define *M2′* **where** ‹*M2′* = *M3* @ *M2*›
have *M′*: ‹*M* = *M2′* @ *Decided K* # *M1*›
unfolding *M M2′-def* **by** *simp*

have *propa-cands-M1*:
‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#− *K′*#})›
unfolding *propa-cands-enqueued.simps*
proof (*intro allI impI*)
fix *L C*
assume
C: ‹*C* ∈# *N* + *U*› **and**
L: ‹*L* ∈# *clause C*› **and**
M1-CNot: ‹*M1* ⊨*as CNot* (*remove1-mset L* (*clause C*))› **and**
undef: ‹*undefined-lit M1 L*›
define *D* **where** ‹*D* = *remove1-mset L* (*clause C*)›
have ‹*add-mset L D* ∈# *clause* ‘# (*N* + *U*)› **and** ‹*M1* ⊨*as CNot D*›
using *C L M1-CNot* **unfolding** *D-def* **by** *auto*

```
  moreover have ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of ?S)›
    using invs unfolding twl-struct-invs-def by blast
  ultimately have False
    using undef M'
    by (auto 7 7 simp: cdcl_W-restart-mset.no-smaller-propa-def trail.simps clauses-def)
  then show ‹(∃ L'. L' ∈# watched C ∧ L' ∈# {#- K'#}) ∨ (∃ L. (L, C) ∈# {#})›
    by fast
qed

have excep-M1: ‹twl-st-exception-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
  using past unfolding past-invs.simps M' by auto

show ?case
  unfolding twl-st-inv.simps Ball-def
proof (intro conjI allI impI)
  fix C :: ‹'a twl-cls›
  assume C: ‹C ∈# N + U›
  show struct: ‹struct-wf-twl-cls C›
    using inv C by (auto simp: twl-st-inv.simps)

  obtain CW CUW where C-W: ‹C = TWL-Clause CW CUW›
      by (cases C)

  {
    assume exception: ‹¬ twl-is-an-exception C {#-K'#} {#}›
    have
      lazy: ‹twl-lazy-update M1 C› and
      watched-max: ‹watched-literals-false-of-max-level M1 C›
      using C past M by (auto simp: past-invs.simps)
    have lev-le-Suc: ‹get-level M Ka ≤ Suc (count-decided M)› for Ka
      using count-decided-ge-get-level le-Suc-eq by blast
    have Lev-M1: ‹get-level (?M1) K ≤ count-decided M1› for K
      by (auto simp: count-decided-ge-get-level get-level-cons-if)

    show ‹twl-lazy-update ?M1 C›
    proof -
      show ?thesis
        using  Lev-M1
        using twl C exception twl n-d'  watched-max
        unfolding C-W
        apply (auto simp:  count-decided-ge-get-level
            twl-is-an-exception-add-mset-to-queue atm-of-eq-atm-of
            dest!: no-has-blit-propagate' no-has-blit-propagate)
          apply (metis count-decided-ge-get-level get-level-skip-beginning get-level-take-beginning)
        using lazy unfolding C-W twl-lazy-update.simps apply blast
         apply (metis count-decided-ge-get-level get-level-skip-beginning get-level-take-beginning)
        using lazy unfolding C-W twl-lazy-update.simps apply blast
        done
    qed

  }

  have ‹watched-literals-false-of-max-level M1 C›
    using past C unfolding M' past-invs.simps by blast
  then show ‹watched-literals-false-of-max-level ?M1 C›
    using has-blit-Cons n-d'
```

by (*auto simp*: *C-W get-level-cons-if*)
qed
case *2*
show *?case*
unfolding *past-invs.simps Ball-def*
proof (*intro allI impI conjI*)
fix *M1'' M2'' K'' C*
assume ‹*?M1 = M2'' @ Decided K'' # M1''*› **and** *C*: ‹*C ∈# N + U*›
then have *M1*: ‹*M1 = tl M2'' @ Decided K'' # M1''*›
by (*cases M2''*) *auto*
have ‹*twl-lazy-update M1'' C*›‹*watched-literals-false-of-max-level M1'' C*›
using *past C* **unfolding** *past-invs.simps M M1 twl-exception-inv.simps* **by** *auto*
moreover {
have ‹*twl-exception-inv* (*M1''*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *past C* **unfolding** *past-invs.simps M M1* **by** *auto*
then have ‹*twl-exception-inv* (*M1''*, *N*, *U*, *None*, *NE*, *add-mset* {#*K'*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *C* **unfolding** *twl-exception-inv.simps* **by** *auto* **}**
ultimately show ‹*twl-lazy-update M1'' C*›‹*watched-literals-false-of-max-level M1'' C*›
‹*twl-exception-inv* (*M1''*, *N*, *U*, *None*, *NE*, *add-mset* {#*K'*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
by *fast+*
next
fix *M1'' M2'' K''*
assume ‹*?M1 = M2'' @ Decided K'' # M1''*›
then have *M1*: ‹*M1 = tl M2'' @ Decided K'' # M1''*›
by (*cases M2''*) *auto*
then show
‹*confl-cands-enqueued* (*M1''*, *N*, *U*, *None*, *NE*, *add-mset* {#*K'*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
‹*propa-cands-enqueued* (*M1''*, *N*, *U*, *None*, *NE*, *add-mset* {#*K'*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
‹*clauses-to-update-inv* (*M1''*, *N*, *U*, *None*, *NE*, *add-mset* {#*K'*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *past* **by** (*auto simp add*: *past-invs.simps M*)
qed
next
case (*backtrack-nonunit-clause K' D K M1 M2 M D' i N U NE UE NS US N0 U0 K''*) **note** *K'-D = this*(*1*) **and**
decomp = this(*2*) **and** *lev-K' = this*(*3*) **and** *i = this*(*5*) **and** *lev-K = this*(*6*) **and** *K'-D' = this*(*10*)
and *K'' = this*(*11*) **and** *lev-K'' = this*(*12*)
case *1* **note** *invs = this*(*1*)
let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?M1* = ‹*Propagated K' D' # M1*›
let *?T* = ‹(*?M1*, *N*, *add-mset* (*TWL-Clause* {#*K'*, *K''*#} (*D'* − {#*K'*, *K''*#})) *U*, *None*, *NE*, *UE*, *NS*, *US*,
N0, *U0*, {#}, {#− *K'*#})›
let *?D* = ‹*TWL-Clause* {#*K'*, *K''*#} (*D'* − {#*K'*, *K''*#})›
have *bt-twl*: ‹*cdcl-twl-o ?S ?T*›
using *cdcl-twl-o.backtrack-nonunit-clause*[*OF backtrack-nonunit-clause.hyps*] **.**
then have ‹*pcdcl-tcore* (*pstate*$_W$ *-of ?S*) (*pstate*$_W$ *-of ?T*)›
by (*rule cdcl-twl-o-cdcl*$_W$*-o*) (*use invs* **in** ‹*simp-all add*: *twl-struct-invs-def pcdcl-all-struct-invs-def*›)
from *pcdcl-tcore-pcdcl-all-struct-invs*[*OF this*]
have *struct-inv-T*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$ *-of ?T*)›
using *invs* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*

have *inv*: ‹*twl-st-inv ?S*› **and**
w-q: ‹*clauses-to-update-inv ?S*› **and**
past: ‹*past-invs ?S*›
using *invs* **unfolding** *twl-struct-invs-def* **by** *blast+*
have *n-d*: ‹*no-dup M*›
using *invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*)
have *n-d'*: ‹*no-dup ?M1*›
using *struct-inv-T* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps*)

have *propa-cands*: ‹*propa-cands-enqueued ?S*› **and**
confl-cands: ‹*confl-cands-enqueued ?S*›
using *invs* **unfolding** *twl-struct-invs-def* **by** *blast+*
obtain *M3* **where** *M*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** *blast*
define *M2'* **where** ‹*M2'* = *M3* @ *M2*›
have *M'*: ‹*M* = *M2'* @ *Decided K* # *M1*›
unfolding *M M2'-def* **by** *simp*
have *struct-inv-S*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?S*)›
using *invs* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
then have ‹*distinct-mset D*›
unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def*
by (*auto simp*: *conflicting.simps*)

have ‹*undefined-lit* (*M3* @ *M2*) *K*›
using *n-d* **unfolding** *M* **by** *auto*
then have *count-M1*: ‹*count-decided M1* = *i*›
using *lev-K* **unfolding** *M* **by** (*auto simp*: *image-Un*)
then have *K''-ne-K*: ‹*K'* ≠ *K''*›
using *lev-K lev-K' lev-K'' count-decided-ge-get-level*[*of M K''*] **unfolding** *M* **by** *auto*
then have *D*:
‹*add-mset K'* (*add-mset K''* (*D'* − {#*K'*, *K''*#})) = *D'*›
‹*add-mset K''* (*add-mset K'* (*D'* − {#*K'*, *K''*#})) = *D'*›
using *K'' K'-D' multi-member-split* **by** *fastforce+*
have *propa-cands-M1*: ‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#},
{#− *K''*#})›
unfolding *propa-cands-enqueued.simps*
proof (*intro allI impI*)
fix *L C*
assume
C: ‹*C* ∈# *N* + *U*› **and**
L: ‹*L* ∈# *clause C*› **and**
M1-CNot: ‹*M1* ⊨*as CNot* (*remove1-mset L* (*clause C*))› **and**
undef: ‹*undefined-lit M1 L*›
define *D* **where** ‹*D* = *remove1-mset L* (*clause C*)›
have ‹*add-mset L D* ∈# *clause* ‘# (*N* + *U*)› **and** ‹*M1* ⊨*as CNot D*›
using *C L M1-CNot* **unfolding** *D-def* **by** *auto*
moreover have ‹$cdcl_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of ?S*)›
using *invs* **unfolding** *twl-struct-invs-def* **by** *blast*
ultimately have *False*
using *undef M'*
by (*auto* 7 7 *simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def trail.simps clauses-def*)
then show ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# {#− *K''*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})›

by *fast*
qed
have ‹$cdcl_W$-*restart-mset.cdcl*$_W$-*conflicting* (*state*$_W$-*of ?T*)›
using *struct-inv-T* **unfolding** $cdcl_W$-*restart-mset.cdcl*$_W$-*all-struct-inv-def twl-struct-invs-def*
by (*auto simp*: *conflicting.simps*)
then have *M1-CNot-D*: ‹*M1* ⊨*as CNot* (*remove1-mset K′ D′*)›
unfolding $cdcl_W$-*restart-mset.cdcl*$_W$-*conflicting-def*
by (*auto simp*: *conflicting.simps trail.simps*)
then have *uK″-M1*: ‹−*K″* ∈ *lits-of-l M1*›
using *K″ K″-ne-K* **unfolding** *true-annots-true-cls-def-iff-negation-in-model*
by (*metis in-remove1-mset-neq*)
then have ‹*undefined-lit* (*M3* @ *M2* @ *Decided K* # []) *K″*›
using *n-d M* **by** (*auto simp*: *atm-of-eq-atm-of dest*: *in-lits-of-l-defined-litD defined-lit-no-dupD*)
then have *lev-M1-K″*: ‹*get-level M1 K″* = *count-decided M1*›
using *lev-K″ count-M1* **unfolding** *M* **by** (*auto simp*: *image-Un*)

have *excep-M1*: ‹*twl-st-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *past* **unfolding** *past-invs.simps M′* **by** *auto*

show *?case*
unfolding *twl-st-inv.simps Ball-def*
proof (*intro conjI allI impI*)
fix *C* :: ‹*′a twl-cls*›
assume *C*: ‹*C* ∈# *N* + *add-mset ?D U*›
have ‹$cdcl_W$-*restart-mset.distinct-cdcl*$_W$-*state* (*state*$_W$-*of ?T*)›
using *struct-inv-T* **unfolding** $cdcl_W$-*restart-mset.cdcl*$_W$-*all-struct-inv-def* **by** *blast*
then have ‹*distinct-mset D′*›
unfolding $cdcl_W$-*restart-mset.distinct-cdcl*$_W$-*state-def*
by (*auto simp*: $cdcl_W$-*restart-mset-state*)
then show *struct*: ‹*struct-wf-twl-cls C*›
using *inv C* **by** (*auto simp*: *twl-st-inv.simps D*)

obtain *CW CUW* **where** *C-W*: ‹*C* = *TWL-Clause CW CUW*›
by (*cases C*)
have
lazy: ‹*twl-lazy-update M1 C*› **and**
watched-max: ‹*watched-literals-false-of-max-level M1 C*› **if** ‹*C* ≠ *?D*›
using *C past M′ that* **by** (*auto simp*: *past-invs.simps*)
from *M1-CNot-D* **have** *in-D-M1*: ‹*L* ∈# *remove1-mset K′ D′* ⟹ − *L* ∈ *lits-of-l M1*› **for** *L*
by (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model*)
then have *in-K-D-M1*: ‹*L* ∈# *D′* − {#*K′*, *K″*#} ⟹ − *L* ∈ *lits-of-l M1*› **for** *L*
by (*metis K′-D′ add-mset-diff-bothsides add-mset-remove-trivial in-diffD mset-add*)
have ‹− *K′* ∉ *lits-of-l M1*›
using *n-d′* **by** (*simp add*: *Decided-Propagated-in-iff-in-lits-of-l*)
have *def-K″*: ‹*defined-lit M1 K″*›
using *n-d′ uK″-M1*
using *Decided-Propagated-in-iff-in-lits-of-l uK″-M1* **by** *blast*
have
lazy-D: ‹*twl-lazy-update ?M1 C*› **if** ‹*C* = *?D*›
using *that n-d′ uK″-M1 def-K″* ‹− *K′* ∉ *lits-of-l M1*› *in-K-D-M1 lev-M1-K″*
by (*auto simp*: *add-mset-eq-add-mset count-decided-ge-get-level get-level-cons-if*
atm-of-eq-atm-of)
have
watched-max-D: ‹*watched-literals-false-of-max-level ?M1 C*› **if** ‹*C* = *?D*›
using *that in-D-M1* **by** (*auto simp add*: *add-mset-eq-add-mset lev-M1-K″ get-level-cons-if*
dest: *in-K-D-M1*)

```
{
  assume excep: ‹¬ twl-is-an-exception C {#−K′#} {#}›

  have lev-le-Suc: ‹get-level M Ka ≤ Suc (count-decided M)› for Ka
    using count-decided-ge-get-level le-Suc-eq by blast
  have Lev-M1: ‹get-level (?M1) K ≤ count-decided M1› for K
    by (auto simp: count-decided-ge-get-level get-level-cons-if)

  have ‹twl-lazy-update ?M1 C› if ‹C ≠ ?D›
  proof −
    have 1: ‹get-level (Propagated K′ D′ # M1) K ≤ get-level (Propagated K′ D′ # M1) L›
      if
        ‹∀ L. L ∈# CW ⟶ − L ∈ lits-of-l M1 ⟶ ¬ has-blit M1 (CW + CUW) L ⟶
           get-level M1 L = count-decided M1› and
        ‹L ∈# CW› and
        ‹− L ∈ lits-of-l M1› and
        ‹K ∈# CUW› and
        ‹¬ has-blit M1 (CW + CUW) L›
      for L :: ‹′a literal› and K :: ‹′a literal›
      using that Lev-M1
      by (metis count-decided-ge-get-level get-level-skip-beginning get-level-take-beginning)
    have 2: False
      if
        ‹L ∈# CW› and
        ‹TWL-Clause CW CUW ∈# N› and
        ‹CW ≠ {#K′, K′′#}› and
        ‹− L ∈ lits-of-l M1› and
        ‹K ∈# CUW› and
        ‹− K ∉ lits-of-l M1› and
        ‹¬ has-blit M1 (CW + CUW) L›
      for L :: ‹′a literal› and K :: ‹′a literal›
      using lazy that unfolding C-W twl-lazy-update.simps by blast

    show ?thesis
      using Lev-M1 C-W that
      using twl C excep twl n-d′ watched-max 1
      unfolding C-W
      apply (auto simp: count-decided-ge-get-level
          twl-is-an-exception-add-mset-to-queue atm-of-eq-atm-of that
          dest!: no-has-blit-propagate′ no-has-blit-propagate dest: 2)
      using lazy unfolding C-W twl-lazy-update.simps apply blast
      using lazy unfolding C-W twl-lazy-update.simps apply blast
      using lazy unfolding C-W twl-lazy-update.simps apply blast
      done
  qed
  then show ‹twl-lazy-update ?M1 C›
    using lazy-D by blast
}

have ‹watched-literals-false-of-max-level M1 C› if ‹C ≠ ?D›
  using past C that unfolding M past-invs.simps by auto
then have ‹watched-literals-false-of-max-level ?M1 C› if ‹C ≠ ?D›
  using has-blit-Cons n-d′ C-W that by (auto simp: get-level-cons-if)
then show ‹watched-literals-false-of-max-level ?M1 C›
  using watched-max-D by blast
```

qed

case *2*
show *?case*
unfolding *past-invs.simps Ball-def*
proof (*intro allI impI conjI*)
fix *M1'' M2'' K''' C*
assume *M1*: ‹*?M1 = M2'' @ Decided K''' # M1''*› **and** *C*: ‹*C ∈# N + add-mset ?D U*›
then have *M1*: ‹*M1 = tl M2'' @ Decided K''' # M1''*›
by (*cases M2''*) *auto*
have ‹*twl-lazy-update M1'' C*›‹*watched-literals-false-of-max-level M1'' C*›
if ‹*C ≠ ?D*›
using *past C that* **unfolding** *past-invs.simps M M1 twl-exception-inv.simps* **by** *auto*
moreover {
have ‹*twl-exception-inv* (*M1''*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*› **if** ‹*C ≠ ?D*›
using *past C* **unfolding** *past-invs.simps M M1* **by** (*auto simp*: *that*)
then have ‹*twl-exception-inv* (*M1''*, *N*, *add-mset ?D U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
if ‹*C ≠ ?D*›
using *C* **unfolding** *twl-exception-inv.simps* **by** (*auto simp*: *that*) **}**
moreover {
have *n-d-M1*: ‹*no-dup ?M1*›
using *struct-inv-T* **unfolding** *cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def **by** (*simp add*: *cdcl$_W$-restart-mset-state*)
then have ‹*undefined-lit M1'' K'*›
unfolding *M1* **by** *auto*
moreover {
have ‹*- K'' ∉ lits-of-l M1''*›
proof (*rule ccontr*)
assume ‹*¬ - K'' ∉ lits-of-l M1''*›
then have ‹*undefined-lit* (*tl M2'' @ Decided K''' # []*) *K''*›
using *n-d-M1* **unfolding** *M1* **by** (*auto simp*: *atm-lit-of-set-lits-of-l*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
defined-lit-map atm-of-eq-atm-of image-Un
dest: *no-dup-uminus-append-in-atm-notin*)
then show *False*
using *lev-M1-K'' count-decided-ge-get-level*[*of M1'' K''*] **unfolding** *M1*
by (*auto simp*: *image-Un Int-Un-distrib*)
qed **}**
ultimately have ‹*twl-lazy-update M1'' ?D*› **and**
‹*watched-literals-false-of-max-level M1'' ?D*› **and**
‹*twl-exception-inv* (*M1''*, *N*, *add-mset* (*TWL-Clause* {#*K'*, *K''*#} (*D'* − {#*K'*, *K''*#})) *U*, *None*,
NE, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *?D*›
by (*auto simp*: *add-mset-eq-add-mset twl-exception-inv.simps get-level-cons-if*
Decided-Propagated-in-iff-in-lits-of-l) **}**
ultimately show ‹*twl-lazy-update M1'' C*›
‹*watched-literals-false-of-max-level M1'' C*›
‹*twl-exception-inv* (*M1''*, *N*, *add-mset* (*TWL-Clause* {#*K'*, *K''*#} (*D'* − {#*K'*, *K''*#})) *U*, *None*,
NE, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
by *blast+*
next
fix *M1'' M2'' K'''*
assume *M1*: ‹*?M1 = M2'' @ Decided K''' # M1''*›

```
    then have M1: ‹M1 = tl M2'' @ Decided K''' # M1''›
      by (cases M2'') auto
    then have confl-cands: ‹confl-cands-enqueued (M1'', N, U, None, NE, UE, NS, US, N0, U0, {#},
{#})› and
      propa-cands: ‹propa-cands-enqueued (M1'', N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
and
      w-q: ‹clauses-to-update-inv (M1'', N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
      using past by (auto simp add: M M1 past-invs.simps simp del: propa-cands-enqueued.simps
          confl-cands-enqueued.simps)
    have uK''-M1'': ‹- K'' ∉ lits-of-l M1''›
    proof (rule ccontr)
      assume K''-M1'': ‹¬ ?thesis›
      have ‹undefined-lit (tl M2'' @ Decided K''' # []) (-K'')›
        apply (rule no-dup-append-in-atm-notin)
         prefer 2 using K''-M1'' apply (simp; fail)
        by (use n-d in ‹auto simp: M M1 no-dup-def; fail›)[]
      then show False
        using lev-M1-K'' count-decided-ge-get-level[of M1'' K''] unfolding M M1
        by (auto simp: image-Un)
    qed
    have uK'-M1'': ‹- K' ∉ lits-of-l M1''›
    proof (rule ccontr)
      assume K'-M1'': ‹¬ ?thesis›
      have ‹undefined-lit (M3 @ M2 @ Decided K # tl M2'' @ Decided K''' # []) (-K')›
        apply (rule no-dup-append-in-atm-notin)
         prefer 2 using K'-M1'' apply (simp; fail)
        by (use n-d in ‹auto simp: M M1; fail›)[]
      then show False
        using lev-K' count-decided-ge-get-level[of M1'' K'] unfolding M M1
        by (auto simp: image-Un)
    qed

    have [simp]: ‹¬clauses-to-update-prop {#} M1'' (L, ?D)› for L
      using uK'-M1'' uK''-M1'' by (auto simp: clauses-to-update-prop.simps add-mset-eq-add-mset)
     show ‹confl-cands-enqueued (M1'', N, add-mset ?D U, None, NE, UE, NS, US, N0, U0, {#},
{#})› and
      ‹propa-cands-enqueued (M1'', N, add-mset ?D U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
and
      ‹clauses-to-update-inv (M1'', N, add-mset ?D U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
      using confl-cands propa-cands w-q uK'-M1'' uK''-M1''
      by (fastforce simp add: twl-st-inv.simps add-mset-eq-add-mset)+
  qed
qed

lemma
  assumes
    cdcl: ‹cdcl-twl-o S T›
  shows
    cdcl-twl-o-valid: ‹valid-enqueued T› and
    cdcl-twl-o-conflict-None-queue:
      ‹get-conflict T ≠ None ⟹ clauses-to-update T = {#} ∧ literals-to-update T = {#}› and
      cdcl-twl-o-no-duplicate-queued: ‹no-duplicate-queued T› and
      cdcl-twl-o-distinct-queued: ‹distinct-queued T›
  using cdcl by (induction rule: cdcl-twl-o.induct) auto

lemma cdcl-twl-o-twl-st-exception-inv:
```

assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-struct-invs S*›
shows
‹*twl-st-exception-inv T*›
using *cdcl twl*
proof (*induction rule*: *cdcl-twl-o.induct*)
case (*decide M L N NE NS N0 U UE US U0*) **note** *undef* = *this*(*1*) **and** *in-atms* = *this*(*2*) **and** *twl* = *this*(*3*)
then have *excep*: ‹*twl-st-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
unfolding *twl-struct-invs-def*
by (*auto simp*: *twl-exception-inv.simps*)
let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
have *struct-inv-T*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?S*)›
using $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-restart-mset.other twl*
unfolding *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *twl* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*)
show *?case*
using *decide.hyps n-d excep*
unfolding *twl-struct-invs-def*
by (*auto simp*: *twl-exception-inv.simps dest*!: *no-has-blit-decide*′)
next
case (*skip L D C*′ *M N U NE UE NS US*)
then show *?case*
unfolding *twl-struct-invs-def* **by** (*auto simp*: *twl-exception-inv.simps*)
next
case (*resolve L D C M N U NE UE NS US*)
then show *?case*
unfolding *twl-struct-invs-def* **by** (*auto simp*: *twl-exception-inv.simps*)
next
case (*backtrack-unit-clause L D K M1 M2 M D*′ *i N U NE UE NS US N0 U0*) **note** *decomp* = *this*(*2*)
and
invs = *this*(*10*)
let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?S*′ = ‹*state*$_W$*-of S*›
let *?T* = ‹(*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?T*′ = ‹*state*$_W$*-of T*›
let *?U* = ‹(*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#− *L*#})›
let *?U*′ = ‹*state*$_W$*-of ?U*›
have ‹*twl-st-inv ?S*› **and** *past*: ‹*past-invs ?S*› **and** *valid*: ‹*valid-enqueued ?S*›
using *invs decomp* **unfolding** *twl-struct-invs-def* **by** *fast*+
then have *excep*: ‹*twl-exception-inv ?T C*› **if** ‹*C* ∈# *N* + *U*› **for** *C*
using *decomp that* **unfolding** *past-invs.simps* **by** *auto*
have *struct-inv-T*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?S*)›
using *invs* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*simp add*: $cdcl_W$*-restart-mset-state pcdcl-all-struct-invs-def*)
then have *n-d*: ‹*no-dup M1*›
using *decomp* **by** (*auto dest*: *no-dup-appendD*)

have *struct-inv-U*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?U*)›
using *cdcl-twl-o-cdcl*$_W$*-o*[*OF cdcl-twl-o.backtrack-unit-clause*[*OF backtrack-unit-clause.hyps*]
‹*twl-st-inv ?S*› *valid struct-inv-T*] *invs*
pcdcl-tcore-pcdcl-all-struct-invs[*of* ‹*pstate*$_W$*-of ?S*› ‹*pstate*$_W$*-of ?U*›]
struct-inv-T **unfolding** *pcdcl-all-struct-invs-def twl-struct-invs-def* **by** *auto*
then have *undef*: ‹*undefined-lit M1 L*›
unfolding *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*simp add*: *cdcl*$_W$*-restart-mset-state*)

show *?case*
using *n-d excep undef*
unfolding *twl-struct-invs-def*
by (*auto simp*: *twl-exception-inv.simps dest*!: *no-has-blit-propagate'*)
next
case (*backtrack-nonunit-clause L D K M1 M2 M D' i N U NE UE NS US N0 U0 L'*) **note** *decomp* = *this*(*2*) **and**
lev-K = *this*(*6*) **and** *lev-L'* = *this*(*12*) **and** *invs* = *this*(*13*)
let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?D* = ‹*TWL-Clause* {#*L*, *L'*#} (*D'* − {#*L*, *L'*#})›
let *?T* = ‹(*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?U* = ‹(*Propagated L D'* # *M1*, *N*, *add-mset ?D U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#−*L*#})›
have ‹*twl-st-inv ?S*› **and** *past*: ‹*past-invs ?S*› **and** *valid*: ‹*valid-enqueued ?S*›
using *invs decomp* **unfolding** *twl-struct-invs-def* **by** *fast+*
then have *excep*: ‹*twl-exception-inv ?T C*› **if** ‹*C* ∈# *N* + *U*› **for** *C*
using *decomp that* **unfolding** *past-invs.simps* **by** *auto*
have *struct-inv-T*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of ?S*)›
using *invs* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
have *n-d-M*: ‹*no-dup M*›
using *invs* **unfolding** *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*simp add*: *cdcl*$_W$*-restart-mset-state*)
then have *n-d*: ‹*no-dup M1*›
using *decomp* **by** (*auto dest*: *no-dup-appendD*)

have *struct-inv-U*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of ?U*)›
using *cdcl-twl-o-cdcl*$_W$*-o*[*OF cdcl-twl-o.backtrack-nonunit-clause*[*OF backtrack-nonunit-clause.hyps*]
‹*twl-st-inv ?S*› *valid struct-inv-T*] *invs*
pcdcl-tcore-pcdcl-all-struct-invs[*of* ‹*pstate*$_W$*-of ?S*› ‹*pstate*$_W$*-of ?U*›]
struct-inv-T **unfolding** *pcdcl-all-struct-invs-def twl-struct-invs-def* **by** *auto*
then have *undef*: ‹*undefined-lit M1 L*›
unfolding *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*simp add*: *cdcl*$_W$*-restart-mset-state*)

have *n-d*: ‹*no-dup* (*Propagated L D'* # *M1*)›
using *struct-inv-U* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
by (*simp add*: *trail.simps*)
have ‹*i* = *count-decided M1*›
using *decomp lev-K n-d-M* **by** (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *get-level-append-if get-level-cons-if*
split: *if-splits*)
then have *lev-L'-M1*: ‹*get-level* (*Propagated L D'* # *M1*) *L'* = *count-decided M1*›
using *decomp lev-L' n-d-M* **by** (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *get-level-append-if get-level-cons-if*

```
      split: if-splits)
  have ‹- L ∉ lits-of-l M1›
    using n-d by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
  moreover have ‹has-blit (Propagated L D' # M1) (add-mset L (add-mset L' (D' - {#L, L'#}))) L'›
    unfolding has-blit-def
    apply (rule exI[of - L])
    using lev-L' lev-L'-M1
    by auto
  ultimately show ?case
    using n-d excep undef
    unfolding twl-struct-invs-def
    by (auto simp: twl-exception-inv.simps dest!: no-has-blit-propagate')
qed


lemma
  assumes
    cdcl: ‹cdcl-twl-o S T› and
    twl: ‹twl-struct-invs S›
  shows
    cdcl-twl-o-confl-cands-enqueued: ‹confl-cands-enqueued T› and
    cdcl-twl-o-propa-cands-enqueued: ‹propa-cands-enqueued T› and
    twl-o-clauses-to-update: ‹clauses-to-update-inv T›
  using cdcl twl
proof (induction rule: cdcl-twl-o.induct)
  case (decide M L N NE NS N0 U UE US U0)
  let ?S = ‹(M, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
  let ?T = ‹(Decided L # M, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#-L#})›
  case 1
  then have confl-cand: ‹confl-cands-enqueued ?S› and
    twl-st-inv: ‹twl-st-inv ?S› and
    excep: ‹twl-st-exception-inv ?S› and
    propa-cands: ‹propa-cands-enqueued ?S› and
    confl-cands: ‹confl-cands-enqueued ?S› and
    w-q: ‹clauses-to-update-inv ?S›
    unfolding twl-struct-invs-def by fast+

  have ‹pcdcl-tcore (pstate_W-of ?S) (pstate_W-of ?T)›
    by (rule cdcl-twl-o-cdcl_W-o) (use cdcl-twl-o.decide[OF decide.hyps] 1 in
        ‹simp-all add: twl-struct-invs-def pcdcl-all-struct-invs-def›)
  from pcdcl-tcore-pcdcl-all-struct-invs[OF this]
  have ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of ?T)›
    using 1 unfolding pcdcl-all-struct-invs-def twl-struct-invs-def by auto
  then have n-d: ‹no-dup (Decided L # M)›
    unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def cdcl_W-restart-mset.cdcl_W-M-level-inv-def
    by (auto simp: trail.simps)
  show ?case
    unfolding confl-cands-enqueued.simps Ball-def
  proof (intro allI impI)
    fix C
    assume
      C: ‹C ∈# N + U› and
      LM-C: ‹Decided L # M ⊨as CNot (clause C)›

    have struct-C: ‹struct-wf-twl-cls C›
      using twl-st-inv C unfolding twl-st-inv.simps by blast
```

then have *dist-C*: ‹*distinct-mset* (*clause C*)›
by (*cases C*) *auto*
obtain *W UW K K′* **where**
C-W: ‹*C* = *TWL-Clause W UW*› **and**
W: ‹*W* = {#*K*, *K′*#}›
using *struct-C* **by** (*cases C*) (*auto simp*: *size-2-iff*)

have ‹¬*M* ⊨*as CNot* (*clause C*)›
using *confl-cand C* **by** *auto*
then have *uL-C*: ‹−*L* ∈# *clause C*› **and** *neg-C*: ‹∀ *K* ∈# *clause C*. −*K* ∈ *lits-of-l* (*Decided L* # *M*)›
using *LM-C* **unfolding** *true-annots-true-cls-def-iff-negation-in-model* **by** *auto*
have ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *excep C* **by** *auto*
then have *H*: ‹*L* ∈# *watched* (*TWL-Clause* {#*K*, *K′*#} *UW*) ⟶
− *L* ∈ *lits-of-l M* ⟶ ¬ *has-blit M* (*clause* (*TWL-Clause* {#*K*, *K′*#} *UW*)) *L* ⟶
L ∉# {#} ⟶
(*L*, *TWL-Clause* {#*K*, *K′*#} *UW*) ∉# {#} ⟶
(∀ *K*∈#*unwatched* (*TWL-Clause* {#*K*, *K′*#} *UW*).
− *K* ∈ *lits-of-l M*)› **for** *L*
unfolding *twl-exception-inv.simps C-W W* **by** *blast*
have *excep*: ‹*L* ∈# *watched* (*TWL-Clause* {#*K*, *K′*#} *UW*) ⟶
− *L* ∈ *lits-of-l M* ⟶ ¬ *has-blit M* (*clause* (*TWL-Clause* {#*K*, *K′*#} *UW*)) *L* ⟶
(∀ *K*∈#*unwatched* (*TWL-Clause* {#*K*, *K′*#} *UW*). − *K* ∈ *lits-of-l M*)› **for** *L*
using *H*[*of L*] **by** *simp*
have ‹−*L* ∈# *watched C*›
proof (*rule ccontr*)
assume *uL-W*: ‹−*L* ∉# *watched C*›
then have *uL-UW*: ‹−*L* ∈# *UW*›
using *uL-C* **unfolding** *C-W* **by** *auto*
have ‹*K* ≠ −*L* ∨ *K′* ≠ −*L*›
using *dist-C C-W W* **by** *auto*
moreover have ‹*K* ∉ *lits-of-l M*› **and** ‹*K′* ∉ *lits-of-l M*› **and** *L-M*: ‹*L* ∉ *lits-of-l M*›
using *neg-C uL-W n-d* **unfolding** *C-W W* **by** (*auto simp*: *lits-of-def uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus Decided-Propagated-in-iff-in-lits-of-l)
ultimately have *disj*: ‹(−*K* ∈ *lits-of-l M* ∧ *K′* ∉ *lits-of-l M*) ∨
(−*K′* ∈ *lits-of-l M* ∧ *K* ∉ *lits-of-l M*)›
using *neg-C* **by** (*auto simp*: *C-W W*)
have ‹¬*has-blit M* (*clause C*) *K*›
using ‹*K* ∉ *lits-of-l M*› ‹*K′* ∉ *lits-of-l M*›
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **by** (*auto dest*!: *multi-member-split*
dest!: *no-dup-consistentD*
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹¬*has-blit M* (*clause C*) *K′*›
using ‹*K′* ∉ *lits-of-l M*› ‹ *K* ∉ *lits-of-l M*›
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **by** (*auto dest*!: *multi-member-split*
dest!: *no-dup-consistentD*
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
ultimately have ‹∀ *K* ∈# *unwatched C*. −*K* ∈ *lits-of-l M*›
apply −
apply (*rule disjE*[*OF disj*])
subgoal
using *excep*[*of K*]
unfolding *C-W twl-clause.sel member-add-mset W*
by *auto*
subgoal

```
      using excep[of K']
      unfolding C-W twl-clause.sel member-add-mset W
      by auto
    done
  then show False
    using uL-W uL-C L-M unfolding C-W W by auto
 qed
 then show ‹(∃L'. L' ∈# watched C ∧ L' ∈# {#- L#}) ∨ (∃L. (L, C) ∈# {#})›
  by auto
qed

case 2
show ?case
 unfolding propa-cands-enqueued.simps Ball-def
proof (intro allI impI)
 fix FK C
 assume
  C: ‹C ∈# N + U› and
  K: ‹FK ∈# clause C› and
  LM-C: ‹Decided L # M ⊨as CNot (remove1-mset FK (clause C))› and
  undef: ‹undefined-lit (Decided L # M) FK›
 have undef-M-K: ‹undefined-lit M FK›
  using undef by (auto simp: defined-lit-map)
 then have ‹¬ M ⊨as CNot (remove1-mset FK (clause C))›
  using propa-cands C K undef by auto
 then have ‹-L ∈# clause C› and
  neg-C: ‹∀K ∈# remove1-mset FK (clause C). -K ∈ lits-of-l (Decided L # M)›
  using LM-C undef-M-K by (force simp: true-annots-true-cls-def-iff-negation-in-model
     dest: in-diffD)+

 have struct-C: ‹struct-wf-twl-cls C›
  using twl-st-inv C unfolding twl-st-inv.simps by blast
 then have dist-C: ‹distinct-mset (clause C)›
  by (cases C) auto

 have ‹-L ∈# watched C›
 proof (rule ccontr)
  assume uL-W: ‹-L ∉# watched C›
  then obtain W UW K K' where
   C-W: ‹C = TWL-Clause W UW› and
   W: ‹W = {#K, K'#}› and
   uK-M: ‹-K ∈ lits-of-l M›
   using struct-C neg-C by (cases C) (auto simp: size-2-iff remove1-mset-add-mset-If
     add-mset-commute split: if-splits)
  have FK-F: ‹FK ≠ K›
   using Decided-Propagated-in-iff-in-lits-of-l uK-M undef-M-K by blast
  have L-M: ‹undefined-lit M L›
   using neg-C uL-W n-d unfolding C-W W by auto
  then have ‹K ≠ -L›
   using uK-M by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
  moreover have ‹K ∉ lits-of-l M›
   using neg-C uL-W n-d uK-M by (auto simp: lits-of-def uminus-lit-swap
      no-dup-cannot-not-lit-and-uminus)
  ultimately have ‹K' ∉ lits-of-l M›
   apply (cases ‹K' = FK›)
   using Decided-Propagated-in-iff-in-lits-of-l undef-M-K apply blast
```

using *neg-C C-W W FK-F n-d uL-W* **by** (*auto simp add*: *remove1-mset-add-mset-If uminus-lit-swap*
lits-of-def no-dup-cannot-not-lit-and-uminus)
moreover have ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *excep C* **by** *auto*

moreover have ‹¬*has-blit M* (*clause C*) *K*›
using ‹*K* ∉ *lits-of-l M*› ‹*K′* ∉ *lits-of-l M*›
using *K in-lits-of-l-defined-litD neg-C undef-M-K n-d* **unfolding** *has-blit-def*
by (*force dest*!: *multi-member-split*
dest!: *no-dup-consistentD*
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹¬*has-blit M* (*clause C*) *K′*›
using ‹*K′* ∉ *lits-of-l M*› ‹ *K* ∉ *lits-of-l M*› *K in-lits-of-l-defined-litD neg-C undef-M-K*
using *n-d* **unfolding** *has-blit-def* **by** (*force dest*!: *multi-member-split*
dest!: *no-dup-consistentD*
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
ultimately have ‹∀ *K* ∈# *unwatched C*. −*K* ∈ *lits-of-l M*›
using *uK-M*
by (*auto simp*: *twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib*)
then show *False*
using *C-W L-M*(*1*) ‹− *L* ∈# *clause C*› *uL-W*
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
then show ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# {#− *L*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})›
by *auto*
qed

case *3*
show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty L C*)
then show *?case* **by** *simp*
next
case (*WS-empty K*)
then show *?case*
using *w-q n-d* **unfolding** *clauses-to-update-prop.simps*
by (*auto simp add*: *filter-mset-empty-conv*
dest!: *no-has-blit-decide′*)
next
case (*Q K C*)
then show *?case*
using *w-q n-d* **by** (*auto dest*!: *no-has-blit-decide′*)
qed
next
case (*skip L D C′ M N U NE UE NS US*)
case *1* **then show** *?case* **by** *auto*
case *2* **then show** *?case* **by** *auto*
case *3* **then show** *?case* **by** *auto*
next
case (*resolve L D C M N U NE UE NS US*)
case *1* **then show** *?case* **by** *auto*
case *2* **then show** *?case* **by** *auto*
case *3* **then show** *?case* **by** *auto*
next
case (*backtrack-unit-clause L D K M1 M2 M D′ i N U NE UE NS US N0 U0*) **note** *decomp* = *this*(*2*)

let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?U* = ‹(*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#− *L*#})›
obtain *M3* **where**
M: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** *blast*

case *1*
then have *twl-st-inv*: ‹*twl-st-inv ?S*› **and**
struct-inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?S*)› **and**
excep: ‹*twl-st-exception-inv ?S*› **and**
past: ‹*past-invs ?S*›
using *decomp* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
then have
confl-cands: ‹*confl-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
propa-cands: ‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›**and**
w-q: ‹*clauses-to-update-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *decomp* **unfolding** *past-invs.simps* **by** (*auto simp del*: *clauses-to-update-inv.simps*)

have *n-d*: ‹*no-dup M*›
using *struct-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
have ‹*pcdcl-tcore* (*pstate*$_W$*-of ?S*) (*pstate*$_W$*-of ?U*)›
using *cdcl-twl-o-*$cdcl_W$*-o*[*OF cdcl-twl-o.backtrack-unit-clause*[*OF backtrack-unit-clause.hyps*]]
1 **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
from *pcdcl-tcore-pcdcl-all-struct-invs*[*OF this*]
have *struct-inv-T*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?U*)›
using *1* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *auto*
then have *n-d-L-M1*: ‹*no-dup* (*Propagated L* {#*L*#} # *M1*)›
using *struct-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
then have *uL-M1*: ‹*undefined-lit M1 L*›
by (*simp-all add*: *atm-lit-of-set-lits-of-l atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)

have *excep-M1*: ‹∀ *C* ∈# *N* + *U*. *twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *past* **unfolding** *past-invs.simps M* **by** *auto*

show *?case*
unfolding *confl-cands-enqueued.simps Ball-def*
proof (*intro allI impI*)
fix *C*
assume
C: ‹*C* ∈# *N* + *U*› **and**
LM-C: ‹*Propagated L* {#*L*#} # *M1* ⊨*as CNot* (*clause C*)›

have *struct-C*: ‹*struct-wf-twl-cls C*›
using *twl-st-inv C* **unfolding** *twl-st-inv.simps* **by** *auto*
then have *dist-C*: ‹*distinct-mset* (*clause C*)›
by (*cases C*) *auto*

obtain *W UW K K′* **where**
C-W: ‹*C* = *TWL-Clause W UW*› **and**
W: ‹*W* = {#*K*, *K′*#}›
using *struct-C* **by** (*cases C*) (*auto simp*: *size-2-iff*)

have ‹$\neg M1 \models as\ CNot\ (clause\ C)$›
using *confl-cands C* **by** *auto*
then have *uL-C*: ‹$-L \in\#\ clause\ C$› **and** *neg-C*: ‹$\forall K \in\#\ clause\ C.\ -K \in lits\text{-}of\text{-}l\ (Decided\ L\ \#\ M1)$›
using *LM-C* **unfolding** *true-annots-true-cls-def-iff-negation-in-model* **by** *auto*
have *K-L*: ‹$K \neq L$› **and** *K'-L*: ‹$K' \neq L$›
apply (*metis C-W LM-C W add-diff-cancel-right' clause.simps consistent-interp-def distinct-consistent-interp in-CNot-implies-uminus*(2) *in-diffD n-d-L-M1 uL-C union-single-eq-member*)
using *C-W LM-C W uL-M1* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
have ‹$-L \in\#\ watched\ C$›
proof (*rule ccontr*)
assume *uL-W*: ‹$-L \notin\#\ watched\ C$›
have ‹$K \neq -L \vee K' \neq -L$›
using *dist-C C-W W* **by** *auto*
moreover have ‹$K \notin lits\text{-}of\text{-}l\ M1$› **and** ‹$K' \notin lits\text{-}of\text{-}l\ M1$› **and** *L-M*: ‹$L \notin lits\text{-}of\text{-}l\ M1$›
proof −
have *f2*: ‹$consistent\text{-}interp\ (lits\text{-}of\text{-}l\ M1)$›
using *distinct-consistent-interp n-d-L-M1* **by** *auto*
have *undef-L*: ‹$undefined\text{-}lit\ M1\ L$›
using *atm-lit-of-set-lits-of-l n-d-L-M1* **by** *force*
then show ‹$K \notin lits\text{-}of\text{-}l\ M1$›
using *f2 neg-C* **unfolding** *C-W W* **by** (*metis* (*no-types*) *C-W W add-diff-cancel-right' atm-of-eq-atm-of clause.simps consistent-interp-def in-diffD insertE list.simps*(15) *lits-of-insert uL-C union-single-eq-member Decided-Propagated-in-iff-in-lits-of-l*)
show ‹$K' \notin lits\text{-}of\text{-}l\ M1$›
using *consistent-interp-def distinct-consistent-interp n-d-L-M1*
using *neg-C uL-W n-d* **unfolding** *C-W W* **by** *auto*
show ‹$L \notin lits\text{-}of\text{-}l\ M1$›
using *undef-L* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
ultimately have ‹$(-K \in lits\text{-}of\text{-}l\ M1 \wedge K' \notin lits\text{-}of\text{-}l\ M1) \vee (-K' \in lits\text{-}of\text{-}l\ M1 \wedge K \notin lits\text{-}of\text{-}l\ M1)$›
using *neg-C* **by** (*auto simp*: *C-W W*)
moreover have ‹$twl\text{-}exception\text{-}inv\ (M1, N, U, None, NE, UE, NS, US, N0, U0, \{\#\}, \{\#\})\ C$›
using *excep-M1 C* **by** *auto*
have ‹$\neg has\text{-}blit\ M1\ (clause\ C)\ K$›
using ‹$K \notin lits\text{-}of\text{-}l\ M1$› ‹$K' \notin lits\text{-}of\text{-}l\ M1$› ‹$L \notin lits\text{-}of\text{-}l\ M1$› *uL-M1 n-d-L-M1 no-dup-cons*
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto dest*!: *multi-member-split dest*!: *no-dup-consistentD*[*OF n-d-L-M1*] *dest*!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
using *n-d-L-M1 no-dup-cons no-dup-consistentD* **by** *blast*
moreover have ‹$\neg has\text{-}blit\ M1\ (clause\ C)\ K'$›
using ‹$K' \notin lits\text{-}of\text{-}l\ M1$› ‹ $K \notin lits\text{-}of\text{-}l\ M1$› ‹$L \notin lits\text{-}of\text{-}l\ M1$› *uL-M1 n-d-L-M1 no-dup-cons no-dup-consistentD*
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto 10 10 dest*!: *multi-member-split dest*!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
using *n-d-L-M1 no-dup-cons no-dup-consistentD* **by** *auto*
ultimately have ‹$\forall K \in\#\ unwatched\ C.\ -K \in lits\text{-}of\text{-}l\ M1$›
using *C twl-clause.sel*(1) *union-single-eq-member w-q*
by (*fastforce simp*: *twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib L-M*)
then show *False*
using *uL-W uL-C L-M K-L uL-M1* **unfolding** *C-W W* **by** *auto*

```
  qed
  then show ‹(∃ L'. L' ∈# watched C ∧ L' ∈# {#- L#}) ∨ (∃ L. (L, C) ∈# {#})›
    by auto
qed
case 2
then show ?case
  unfolding propa-cands-enqueued.simps Ball-def
proof (intro allI impI)
  fix FK C
  assume
    C: ‹C ∈# N + U› and
    K: ‹FK ∈# clause C› and
    LM-C: ‹Propagated L {#L#} # M1 ⊨as CNot (remove1-mset FK (clause C))› and
    undef: ‹undefined-lit (Propagated L {#L#} # M1) FK›
  have undef-M-K: ‹undefined-lit (Propagated L D # M1) FK›
    using undef by (auto simp: defined-lit-map)
  then have ‹¬ M1 ⊨as CNot (remove1-mset FK (clause C))›
    using propa-cands C K undef by (auto simp: defined-lit-map)
  then have uL-C: ‹-L ∈# clause C› and
    neg-C: ‹∀ K ∈# remove1-mset FK (clause C). -K ∈ lits-of-l (Propagated L D # M1)›
    using LM-C undef-M-K by (force simp: true-annots-true-cls-def-iff-negation-in-model
        dest: in-diffD)+

  have struct-C: ‹struct-wf-twl-cls C›
    using twl-st-inv C unfolding twl-st-inv.simps by blast
  then have dist-C: ‹distinct-mset (clause C)›
    by (cases C) auto

  moreover have ‹-L ∈# watched C›
  proof (rule ccontr)
    assume uL-W: ‹-L ∉# watched C›
    then obtain W UW K K' where
      C-W: ‹C = TWL-Clause W UW› and
      W: ‹W = {#K, K'#}› and
      uK-M: ‹-K ∈ lits-of-l M1›
      using struct-C neg-C by (cases C) (auto simp: size-2-iff remove1-mset-add-mset-If
          add-mset-commute split: if-splits)
    have ‹K ∉ lits-of-l M1› and L-M: ‹L ∉ lits-of-l M1›
    proof -
      have f2: ‹consistent-interp (lits-of-l M1)›
        using distinct-consistent-interp n-d-L-M1 by auto
      have undef-L: ‹undefined-lit M1 L›
        using atm-lit-of-set-lits-of-l n-d-L-M1 by force
      then show ‹K ∉ lits-of-l M1›
        using f2 neg-C unfolding C-W W
        using n-d-L-M1 no-dup-cons no-dup-consistentD uK-M by blast
      show ‹L ∉ lits-of-l M1›
        using undef-L by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
    qed
    have FK-F: ‹FK ≠ K›
      using uK-M undef-M-K unfolding Decided-Propagated-in-iff-in-lits-of-l by auto
    have ‹K ≠ -L›
      using uK-M uL-M1 by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
    moreover have ‹K ∉ lits-of-l M1›
      using neg-C uL-W n-d uK-M n-d-L-M1 by (auto simp: lits-of-def uminus-lit-swap
          no-dup-cannot-not-lit-and-uminus dest: no-dup-cannot-not-lit-and-uminus)
```

```
    ultimately have ‹K' ∉ lits-of-l M1›
      apply (cases ‹K' = FK›)
      using undef-M-K apply (force simp: Decided-Propagated-in-iff-in-lits-of-l)
      using neg-C C-W W FK-F n-d uL-W n-d-L-M1 by (auto simp add: remove1-mset-add-mset-If
          uminus-lit-swap lits-of-def no-dup-cannot-not-lit-and-uminus
          dest: no-dup-cannot-not-lit-and-uminus)
    moreover have ‹twl-exception-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#}) C›
      using excep-M1 C by auto
    moreover have ‹¬has-blit M1 (clause C) K›
      using ‹K ∉ lits-of-l M1› ‹K' ∉ lits-of-l M1› ‹L ∉ lits-of-l M1› uL-M1
        n-d-L-M1 no-dup-cons K undef
      using uL-C neg-C n-d unfolding has-blit-def apply (auto dest!: multi-member-split
          dest!: no-dup-consistentD[OF n-d-L-M1]
          dest!: in-lits-of-l-defined-litD[of ‹-L›] simp: add-mset-eq-add-mset)
      by (smt add-mset-commute add-mset-eq-add-mset defined-lit-uminus in-lits-of-l-defined-litD
          insert-DiffM no-dup-consistentD set-subset-Cons true-annot-mono true-annot-singleton)+
    moreover have ‹¬has-blit M1 (clause C) K'›
      using ‹K' ∉ lits-of-l M1› ‹ K ∉ lits-of-l M1› ‹L ∉ lits-of-l M1› uL-M1
        n-d-L-M1 no-dup-cons no-dup-consistentD K undef
      using uL-C neg-C n-d unfolding has-blit-def apply (auto 10 10 dest!: multi-member-split
          dest!: in-lits-of-l-defined-litD[of ‹-L›] simp: add-mset-eq-add-mset)
      by (smt add-mset-commute add-mset-eq-add-mset defined-lit-uminus in-lits-of-l-defined-litD
          insert-DiffM no-dup-consistentD set-subset-Cons true-annot-mono true-annot-singleton)+
    ultimately have ‹∀K ∈# unwatched C. -K ∈ lits-of-l M1›
      using uK-M
      by (auto simp: twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib)
    then show False
      using C-W uL-M1 ‹- L ∈# clause C› uL-W
      by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
  qed
  then show ‹(∃L'. L' ∈# watched C ∧ L' ∈# {#- L#}) ∨ (∃L. (L, C) ∈# {#})›
    by auto
qed

case 3
have
  2: ‹⋀L. Pair L `# {#C ∈# N + U. clauses-to-update-prop {#} M1 (L, C)#} = {#}› and
  3: ‹⋀L C. C ∈# N + U ⟹ L ∈# watched C ⟹ - L ∈ lits-of-l M1 ⟹
    ¬ has-blit M1 (clause C) L ⟹ (L, C) ∉# {#} ⟹ L ∈# {#}›
  using w-q unfolding clauses-to-update-inv.simps by auto


show ?case
proof (induction rule: clauses-to-update-inv-cases)
  case (WS-nempty L C)
  then show ?case by simp
next
  case (WS-empty K)
  then show ?case
    using 2[of K]  n-d-L-M1
    apply (simp only: filter-mset-empty-conv Ball-def image-mset-is-empty-iff)
    by (auto simp add: clauses-to-update-prop.simps)
next
  case (Q K C)
  then show ?case
    using 3[of C K] has-blit-Cons n-d-L-M1 by (fastforce simp add: clauses-to-update-prop.simps)
```

```
  qed
next
  case (backtrack-nonunit-clause L D K M1 M2 M D' i N U NE UE NS US N0 U0 L') note LD =
this(1) and
    decomp = this(2) and lev-L = this(3) and lev-max-L = this(4) and i = this(5) and lev-K =
this(6)
    and LD' = this(11) and lev-L' = this(12)
  let ?S = ‹(M, N, U, Some D, NE, UE, NS, US, N0, U0, {#}, {#})›
  let ?D = ‹TWL-Clause {#L, L'#} (D' − {#L, L'#})›
  let ?U = ‹(Propagated L D' # M1, N, add-mset ?D U, None, NE,
    UE, NS, US, N0, U0, {#}, {#− L#})›
  obtain M3 where
    M: ‹M = M3 @ M2 @ Decided K # M1›
    using decomp by blast

  case 1
  then have twl-st-inv: ‹twl-st-inv ?S› and
    struct-inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of ?S)› and
    excep: ‹twl-st-exception-inv ?S› and
    past: ‹past-invs ?S›
    using decomp unfolding twl-struct-invs-def pcdcl-all-struct-invs-def by auto
  then have
    confl-cands: ‹confl-cands-enqueued (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})› and
    propa-cands: ‹propa-cands-enqueued (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})› and
    w-q: ‹clauses-to-update-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
    using decomp unfolding past-invs.simps by auto

  have n-d: ‹no-dup M›
    using struct-inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
     cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps)

  have ‹undefined-lit (M3 @ M2 @ M1) K›
    by (rule no-dup-append-in-atm-notin[of - ‹[Decided K]›])
      (use n-d M in ‹auto simp: no-dup-def›)
  then have L-uL': ‹L ≠ − L'›
    using lev-L lev-L' lev-K unfolding M by (auto simp: image-Un)

  have ‹pcdcl-tcore (pstate_W-of ?S) (pstate_W-of ?U)›
    using cdcl-twl-o-cdcl_W-o[OF cdcl-twl-o.backtrack-nonunit-clause[OF backtrack-nonunit-clause.hyps]]
      1 unfolding twl-struct-invs-def pcdcl-all-struct-invs-def by auto
  from pcdcl-tcore-pcdcl-all-struct-invs[OF this]
  have struct-inv-T: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of ?U)›
    using 1 unfolding twl-struct-invs-def pcdcl-all-struct-invs-def by auto
  then have n-d-L-M1: ‹no-dup (Propagated L D' # M1)›
    using struct-inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
     cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps)
  then have uL-M1: ‹undefined-lit M1 L›
    by simp

  have M1-CNot-L-D: ‹M1 ⊨as CNot (remove1-mset L D')›
    using struct-inv-T unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
     cdcl_W-restart-mset.cdcl_W-conflicting-def by (auto simp: trail.simps)

  have L-M1: ‹− L ∉ lits-of-l M1› ‹L ∉ lits-of-l M1›
    using n-d n-d-L-M1 uL-M1 by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
```

```
  have excep-M1: ‹∀ C ∈# N + U. twl-exception-inv (M1, N, U, None, NE, UE, NS, US, N0, U0,
{#}, {#}) C›
    using past unfolding past-invs.simps M by auto
  show ?case
    unfolding confl-cands-enqueued.simps Ball-def
  proof (intro allI impI)
    fix C
    assume
      C: ‹C ∈# N + add-mset ?D U› and
      LM-C: ‹Propagated L D' # M1 ⊨as CNot (clause C)›
    have ‹twl-st-inv ?U›
      using cdcl-twl-o.backtrack-nonunit-clause[OF backtrack-nonunit-clause.hyps] 1.prems
        cdcl-twl-o-twl-st-inv by blast
    then have ‹struct-wf-twl-cls ?D›
      unfolding twl-st-inv.simps by auto

    show ‹(∃ L'. L' ∈# watched C ∧ L' ∈# {#- L#}) ∨ (∃ L. (L, C) ∈# {#})›
    proof (cases ‹C = ?D›)
      case True
      then have False
        using LM-C L-uL' uL-M1 by (auto simp: true-annots-true-cls-def-iff-negation-in-model
            Decided-Propagated-in-iff-in-lits-of-l)
      then show ?thesis by fast
    next
      case False
      have struct-C: ‹struct-wf-twl-cls C›
        using twl-st-inv C False unfolding twl-st-inv.simps by auto
      then have dist-C: ‹distinct-mset (clause C)›
        by (cases C) auto

      have C: ‹C ∈# N + U›
        using C False by auto
      obtain W UW K K' where
        C-W: ‹C = TWL-Clause W UW› and
        W: ‹W = {#K, K'#}›
        using struct-C by (cases C) (auto simp: size-2-iff)

      have ‹¬M1 ⊨as CNot (clause C)›
        using confl-cands C by auto
      then have uL-C: ‹-L ∈# clause C› and neg-C: ‹∀ K ∈# clause C. -K ∈ lits-of-l (Decided L #
M1)›
        using LM-C unfolding true-annots-true-cls-def-iff-negation-in-model by auto
      have K-L: ‹K ≠ L› and K'-L: ‹K' ≠ L›
       apply (metis C-W LM-C W add-diff-cancel-right' clause.simps consistent-interp-def
          distinct-consistent-interp in-CNot-implies-uminus(2) in-diffD n-d-L-M1 uL-C
          union-single-eq-member)
        using C-W LM-C W uL-M1 by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
      have ‹-L ∈# watched C›
      proof (rule ccontr)
        assume uL-W: ‹-L ∉# watched C›
        have ‹K ≠ -L ∨ K' ≠ -L›
          using dist-C C-W W by auto
        moreover have ‹K ∉ lits-of-l M1› and ‹K' ∉ lits-of-l M1› and L-M: ‹L ∉ lits-of-l M1›
        proof -
          have f2: ‹consistent-interp (lits-of-l M1)›
            using distinct-consistent-interp n-d-L-M1 by auto
```

```
      have undef-L: ‹undefined-lit M1 L›
        using atm-lit-of-set-lits-of-l n-d-L-M1 by force
      then show ‹K ∉ lits-of-l M1›
        using f2 neg-C unfolding C-W W by (metis (no-types) C-W W add-diff-cancel-right'
            atm-of-eq-atm-of clause.simps consistent-interp-def in-diffD insertE list.simps(15)
            lits-of-insert uL-C union-single-eq-member Decided-Propagated-in-iff-in-lits-of-l)
      show ‹K' ∉ lits-of-l M1›
        using consistent-interp-def distinct-consistent-interp n-d-L-M1
        using neg-C uL-W n-d unfolding C-W W by auto
      show ‹L ∉ lits-of-l M1›
        using undef-L by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
    qed
    ultimately have ‹(−K ∈ lits-of-l M1 ∧ K' ∉ lits-of-l M1) ∨
      (−K' ∈ lits-of-l M1 ∧ K ∉ lits-of-l M1)›
      using neg-C by (auto simp: C-W W)
    moreover have ‹¬has-blit M1 (clause C) K›
      using ‹K ∉ lits-of-l M1› ‹K' ∉ lits-of-l M1› ‹L ∉ lits-of-l M1› uL-M1
        n-d-L-M1 no-dup-cons
      using uL-C neg-C n-d unfolding has-blit-def apply (auto dest!: multi-member-split
          dest!: no-dup-consistentD[OF n-d-L-M1]
          dest!: in-lits-of-l-defined-litD[of ‹−L›] simp: add-mset-eq-add-mset)
      using n-d-L-M1 no-dup-cons no-dup-consistentD by blast
    moreover have ‹¬has-blit M1 (clause C) K'›
      using ‹K' ∉ lits-of-l M1› ‹ K ∉ lits-of-l M1› ‹L ∉ lits-of-l M1› uL-M1
        n-d-L-M1 no-dup-cons no-dup-consistentD
      using uL-C neg-C n-d unfolding has-blit-def apply (auto 10 10 dest!: multi-member-split
          dest!: in-lits-of-l-defined-litD[of ‹−L›] simp: add-mset-eq-add-mset)
      using n-d-L-M1 no-dup-cons no-dup-consistentD by auto
    moreover have ‹twl-exception-inv (M1, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#}) C›
      using excep-M1 C by auto
    ultimately have ‹∀K ∈# unwatched C. −K ∈ lits-of-l M1›
      using C twl-clause.sel(1) union-single-eq-member w-q
      by (fastforce simp: twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib
          L-M)
    then show False
      using uL-W uL-C L-M K-L uL-M1 unfolding C-W W by auto
  qed
  then show ‹(∃L'. L' ∈# watched C ∧ L' ∈# {#− L#}) ∨ (∃L. (L, C) ∈# {#})›
    by auto
 qed
qed

case 2
then show ?case
 unfolding propa-cands-enqueued.simps Ball-def
proof (intro allI impI)
 fix FK C
 assume
   C: ‹C ∈# N + add-mset ?D U› and
   K: ‹FK ∈# clause C› and
   LM-C: ‹Propagated L D' # M1 ⊨as CNot (remove1-mset FK (clause C))› and
   undef: ‹undefined-lit (Propagated L D' # M1) FK›
 show ‹(∃L'. L' ∈# watched C ∧ L' ∈# {#− L#}) ∨ (∃L. (L, C) ∈# {#})›
 proof (cases ‹C = ?D›)
   case False
   then have C: ‹C ∈# N + U›
```

using *C* **by** *auto*
have *undef-M-K*: ‹*undefined-lit* (*Propagated L D # M1*) *FK*›
using *undef* **by** (*auto simp*: *defined-lit-map*)
then have ‹¬ *M1* ⊨*as CNot* (*remove1-mset FK* (*clause C*))›
using *propa-cands C K undef* **by** (*auto simp*: *defined-lit-map*)
then have ‹−*L* ∈# *clause C*› **and**
neg-C: ‹∀ *K* ∈# *remove1-mset FK* (*clause C*). −*K* ∈ *lits-of-l* (*Propagated L D # M1*)›
using *LM-C undef-M-K* **by** (*force simp*: *true-annots-true-cls-def-iff-negation-in-model*
dest: *in-diffD*)+

have *struct-C*: ‹*struct-wf-twl-cls C*›
using *twl-st-inv C* **unfolding** *twl-st-inv.simps* **by** *blast*
then have *dist-C*: ‹*distinct-mset* (*clause C*)›
by (*cases C*) *auto*

have ‹−*L* ∈# *watched C*›
proof (*rule ccontr*)
assume *uL-W*: ‹−*L* ∉# *watched C*›
then obtain *W UW K K′* **where**
C-W: ‹*C* = *TWL-Clause W UW*› **and**
W: ‹*W* = {#*K*, *K′*#}› **and**
uK-M: ‹−*K* ∈ *lits-of-l M1*›
using *struct-C neg-C* **by** (*cases C*) (*auto simp*: *size-2-iff remove1-mset-add-mset-If*
add-mset-commute split: *if-splits*)
have *FK-F*: ‹*FK* ≠ *K*›
using *uK-M undef-M-K* **unfolding** *Decided-Propagated-in-iff-in-lits-of-l* **by** *auto*
have ‹*K* ≠ −*L*›
using *uK-M uL-M1* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
moreover have ‹*K* ∉ *lits-of-l M1*›
using *neg-C uL-W n-d uK-M n-d-L-M1* **by** (*auto simp*: *lits-of-def uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus dest: *no-dup-cannot-not-lit-and-uminus*)
ultimately have ‹*K′* ∉ *lits-of-l M1*›
apply (*cases* ‹*K′* = *FK*›)
using *undef-M-K* **apply** (*force simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
using *neg-C C-W W FK-F n-d uL-W n-d-L-M1* **by** (*auto simp add*: *remove1-mset-add-mset-If*
uminus-lit-swap lits-of-def no-dup-cannot-not-lit-and-uminus
dest: *no-dup-cannot-not-lit-and-uminus*)
moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *excep-M1 C* **by** *auto*
moreover have ‹¬*has-blit M1* (*clause C*) *K*›
using ‹*K* ∉ *lits-of-l M1*› ‹*K′* ∉ *lits-of-l M1*› *uL-M1*
n-d-L-M1 no-dup-cons
using *n-d-L-M1 no-dup-cons no-dup-consistentD*
using *K in-lits-of-l-defined-litD undef*
using *neg-C n-d* **unfolding** *has-blit-def* **by** (*fastforce dest*!: *multi-member-split*
dest!: *no-dup-consistentD*[*OF n-d-L-M1*]
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹¬*has-blit M1* (*clause C*) *K′*›
using ‹*K′* ∉ *lits-of-l M1*› ‹ *K* ∉ *lits-of-l M1*› *uL-M1*
n-d-L-M1 no-dup-cons no-dup-consistentD
using *n-d-L-M1 no-dup-cons no-dup-consistentD*
using *K in-lits-of-l-defined-litD undef*
using *neg-C n-d* **unfolding** *has-blit-def* **by** (*fastforce dest*!: *multi-member-split*
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*›
using *excep-M1 C* **by** *auto*

```
    ultimately have ‹∀K ∈# unwatched C. −K ∈ lits-of-l M1›
      using uK-M
      by (auto simp: twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib)
    then show False
      using C-W uL-M1 ‹− L ∈# clause C› uL-W
      by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
  qed
  then show ‹(∃L'. L' ∈# watched C ∧ L' ∈# {#− L#}) ∨ (∃L. (L, C) ∈# {#})›
    by auto
next
  case True
  then have ‹∀K∈#remove1-mset L D'. − K ∈ lits-of-l (Propagated L D' # M1)›
    using M1-CNot-L-D by (auto simp: true-annots-true-cls-def-iff-negation-in-model)
  then have ‹∀K∈#remove1-mset L D'. defined-lit (Propagated L D' # M1) K›
    using Decided-Propagated-in-iff-in-lits-of-l by blast
  moreover have ‹defined-lit (Propagated L D' # M1) L›
    by (auto simp: defined-lit-map)
  ultimately have ‹∀K∈#D'. defined-lit (Propagated L D' # M1) K›
    by (metis in-remove1-mset-neq)
  then have ‹∀K∈#clause ?D. defined-lit (Propagated L D' # M1) K›
    using LD' ‹defined-lit (Propagated L D' # M1) L› by (auto dest: in-diffD)
  then have False
    using K undef unfolding True by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
  then show ?thesis by fast
qed
qed

case 3
then have
  2: ‹⋀L. Pair L `# {#C ∈# N + U. clauses-to-update-prop {#} M1 (L, C)#} = {#}› and
  3: ‹⋀L C. C ∈# N + U ⟹ L ∈# watched C ⟹ − L ∈ lits-of-l M1 ⟹
    ¬ has-blit M1 (clause C) L ⟹ (L, C) ∉# {#} ⟹ L ∈# {#}›
  using w-q unfolding clauses-to-update-inv.simps by auto
have ‹i = count-decided M1›
  using decomp lev-K n-d by (auto dest!: get-all-ann-decomposition-exists-prepend
      simp: get-level-append-if get-level-cons-if
      split: if-splits)
then have lev-L'-M1: ‹get-level (Propagated L D' # M1) L' = count-decided M1›
  using decomp lev-L' n-d by (auto dest!: get-all-ann-decomposition-exists-prepend
      simp: get-level-append-if get-level-cons-if
      split: if-splits)
have blit-L': ‹has-blit (Propagated L D' # M1) (add-mset L (add-mset L' (D' − {#L, L'#}))) L'›
  unfolding has-blit-def
  by (rule-tac x=L in exI) (auto simp: lev-L'-M1)
show ?case
proof (induction rule: clauses-to-update-inv-cases)
  case (WS-nempty L C)
  then show ?case by simp
next
  case (WS-empty K')

  show ?case
    using 2[of K] 3 n-d-L-M1 L-M1 blit-L'
    apply (simp only: filter-mset-empty-conv Ball-def image-mset-is-empty-iff)
    by (fastforce simp add: clauses-to-update-prop.simps )
next
```

```
    case (Q K' C)
    then show ?case
      using 3[of C K'] uL-M1 blit-L' n-d-L-M1 has-blit-Cons
      by (fastforce simp add: clauses-to-update-prop.simps
          add-mset-eq-add-mset Decided-Propagated-in-iff-in-lits-of-l)
  qed
qed

lemma no-dup-append-decided-Cons-lev:
  assumes ‹no-dup (M2 @ Decided K # M1)›
  shows ‹count-decided M1 = get-level (M2 @ Decided K # M1) K - 1›
proof -
  have ‹undefined-lit (M2 @ M1) K›
    by (rule no-dup-append-in-atm-notin[of -
          ‹[Decided K]›])
      (use assms in auto)
  then show ?thesis
    by (auto)
qed
```

The Strategy

```
lemma no-literals-to-update-no-cp:
  assumes
    WS: ‹clauses-to-update S = {#}› and Q: ‹literals-to-update S = {#}› and
    twl: ‹twl-struct-invs S›
  shows
    ‹no-step cdcl-propagate (pstate_W-of S)› and
    ‹no-step cdcl-conflict (pstate_W-of S)›
proof -
  obtain M N U NE UE NS US N0 U0 D where
      S: ‹S = (M, N, U, D, NE, UE, NS, US, N0, U0, {#}, {#})›
    using WS Q by (cases S) auto

  {
    assume confl: ‹get-conflict S = None›
    then have S: ‹S = (M, N, U, None, NE, UE, NS, US, N0, U0, {#}, {#})›
      using WS Q S by auto

    have twl-st-inv: ‹twl-st-inv S› and
      struct-inv: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)› and
      excep: ‹twl-st-exception-inv S› and
      confl-cands: ‹confl-cands-enqueued S› and
      propa-cands: ‹propa-cands-enqueued S›
      using twl unfolding twl-struct-invs-def pcdcl-all-struct-invs-def by auto
    have n-d: ‹no-dup M›
      using struct-inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
        cdcl_W-restart-mset.cdcl_W-M-level-inv-def by (auto simp: trail.simps S)
    then have L-uL: ‹L ∈ lits-of-l M ⟹ -L ∉ lits-of-l M› for L
      using consistent-interp-def distinct-consistent-interp by blast
    have no-confl: ‹∀ C ∈# N + U. ¬M⊨as CNot (clause C)›
      using confl-cands unfolding S by auto
    then have ns-confl: ‹no-step cdcl-conflict (pstate_W-of S)›
      by (auto simp: S trail.simps clauses-def cdcl-conflict.simps)

    have ns-propa: ‹no-step cdcl-propagate (pstate_W-of S)›
```

```
    proof (rule ccontr)
      assume ‹¬ ?thesis›
      then obtain C L where
        C: ‹C ∈# clause `# (N + U)› and
        L: ‹L ∈# C› and
        M: ‹M ⊨as CNot (remove1-mset L C)› and
        undef: ‹undefined-lit M L›
        by (auto simp: S trail.simps clauses-def cdcl-propagate.simps)
      then show False
        using propa-cands L M undef by (auto simp: S)
    qed
    note ns-confl ns-propa
  }
  moreover {
    assume ‹get-conflict S ≠ None›
    then have ‹no-step cdcl-propagate (pstate_W-of S)›
      ‹no-step cdcl-conflict (pstate_W-of S)›
      by (auto simp: S conflicting.simps cdcl-conflict.simps cdcl-propagate.simps)
  }
  ultimately show ‹no-step cdcl-propagate (pstate_W-of S)›
      ‹no-step cdcl-conflict (pstate_W-of S)›
    by blast+
qed
```

When popping a literal from *literals-to-update* to the *clauses-to-update*, we do not do any transition in the abstract transition system. Therefore, we use *rtranclp* or a case distinction.

```
lemma cdcl-twl-stgy-cdcl_W-stgy2:
  assumes ‹cdcl-twl-stgy S T› and twl: ‹twl-struct-invs S›
  shows ‹pcdcl-tcore-stgy (pstate_W-of S) (pstate_W-of T) ∨
    (pstate_W-of S = pstate_W-of T ∧ (literals-to-update-measure T, literals-to-update-measure S)
    ∈ lexn less-than 2)›
  using assms(1)
proof (induction rule: cdcl-twl-stgy.induct)
  case (cp S')
  then show ?case
    using twl cdcl-twl-cp-cdcl_W-stgy[of S S'] by (metis (full-types)
    cdcl-twl-cp-twl-struct-invs pcdcl-all-struct-invs-def
    pcdcl-core-stgy.intros(1) pcdcl-core-stgy.intros(2) pcdcl-tcore-stgy.intros(1) state_W-of-def
    twl-cp-propagate-or-conflict twl-struct-invs-def)
next
  case (other' S') note o = this(1)
  have wq: ‹clauses-to-update S = {#}› and p: ‹literals-to-update S = {#}›
    using o by (cases rule: cdcl-twl-o.cases; auto)+
  have ‹pcdcl-tcore-stgy (pstate_W-of S) (pstate_W-of S')›
    using no-literals-to-update-no-cp[OF wq p twl] cdcl-twl-o-cdcl_W-o[of S S', OF o] twl
      pcdcl-tcore-nocp-pcdcl-tcore-stgy[of ‹pstate_W-of S› ‹pstate_W-of S'›]
   by (auto simp: twl-struct-invs-def pcdcl-all-struct-invs-def)

  then show ?case
    by auto
qed

lemma cdcl-twl-stgy-cdcl_W-stgy:
  assumes ‹cdcl-twl-stgy S T› and twl: ‹twl-struct-invs S›
  shows ‹pcdcl-tcore-stgy** (pstate_W-of S) (pstate_W-of T)›
  using cdcl-twl-stgy-cdcl_W-stgy2[OF assms] by auto
```

lemma *cdcl-twl-o-twl-struct-invs*:
  assumes
    cdcl: ‹*cdcl-twl-o S T*› **and**
    twl: ‹*twl-struct-invs S*›
  shows ‹*twl-struct-invs T*›
proof −
  have $cdcl_W$: ‹*pcdcl-tcore* (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*)›
    by (*metis cdcl cdcl-twl-o-cdcl*$_W$ *-o pcdcl-all-struct-invs-def state*$_W$ *-of-def twl twl-struct-invs-def*)

  have *wq*: ‹*clauses-to-update S* = {#}› **and** *p*: ‹*literals-to-update S* = {#}›
    using *cdcl* **by** (*cases rule*: *cdcl-twl-o.cases*; *auto*)+

  have *struct-invs*: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of T*)›
    using $cdcl_W$ *pcdcl-tcore-pcdcl-all-struct-invs twl twl-struct-invs-def* **by** *blast*

 show *?thesis*
    unfolding *twl-struct-invs-def*
    apply (*intro conjI*)
    subgoal by (*use cdcl cdcl-twl-o-twl-st-inv twl* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-valid* **in** ‹*blast*; *fail*›)
    subgoal by (*rule struct-invs*)
    subgoal
      by (*metis* $cdcl_W$ *no-literals-to-update-no-cp*(*1*) *no-literals-to-update-no-cp*(*2*) *p*
        pcdcl-tcore-nocp-pcdcl-tcore-stgy pcdcl-tcore-stgy-no-smaller-propa state$_W$ *-of-def*
        twl twl-struct-invs-def wq)
    subgoal by (*use cdcl cdcl-twl-o-twl-st-exception-inv twl* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-no-duplicate-queued* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-distinct-queued* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-confl-cands-enqueued twl twl-struct-invs-def* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-propa-cands-enqueued twl twl-struct-invs-def* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl twl cdcl-twl-o-conflict-None-queue* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl twl-o-clauses-to-update twl* **in** *blast*)
    subgoal by (*use cdcl cdcl-twl-o-past-invs twl twl-struct-invs-def* **in** *blast*)
    done
qed

lemma *cdcl-twl-stgy-twl-struct-invs*:
  assumes
    cdcl: ‹*cdcl-twl-stgy S T*› **and**
    twl: ‹*twl-struct-invs S*›
  shows ‹*twl-struct-invs T*›
  using *cdcl* **by** (*induction rule*: *cdcl-twl-stgy.induct*)
    (*simp-all add*: *cdcl-twl-cp-twl-struct-invs cdcl-twl-o-twl-struct-invs twl*)

lemma *rtranclp-cdcl-twl-stgy-twl-struct-invs*:
  assumes
    cdcl: ‹*cdcl-twl-stgy*** *S T*› **and**
    twl: ‹*twl-struct-invs S*›
  shows ‹*twl-struct-invs T*›
  using *cdcl* **by** (*induction rule*: *rtranclp-induct*) (*simp-all add*: *cdcl-twl-stgy-twl-struct-invs twl*)

lemma *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*:
  assumes ‹*cdcl-twl-stgy*** *S T*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹*pcdcl-tcore-stgy*** (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*)›
  using *assms* **by** (*induction rule*: *rtranclp-induct*)

(*auto dest*!: *cdcl-twl-stgy-cdcl$_W$-stgy intro*: *rtranclp-cdcl-twl-stgy-twl-struct-invs*)

lemma *no-step-cdcl-twl-cp-no-step-cdcl$_W$-cp*:
assumes *ns-cp*: ‹*no-step cdcl-twl-cp S*› **and** *twl*: ‹*twl-struct-invs S*›
shows ‹*literals-to-update S* = {#} ∧ *clauses-to-update S* = {#}›
proof (*cases* ‹*get-conflict S*›)
case (*Some a*)
then show *?thesis*
using *twl* **unfolding** *twl-struct-invs-def* **by** *simp*
next
case *None* **note** *confl* = *this*(*1*)
then obtain *M N U UE NE NS US N0 U0 WS Q* **where** *S*: ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases S*) *auto*
have *valid*: ‹*valid-enqueued S*› **and** *twl*: ‹*twl-st-inv S*›
using *twl* **unfolding** *twl-struct-invs-def* **by** *fast*+
have *wq*: ‹*clauses-to-update S* = {#}›
proof (*rule ccontr*)
assume ‹*clauses-to-update S* ≠ {#}›
then obtain *L C WS′* **where** *LC*: ‹(*L*, *C*) ∈# *clauses-to-update S*› **and**
WS′: ‹*WS* = *add-mset* (*L*, *C*) *WS′*›
by (*cases WS*) (*auto simp*: *S*)

have *C-N-U*: ‹*C* ∈# *N* + *U*› **and** *L-C*: ‹*L* ∈# *watched C*› **and** *uL-M*: ‹− *L* ∈ *lits-of-l M*›
using *valid LC* **unfolding** *S* **by** *auto*

have ‹*struct-wf-twl-cls C*›
using *C-N-U twl* **unfolding** *S* **by** (*auto simp*: *twl-st-inv.simps*)
then obtain *L′* **where** *watched*: ‹*watched C* = {#*L*, *L′*#}›
using *L-C* **by** (*cases C*) (*auto simp*: *size-2-iff*)
then have ‹*L* ∈# *clause C*›
by (*cases C*) *auto*
then have *L′-M*: ‹*L′* ∉ *lits-of-l M*›
using *cdcl-twl-cp.delete-from-working*[*of L′ C M N U NE UE NS US N0 U0 L WS′ Q*] *watched*
ns-cp **unfolding** *S WS′* **by** (*cases C*) *auto*
then have ‹*undefined-lit M L′* ∨ −*L′* ∈ *lits-of-l M*›
using *Decided-Propagated-in-iff-in-lits-of-l* **by** *blast*
then have ‹¬ (∀ *L* ∈# *unwatched C*. −*L* ∈ *lits-of-l M*)›
using *cdcl-twl-cp.conflict*[*of C L L′ M N U NE UE NS US N0 U0 WS′ Q*]
cdcl-twl-cp.propagate[*of C L L′ M N U NE UE NS US N0 U0 WS′ Q*] *watched*
ns-cp **unfolding** *S WS′* **by** *fast*
then obtain *K* **where** *K*: ‹*K* ∈# *unwatched C*› **and** *uK-M*: ‹−*K* ∉ *lits-of-l M*›
by *auto*
then have *undef-K-K-M*: ‹*undefined-lit M K* ∨ *K* ∈ *lits-of-l M*›
using *Decided-Propagated-in-iff-in-lits-of-l* **by** *blast*
define *NU* **where** ‹*NU* = (*if C* ∈# *N then* (*add-mset* (*update-clause C L K*) (*remove1-mset C N*), *U*)
else (*N*, *add-mset* (*update-clause C L K*) (*remove1-mset C U*)))›
have *upd*: ‹*update-clauses* (*N*, *U*) *C L K NU*›
using *C-N-U* **unfolding** *NU-def* **by** (*auto simp*: *update-clauses.intros*)
have *NU*: ‹*NU* = (*fst NU*, *snd NU*)›
by *simp*
show *False*
using *cdcl-twl-cp.update-clause*[*of C L L′ M K N U* ‹*fst NU*› ‹*snd NU*› *NE UE NS US N0 U0 WS′ Q*]
watched uL-M L′-M K undef-K-K-M upd ns-cp **unfolding** *S WS′* **by** *simp*

```
  qed
  then have p: ‹literals-to-update S = {#}›
    using cdcl-twl-cp.pop[of M N U NE UE] S ns-cp by (cases ‹Q›) fastforce+
  show ?thesis using wq p by blast
qed

lemma no-step-cdcl-twl-o-no-step-cdcl_W-o:
  assumes
    ns-o: ‹no-step cdcl-twl-o S› and
    twl: ‹twl-struct-invs S› and
    p: ‹literals-to-update S = {#}› and
    w-q: ‹clauses-to-update S = {#}›
  shows ‹no-step cdcl-decide (pstate_W-of S) ∧ no-step cdcl-skip (pstate_W-of S) ∧
     no-step cdcl-resolve (pstate_W-of S) ∧ no-step cdcl-backtrack (pstate_W-of S)›
proof (rule ccontr)
  obtain M N U D NE UE NS US N0 U0 where S: ‹S = (M, N, U, D, NE, UE, NS, US, N0, U0,
{#}, {#})›
    using p w-q by (cases S) auto
  assume ‹¬ ?thesis›
  then consider
    (decide) T where ‹cdcl-decide (pstate_W-of S) T› |
    (skip) T where ‹cdcl-skip (pstate_W-of S) T› |
    (resolve) T where ‹cdcl-resolve (pstate_W-of S) T› |
    (backtrack) T where ‹cdcl-backtrack (pstate_W-of S) T›
    by blast
  then show False
  proof (cases)
    case (decide T)
    then show ?thesis
      apply (cases rule: cdcl-decide.cases)
      subgoal for M' L' N' NE' NS' U' UE' US'
        using cdcl-twl-o.decide[of M L' N NE NS N0 U UE US U0] ns-o unfolding S
        by (auto simp: cdcl_W-restart-mset-state)
      done
  next
    case (skip T)
    then show ?thesis
      apply (cases rule: cdcl-skip.cases)
      subgoal for L E D M'
       using cdcl-twl-o.skip[of L E D M' N U NE UE NS US N0 U0] ns-o unfolding S
       by (auto simp: cdcl_W-restart-mset-state)
      done
  next
    case (resolve T)
    then show ?thesis
      apply (cases rule: cdcl-resolve.cases)
      subgoal for L' D E M'
       using cdcl-twl-o.resolve[of L' D E M' N U NE UE NS US N0 U0] ns-o unfolding S
       by (auto simp: cdcl_W-restart-mset-state)
      done
  next
    case (backtrack T)
    then show ?thesis
      apply (cases rule: cdcl-backtrack.cases)
      subgoal for K M1 M2 M L D' i C N' U' NE UE NS US N0 U0
        using cdcl-twl-o.backtrack-unit-clause[of L ‹add-mset L C› K M1 M2 M
```

```
      ‹add-mset L D'› i N U NE UE NS US N0 U0]
    using cdcl-twl-o.backtrack-nonunit-clause[of L ‹add-mset L C› K M1 M2 M ‹add-mset L D'›
      i N U NE UE NS US N0 U0] ns-o get-maximum-level-exists-lit-of-max-level[of D' M]
    unfolding S
    by (cases ‹D' ≠ {#}›) auto
  done
 qed
qed

lemma no-step-cdcl-twl-stgy-no-step-cdcl_W-stgy:
 assumes ns: ‹no-step cdcl-twl-stgy S› and twl: ‹twl-struct-invs S›
 shows ‹no-step pcdcl-core-stgy (pstate_W-of S)›
proof −
 have ns-cp: ‹no-step cdcl-twl-cp S› and ns-o: ‹no-step cdcl-twl-o S›
  using ns by (auto simp: cdcl-twl-stgy.simps)
 then have w-q: ‹clauses-to-update S = {#}› and p: ‹literals-to-update S = {#}›
  using ns-cp no-step-cdcl-twl-cp-no-step-cdcl_W-cp twl by blast+
 then have
  ‹no-step cdcl-propagate (pstate_W-of S)› and
  ‹no-step cdcl-conflict (pstate_W-of S)›
  using no-literals-to-update-no-cp twl by blast+
 moreover have ‹no-step cdcl-decide (pstate_W-of S) ∧ no-step cdcl-skip (pstate_W-of S) ∧
   no-step cdcl-resolve (pstate_W-of S) ∧ no-step cdcl-backtrack (pstate_W-of S)›
  using w-q p ns-o no-step-cdcl-twl-o-no-step-cdcl_W-o twl by blast
 ultimately show ?thesis
  by (auto simp: cdcl_W-restart-mset.cdcl_W-stgy.simps pcdcl-core-stgy.simps)
qed
```

This where things get different from the direct inheritance from CDCL. Originally, we had the following theorem:

```
lemma full-cdcl-twl-stgy-cdcl_W-stgy:
 assumes ‹full cdcl-twl-stgy S T› and twl: ‹twl-struct-invs S›
 shows ‹full cdcl_W-restart-mset.cdcl_W-stgy (state_W-of S) (state_W-of T)›
 oops
```

However, now we have to split the steps part from the end part.

```
lemma full-cdcl-twl-stgy-cdcl_W-stgy:
 assumes ‹full cdcl-twl-stgy S T› and twl: ‹twl-struct-invs S›
 shows ‹full2 pcdcl-tcore-stgy pcdcl-core (pstate_W-of S) (pstate_W-of T)›
 using assms
 unfolding full2-def full-def
 by (meson no-step-cdcl-twl-stgy-no-step-cdcl_W-stgy pcdcl-core.simps pcdcl-core-stgy.simps
  rtranclp-cdcl-twl-stgy-cdcl_W-stgy rtranclp-cdcl-twl-stgy-twl-struct-invs)



definition init-state-twl where
 ‹init-state-twl N ≡ ([], N, {#}, None, {#}, {#}, {#}, {#}, {#}, {#}, {#}, {#})›
lemma
 assumes
  struct: ‹∀ C ∈# N. struct-wf-twl-cls C› and
  tauto: ‹∀ C ∈# N. ¬tautology (clause C)›
 shows
  twl-stgy-invs-init-state-twl: ‹twl-stgy-invs (init-state-twl N)› and
  twl-struct-invs-init-state-twl: ‹twl-struct-invs (init-state-twl N)›
```

proof −
have [*simp*]: ‹*twl-lazy-update* [] *C*› ‹*watched-literals-false-of-max-level* [] *C*›
‹*twl-exception-inv* ([], *N*, {#}, *None*, {#}, {#}, {#}, {#}, {#}, {#}, {#}, {#}) *C*› **for** *C*
by (*cases C*; *solves* ‹*auto simp*: *twl-exception-inv.simps*›)+

have *size-C*: ‹*size* (*clause C*) ≥ *2*› **if** ‹*C* ∈# *N*› **for** *C*
proof −
have ‹*struct-wf-twl-cls C*›
using *that struct* **by** *auto*
then show *?thesis* **by** (*cases C*) *auto*
qed
have
[*simp*]: ‹*clause C* ≠ {#}› (**is** *?G1*) **and**
[*simp*]: ‹*remove1-mset L* (*clause C*) ≠ {#}› (**is** *?G2*) **if** ‹*C* ∈# *N*› **for** *C L*
by (*rule size-neq-size-imp-neq*[*of* - ‹{#}›]; *use size-C*[*OF that*] **in**
‹*auto simp*: *remove1-mset-empty-iff union-is-single*›)+

have ‹*distinct-mset* (*clause C*)› **if** ‹*C* ∈# *N*› **for** *C*
using *struct that* **by** (*cases C*) (*auto*)
then have *dist*: ‹*distinct-mset-mset* (*clause* '# *N*)›
by (*auto simp*: *distinct-mset-set-def*)
then have [*simp*]: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ([], *clause* '# *N*, {#}, *None*)›
using *struct* **unfolding** *init-state.simps*[*symmetric*] $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-S0*
by (*simp only*: $cdcl_W$*-restart-mset.no-strange-atm-S0*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-S0-*$cdcl_W$*-restart*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-S0-*$cdcl_W$*-restart*[*OF dist*]
$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-S0-*$cdcl_W$*-restart dist*
$cdcl_W$*-restart-mset.all-invariant-S0-*$cdcl_W$*-restart*
$cdcl_W$*-restart-mset.all-invariant-S0-*$cdcl_W$*-restart*(*1*)[*OF dist*], *simp*)

have [*simp*]: ‹$cdcl_W$*-restart-mset.no-smaller-propa* ([], *clause* '# *N*, {#}, *None*)›
by(*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def* $cdcl_W$*-restart-mset-state*)
have [*simp*]: ‹*entailed-clss-inv* ([], *clauses N*, {#}, *None*, {#}, {#}, {#}, {#}, {#}, {#})›
‹*psubsumed-invs* ([], *clauses N*, {#}, *None*, {#}, {#}, {#}, {#}, {#}, {#})›
by (*auto simp*: *entailed-clss-inv-def psubsumed-invs-def*)
show *stgy-invs*: ‹*twl-stgy-invs* (*init-state-twl N*)›
by (*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
$cdcl_W$*-restart-mset-state* $cdcl_W$*-restart-mset.no-smaller-confl-def init-state-twl-def*)
show ‹*twl-struct-invs* (*init-state-twl N*)›
using *struct tauto*
by (*auto simp*: *twl-struct-invs-def twl-st-inv.simps clauses-to-update-prop.simps*
past-invs.simps $cdcl_W$*-restart-mset-state init-state-twl-def clauses0-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def pcdcl-all-struct-invs-def*)
qed

lemma *cdcl-twl-o-*$cdcl_W$*-o-stgy*:
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
inv: ‹*twl-struct-invs S*›
shows ‹*pcdcl-tcore-stgy* ($pstate_W$*-of S*) ($pstate_W$*-of T*)›
using *cdcl inv*
proof (*induction rule*: *cdcl-twl-o.induct*)
case (*decide M L N NE NS N0 U UE US U0*) **note** *undef* = *this*(*1*) **and** *atm* = *this*(*2*) **and** *inv* =

this(*3*)
have *0*: ‹*cdcl-decide* (*pstate*$_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))
(*pstate*$_W$*-of* (*Decided L # M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#−*L*#}))›
unfolding *pstate*$_W$*-of.simps*
apply (*rule cdcl-decide.intros*)
using *undef* **apply** (*simp add*: *trail.simps*; *fail*)
using *atm* **apply** (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
done
then show *dec*: *?case*
using *no-literals-to-update-no-cp*[*of* ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›] *inv*
by (*auto dest*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-o.intros pcdcl-core.intros simp*: *pcdcl-tcore-stgy.simps*
pcdcl-core-stgy.simps)
next
case (*skip L D C′ M N U NE UE NS US N0 U0*) **note** *LD* = *this*(*1*) **and** *D* = *this*(*2*) **and** *inv* = *this*(*3*)
have *skip′*: ‹*cdcl-skip* (*pstate*$_W$*-of* (*Propagated L C′ # M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))
(*pstate*$_W$*-of* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›
unfolding *pstate*$_W$*-of.simps*
apply (*rule cdcl-skip.intros*)
using *LD* **apply** (*simp*; *fail*)
using *D* **apply** (*simp*; *fail*)
done
show *?case*
by (*rule pcdcl-tcore-stgy.intros*(*1*), *rule pcdcl-core-stgy.intros*(*4*)) (*rule skip′*)
next
case (*resolve L D C M N U NE UE NS US N0 U0*) **note** *LD* = *this*(*1*) **and** *lev* = *this*(*2*) **and** *inv* = *this*(*3*)
have ‹∀ *La mark a b*. *a* @ *Propagated La mark* # *b* = *Propagated L C* # *M* ⟶
b ⊨*as CNot* (*remove1-mset La mark*) ∧ *La* ∈# *mark*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-conflicting-def twl-struct-invs-def pcdcl-all-struct-invs-def*
by (*auto simp*: *trail.simps*)
then have *LC*: ‹*L* ∈# *C*›
by *blast*
have *resolve*: ‹*cdcl-resolve* (*pstate*$_W$*-of* (*Propagated L C* # *M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))
(*pstate*$_W$*-of*
(*M*, *N*, *U*, *Some* (*remove1-mset* (− *L*) *D* ∪# *remove1-mset L C*), *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›
unfolding *pstate*$_W$*-of.simps*
apply (*rule cdcl-resolve.intros*)
using *LD* **apply** (*simp*; *fail*)
using *lev* **apply** (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
using *LC* **apply** (*simp add*: *trail.simps*; *fail*)
done
show *?case*
by (*rule pcdcl-tcore-stgy.intros*(*1*), *rule pcdcl-core-stgy.intros*(*5*))
(*rule resolve*)
next
case (*backtrack-unit-clause L D K M1 M2 M D′ i N U NE UE NS US N0 U0*) **note** *L-D* = *this*(*1*)
and
decomp = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *max-D′-L* = *this*(*4*) **and** *lev-D* = *this*(*5*) **and**
lev-K = *this*(*6*) **and** *D′-D* = *this*(*8*) **and** *NU-D′* = *this*(*9*) **and**
D′[*simp*] = *this*(*7*) **and** *inv* = *this*(*10*)
have *D*: ‹*D* = *add-mset L* (*remove1-mset L D*)›

using *L-D* **by** *auto*
have *bt*: ‹*cdcl-backtrack-unit* (*pstate*$_W$ *-of* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))
(*pstate*$_W$ *-of*
(*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#− *L*#}))›
unfolding *pstate*$_W$ *-of.simps*
apply (*subst D*)
apply (*rule cdcl-backtrack-unit.intros*)
using *backtrack-unit-clause* **by** *auto*
then show *?case*
by (*rule pcdcl-tcore-stgy.intros*(*4*))
next
case (*backtrack-nonunit-clause L D K M1 M2 M D′ i N U NE UE NS US N0 U0 L′*) **note** *LD* = *this*(*1*) **and**
decomp = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *max-lev* = *this*(*4*) **and** *i* = *this*(*5*) **and** *lev-K* = *this*(*6*)
and *D′-D* = *this*(*8*) **and** *NU-D′* = *this*(*9*) **and** *L-D′* = *this*(*10*) **and** *L′* = *this*(*11*−*12*) **and**
inv = *this*(*13*)
let *?S* = ‹*state*$_W$ *-of* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
let *?T* = ‹*state*$_W$ *-of* (*Propagated L D* # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#*L*#})›
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-all-struct-inv-def*
cdcl$_W$ *-restart-mset.cdcl*$_W$ *-M-level-inv-def twl-struct-invs-def pcdcl-all-struct-invs-def*
by (*simp add*: *cdcl*$_W$ *-restart-mset-state*)
have ‹*undefined-lit M1 L*›
apply (*rule cdcl*$_W$ *-restart-mset.backtrack-lit-skiped*[*of ?S - K - M2 i*])
subgoal
using *lev-L inv* **unfolding** *cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-all-struct-inv-def*
cdcl$_W$ *-restart-mset.cdcl*$_W$ *-M-level-inv-def*
by (*simp add*: *cdcl*$_W$ *-restart-mset-state*; *fail*)
subgoal using *decomp* **by** (*simp add*: *trail.simps*; *fail*)
subgoal using *lev-L inv*
unfolding *cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-all-struct-inv-def cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-M-level-inv-def*
twl-struct-invs-def pcdcl-all-struct-invs-def
by (*simp add*: *cdcl*$_W$ *-restart-mset-state*; *fail*)
subgoal using *lev-K* **by** (*simp add*: *trail.simps*; *fail*)
done
obtain *M3* **where** *M3*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** (*blast dest*!: *get-all-ann-decomposition-exists-prepend*)

have ‹*undefined-lit* (*M3* @ *M2*) *K*›
using *n-d* **unfolding** *M3* **by** (*auto simp*: *lits-of-def*)
then have *count-M1*: ‹*count-decided M1* = *i*›
using *lev-K* **unfolding** *M3* **by** (*auto simp*: *image-Un*)
have ‹*L* ≠ *L′*›
using *L′ lev-L lev-K count-decided-ge-get-level*[*of M K*] *L′* **by** *auto*
then have *D*: ‹*add-mset L* (*add-mset L′* (*D′* − {#*L*, *L′*#})) = *D′*›
using *L′ L-D′*
by (*metis add-mset-diff-bothsides diff-single-eq-union insert-noteq-member mset-add*)
then have *D4*: ‹({#*L*, *L′*#} + (*D′* − {#*L*, *L′*#})) = *add-mset L* (*add-mset L′* (*D′* − {#*L*, *L′*#}))›
by *auto*
have *D′*: ‹*remove1-mset L D′* = *add-mset L′* (*D′* − {#*L*, *L′*#})›
by (*subst D*[*symmetric*]) *auto*
have *D″*: ‹*D* = *add-mset L* (*remove1-mset L D*)›
using *L-D′ D′-D* **by** *auto*
show *?case*

apply (*subst D*[*symmetric*])
apply (*subst D''*)
apply (*rule pcdcl-tcore-stgy.intros*(*1*), *rule pcdcl-core-stgy.intros*(*6*))
unfolding *pstate*$_W$*-of.simps image-mset-add-mset clause.simps D4*
apply (*rule cdcl-backtrack.intros*[*of K M1 M2 - - - i*])
subgoal using *decomp* **by** (*simp add*: *trail.simps*)
subgoal using *lev-L* **by** (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
subgoal using *max-lev L-D'* **by** (*simp add*: *cdcl*$_W$*-restart-mset-state get-maximum-level-add-mset D*)
subgoal using *i* **by** (*simp add*: *cdcl*$_W$*-restart-mset-state D'*)
subgoal using *lev-K i* **unfolding** *D'* **by** (*simp add*: *trail.simps*)
subgoal using *D'-D* **by** (*metis D' mset-le-subtract*)
subgoal using *NU-D' L-D'* **by** (*simp add*: *mset-le-subtract clauses-def ac-simps D*)
done
qed

lemma *pcdcl-tcore-stgy-conflict-non-zero-unless-level-0*:
‹*pcdcl-tcore-stgy S T* ⟹ *cdcl*$_W$*-restart-mset.conflict-non-zero-unless-level-0* (*state-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant* (*state-of S*) ⟹
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0* (*state-of T*)›
apply (*induction rule*: *pcdcl-tcore-stgy.induct*)
subgoal
using *pcdcl-core-is-cdcl*[*of S T*]
cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-conflict-non-zero-unless-level-0*[*of* ‹*state-of S*› ‹*state-of T*›]
by (*auto dest*!: *pcdcl-core-stgy-pcdcl cdcl*$_W$*-restart-mset.cdcl*$_W$*-cdcl*$_W$*-restart*)
subgoal
by (*auto simp*: *cdcl-subsumed.simps cdcl*$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)
subgoal
by (*auto simp*: *cdcl-flush-unit.simps cdcl*$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)
subgoal
by (*metis cdcl*$_W$*-restart-mset.cdcl*$_W$*-restart-conflict-non-zero-unless-level-0*
cdcl$_W$*-restart-mset.cdcl*$_W$*-cdcl*$_W$*-restart cdcl-backtrack-unit-is-backtrack*
cdcl-flush-unit-unchanged pcdcl-core.intros(*6*) *pcdcl-core-is-cdcl*)
done

lemma *cdcl-twl-o-twl-stgy-invs*:
‹*cdcl-twl-o S T* ⟹ *twl-struct-invs S* ⟹ *twl-stgy-invs S* ⟹ *twl-stgy-invs T*›
using *cdcl-twl-o-cdcl*$_W$*-o-stgy*[*of S T*]
pcdcl-tcore-stgy-conflict-non-zero-unless-level-0[*of* ‹*pstate*$_W$*-of S*› ‹*pstate*$_W$*-of T*›]
by (*auto simp*: *twl-struct-invs-def twl-stgy-invs-def pcdcl-all-struct-invs-def*
intro!: *rtranclp-pcdcl-stgy-stgy-invariant*[*of* ‹*pstate*$_W$*-of S*› ‹*pstate*$_W$*-of T*›]
dest: *pcdcl-tcore-stgy-pcdcl-stgy'*)

Well-foundedness **lemma** *wf-cdcl*$_W$*-stgy-state*$_W$*-of*:
‹*wf* {(*T*, *S*). *pcdcl-all-struct-invs* (*pstate*$_W$*-of S*) ∧ *pcdcl-tcore* (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)}›
using *wf-if-measure-f*[*OF wf-pcdcl-tcore*, *of pstate*$_W$*-of*] **by** *simp*

lemma *wf-cdcl-twl-cp*:
‹*wf* {(*T*, *S*). *twl-struct-invs S* ∧ *cdcl-twl-cp S T*}› (**is** ‹*wf ?TWL*›)
proof −
let *?CDCL* = ‹{(*T*, *S*). *pcdcl-all-struct-invs* (*pstate*$_W$*-of S*) ∧
pcdcl-tcore (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)}›
let *?P* = ‹{(*T*, *S*). *pstate*$_W$*-of S* = *pstate*$_W$*-of T* ∧
(*literals-to-update-measure T*, *literals-to-update-measure S*) ∈ *lexn less-than 2*}›

have *wf-p-m*:

```
    ‹wf {(T, S). (literals-to-update-measure T, literals-to-update-measure S) ∈ lexn less-than 2}›
    using wf-if-measure-f[of ‹lexn less-than 2› literals-to-update-measure] by (auto simp: wf-lexn)
  have ‹wf ?CDCL›
    by (rule wf-subset[OF wf-cdcl_W-stgy-state_W-of])
      (auto simp: twl-struct-invs-def)
  moreover have ‹wf ?P›
    by (rule wf-subset[OF wf-p-m]) auto
  moreover have ‹?CDCL O ?P ⊆ ?CDCL› by auto
  ultimately have ‹wf (?CDCL ∪ ?P)›
    by (rule wf-union-compatible)

  moreover have ‹?TWL ⊆ ?CDCL ∪ ?P›
  proof
    fix x
    assume x-TWL: ‹x ∈ ?TWL›
    then obtain S T where x: ‹x = (T, S)› by auto

    have twl: ‹twl-struct-invs S› and cdcl: ‹cdcl-twl-cp S T›
      using x-TWL x by auto
    have ‹pcdcl-all-struct-invs (pstate_W-of S)›
      using twl by (auto simp: twl-struct-invs-def)
    moreover have ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)›
      using twl by (auto simp: twl-struct-invs-def pcdcl-all-struct-invs-def)
    moreover have ‹pcdcl-tcore (pstate_W-of S) (pstate_W-of T) ∨
      (pstate_W-of S = pstate_W-of T ∧
        (literals-to-update-measure T, literals-to-update-measure S) ∈ lexn less-than 2)›
      using calculation cdcl cdcl-twl-cp-cdcl_W-stgy[of S T] twl apply (auto simp: pcdcl-tcore.intros
          pcdcl-tcore-stgy.simps twl-cp-propagate-or-conflict pcdcl-core.intros
        dest: pcdcl-core-stgy-pcdcl)
     using calculation pcdcl-core.intros(1,2) pcdcl-tcore.intros(1) twl-cp-propagate-or-conflict
       twl-struct-invs-def by blast

    ultimately show ‹x ∈ ?CDCL ∪ ?P›
      unfolding x by auto
  qed
  ultimately show ?thesis
    using wf-subset[of ‹?CDCL ∪ ?P›] by blast
qed

lemma tranclp-wf-cdcl-twl-cp:
  ‹wf {(T, S). twl-struct-invs S ∧ cdcl-twl-cp^++ S T}›
proof −
  have H: ‹{(T, S). twl-struct-invs S ∧ cdcl-twl-cp^++ S T} ⊆
    {(T, S). twl-struct-invs S ∧ cdcl-twl-cp S T}^+›
  proof −
    { fix T S :: ‹'v twl-st›
      assume ‹cdcl-twl-cp^++ S T› ‹twl-struct-invs S›
      then have ‹(T, S) ∈ {(T, S). twl-struct-invs S ∧ cdcl-twl-cp S T}^+› (is ‹- ∈ ?S^+›)
      proof (induction rule: tranclp-induct)
        case (base y)
        then show ?case by auto
      next
        case (step T U) note st = this(1) and cp = this(2) and IH = this(3)[OF this(4)] and
          twl = this(4)
        have ‹twl-struct-invs T›
          by (metis (no-types, lifting) IH Nitpick.tranclp-unfold cdcl-twl-cp-twl-struct-invs
```

```
        converse-tranclpE)
      then have ‹(U, T) ∈ ?S⁺›
        using cp by auto
      then show ?case using IH by auto
    qed
  }
  then show ?thesis by blast
 qed
 show ?thesis using wf-trancl[OF wf-cdcl-twl-cp]  wf-subset[OF - H] by blast
qed

lemma wf-cdcl-twl-stgy:
 ‹wf {(T, S). twl-struct-invs S ∧ cdcl-twl-stgy S T}› (is ‹wf ?TWL›)
proof −
 let ?CDCL = ‹{(T, S). pcdcl-all-struct-invs (pstate_W-of S) ∧
    pcdcl-tcore (pstate_W-of S) (pstate_W-of T)}›
 let ?P = ‹{(T, S). pstate_W-of S = pstate_W-of T ∧
   (literals-to-update-measure T, literals-to-update-measure S) ∈ lexn less-than 2}›

 have wf-p-m:
   ‹wf {(T, S). (literals-to-update-measure T, literals-to-update-measure S) ∈ lexn less-than 2}›
   using wf-if-measure-f[of ‹lexn less-than 2› literals-to-update-measure] by (auto simp: wf-lexn)
 have ‹wf ?CDCL›
   by (rule wf-subset[OF wf-cdcl_W-stgy-state_W-of])
     (auto simp: twl-struct-invs-def)
 moreover have ‹wf ?P›
   by (rule wf-subset[OF wf-p-m]) auto
 moreover have ‹?CDCL O ?P ⊆ ?CDCL›
   by auto
 ultimately have ‹wf (?CDCL ∪ ?P)›
   by (rule wf-union-compatible)

 moreover have ‹?TWL ⊆ ?CDCL ∪ ?P›
 proof
   fix x
   assume x-TWL: ‹x ∈ ?TWL›
   then obtain S T where x: ‹x = (T, S)› by auto

   have twl: ‹twl-struct-invs S› and cdcl: ‹cdcl-twl-stgy S T›
     using x-TWL x by auto
   have ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)› and
     ‹pcdcl-all-struct-invs (pstate_W-of S)›
     using twl by (auto simp: twl-struct-invs-def pcdcl-all-struct-invs-def)
   moreover have ‹pcdcl-tcore (pstate_W-of S) (pstate_W-of T) ∨
     (pstate_W-of S = pstate_W-of T ∧
       (literals-to-update-measure T, literals-to-update-measure S) ∈ lexn less-than 2)›
     using cdcl cdcl-twl-stgy-cdcl_W-stgy2[OF cdcl twl] by (auto simp: pcdcl-core-stgy-pcdcl
       pcdcl-tcore.simps pcdcl-tcore-stgy.simps)
   ultimately show ‹x ∈ ?CDCL ∪ ?P›
     unfolding x by blast
 qed
 ultimately show ?thesis
   using wf-subset[of ‹?CDCL ∪ ?P›] by blast
qed

lemma tranclp-wf-cdcl-twl-stgy:
```

```
  ‹wf {(T, S). twl-struct-invs S ∧ cdcl-twl-stgy⁺⁺ S T}›
proof −
  have H: ‹{(T, S). twl-struct-invs S ∧ cdcl-twl-stgy⁺⁺ S T} ⊆
    {(T, S). twl-struct-invs S ∧ cdcl-twl-stgy S T}⁺›
  proof −
    { fix T S :: ‹'v twl-st›
      assume ‹cdcl-twl-stgy⁺⁺ S T› ‹twl-struct-invs S›
      then have ‹(T, S) ∈ {(T, S). twl-struct-invs S ∧ cdcl-twl-stgy S T}⁺› (is ‹- ∈ ?S⁺›)
      proof (induction rule: tranclp-induct)
        case (base y)
        then show ?case by auto
      next
        case (step T U) note st = this(1) and stgy = this(2) and IH = this(3)[OF this(4)] and
          twl = this(4)
        have ‹twl-struct-invs T›
          by (metis (no-types, lifting) IH Nitpick.tranclp-unfold cdcl-twl-stgy-twl-struct-invs
           converse-tranclpE)
        then have ‹(U, T) ∈ ?S⁺›
          using stgy by auto
        then show ?case using IH by auto
      qed
    }
    then show ?thesis by blast
  qed
  show ?thesis using wf-trancl[OF wf-cdcl-twl-stgy]  wf-subset[OF - H] by blast
qed

lemma rtranclp-cdcl-twl-o-stgyD: ‹cdcl-twl-o** S T ⟹ cdcl-twl-stgy** S T›
  using rtranclp-mono[of cdcl-twl-o cdcl-twl-stgy] cdcl-twl-stgy.intros(2)
  by blast

lemma rtranclp-cdcl-twl-cp-stgyD: ‹cdcl-twl-cp** S T ⟹ cdcl-twl-stgy** S T›
  using rtranclp-mono[of cdcl-twl-cp cdcl-twl-stgy] cdcl-twl-stgy.intros(1)
  by blast

lemma tranclp-cdcl-twl-o-stgyD: ‹cdcl-twl-o⁺⁺ S T ⟹ cdcl-twl-stgy⁺⁺ S T›
  using tranclp-mono[of cdcl-twl-o cdcl-twl-stgy] cdcl-twl-stgy.intros(2)
  by blast

lemma tranclp-cdcl-twl-cp-stgyD: ‹cdcl-twl-cp⁺⁺ S T ⟹ cdcl-twl-stgy⁺⁺ S T›
  using tranclp-mono[of cdcl-twl-cp cdcl-twl-stgy] cdcl-twl-stgy.intros(1)
  by blast

lemma wf-cdcl-twl-o:
  ‹wf {(T, S::'v twl-st). twl-struct-invs S ∧ cdcl-twl-o S T}›
  by (rule wf-subset[OF wf-cdcl-twl-stgy]) (auto intro: cdcl-twl-stgy.intros)

lemma tranclp-wf-cdcl-twl-o:
  ‹wf {(T, S::'v twl-st). twl-struct-invs S ∧ cdcl-twl-o⁺⁺ S T}›
  by (rule wf-subset[OF tranclp-wf-cdcl-twl-stgy]) (auto dest: tranclp-cdcl-twl-o-stgyD)

lemma (in −)propa-cands-enqueued-mono:
  ‹U' ⊆# U ⟹ N' ⊆# N ⟹
    propa-cands-enqueued (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) ⟹
     propa-cands-enqueued (M, N', U', D, NE', UE', NS, US, N0, U0, WS, Q)›
  by (cases D) (auto 5 5)
```

lemma (**in** −)*confl-cands-enqueued-mono*:
‹$U' \subseteq\# U \Longrightarrow N' \subseteq\# N \Longrightarrow$
confl-cands-enqueued $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
confl-cands-enqueued $(M, N', U', D, NE', UE', NS, US, N0, U0, WS, Q)$›
by (*cases D*) *auto*

lemma (**in** −)*twl-st-exception-inv-mono*:
‹$U' \subseteq\# U \Longrightarrow N' \subseteq\# N \Longrightarrow$
twl-st-exception-inv $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
twl-st-exception-inv $(M, N', U', D, NE', UE', NS, US, N0, U0, WS, Q)$›
by (*cases D*) (*fastforce simp*: *twl-exception-inv.simps*)+

lemma (**in** −)*twl-st-inv-mono*:
‹$U' \subseteq\# U \Longrightarrow N' \subseteq\# N \Longrightarrow$
twl-st-inv $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
twl-st-inv $(M, N', U', D, NE', UE', NS, US, N0, U0, WS, Q)$›
by (*cases D*) (*fastforce simp*: *twl-st-inv.simps*)+

lemma (**in** −)*propa-cands-enqueued-subsumed-mono*:
‹$US' \subseteq\# US \Longrightarrow$
propa-cands-enqueued $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
propa-cands-enqueued $(M, N, U, D, NE, UE, NS, US', N0, U0, WS, Q)$›
by (*cases D*) (*auto 5 5*)

lemma (**in** −)*propa-cands-enqueued-U0-mono*:
‹$U0' \subseteq\# U0 \Longrightarrow$
propa-cands-enqueued $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
propa-cands-enqueued $(M, N, U, D, NE, UE, NS, US', N0, U0', WS, Q)$›
by (*cases D*) (*auto 5 5*)

lemma (**in** −)*confl-cands-enqueued-subsumed-mono*:
‹$US' \subseteq\# US \Longrightarrow$
confl-cands-enqueued $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
confl-cands-enqueued $(M, N, U, D, NE, UE, NS, US', N0, U0, WS, Q)$›
by (*cases D*) *auto*

lemma (**in** −)*confl-cands-enqueued-U0-mono*:
‹$U0' \subseteq\# U0 \Longrightarrow$
confl-cands-enqueued $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
confl-cands-enqueued $(M, N, U, D, NE, UE, NS, US, N0, U0', WS, Q)$›
by (*cases D*) *auto*

lemma (**in** −)*twl-st-exception-inv-subsumed-mono*:
‹$US' \subseteq\# US \Longrightarrow$
twl-st-exception-inv $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
twl-st-exception-inv $(M, N, U, D, NE, UE, NS, US', N0, U0, WS, Q)$›
by (*cases D*) (*fastforce simp*: *twl-exception-inv.simps*)+

lemma (**in** −)*twl-st-exception-inv-U0-mono*:
‹$U0' \subseteq\# U0 \Longrightarrow$
twl-st-exception-inv $(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q) \Longrightarrow$
twl-st-exception-inv $(M, N, U, D, NE, UE, NS, US, N0, U0', WS, Q)$›
by (*cases D*) (*fastforce simp*: *twl-exception-inv.simps*)+

lemma (**in** −)*twl-st-inv-subsumed-mono*:

‹*US'* ⊆# *US* ⟹
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US'*, *N0*, *U0'*, *WS*, *Q*)›
by (*cases D*) (*fastforce simp*: *twl-st-inv.simps*)+

lemma (**in** −)*twl-st-inv-U0-subsumed-mono*:
‹*U0'* ⊆# *U0* ⟹
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0'*, *WS*, *Q*)›
by (*cases D*) (*fastforce simp*: *twl-st-inv.simps*)+

lemma (**in** −) *rtranclp-cdcl-twl-stgy-twl-stgy-invs*:
assumes
‹*cdcl-twl-stgy*** *S T*› **and**
‹*twl-struct-invs S*› **and**
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-cp-twl-stgy-invs cdcl-twl-o-twl-stgy-invs cdcl-twl-stgy.simps*
rtranclp-cdcl-twl-stgy-twl-struct-invs **by** *blast*
using *assms* $cdcl_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-cdcl*$_W$*-stgy-invariant*
rtranclp-cdcl-twl-stgy-cdcl$_W$*-stgy*
done

lemma *cdcl-twl-stgy-get-init-learned-clss-mono*:
assumes ‹*cdcl-twl-stgy S T*›
shows ‹*get-init-learned-clss S* ⊆# *get-init-learned-clss T*›
using *assms*
by *induction* (*auto simp*: *cdcl-twl-cp.simps cdcl-twl-o.simps*)

lemma *rtranclp-cdcl-twl-stgy-get-init-learned-clss-mono*:
assumes ‹*cdcl-twl-stgy*** *S T*›
shows ‹*get-init-learned-clss S* ⊆# *get-init-learned-clss T*›
using *assms*
by *induction* (*auto dest*!: *cdcl-twl-stgy-get-init-learned-clss-mono*)

lemma *cdcl-twl-o-all-learned-diff-learned*:
assumes ‹*cdcl-twl-o S T*›
shows
‹*clause* ‘# *get-learned-clss S* ⊆# *clause* ‘# *get-learned-clss T* ∧
get-init-learned-clss S ⊆# *get-init-learned-clss T* ∧
get-all-init-clss S = *get-all-init-clss T*›
by (*use assms* **in** ‹*induction rule*: *cdcl-twl-o.induct*›)
(*auto simp*: *update-clauses.simps size-Suc-Diff1*)

lemma *cdcl-twl-cp-all-learned-diff-learned*:
assumes ‹*cdcl-twl-cp S T*›
shows
‹*clause* ‘# *get-learned-clss S* = *clause* ‘# *get-learned-clss T* ∧
get-init-learned-clss S = *get-init-learned-clss T* ∧
get-all-init-clss S = *get-all-init-clss T*›
apply (*use assms* **in** ‹*induction rule*: *cdcl-twl-cp.induct*›)

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *D*
by (*cases D*)
(*auto simp*: *update-clauses.simps size-Suc-Diff1 dest*!: *multi-member-split*)
done

lemma *cdcl-twl-stgy-all-learned-diff-learned*:
assumes ‹*cdcl-twl-stgy S T*›
shows
‹*clause* ‘# *get-learned-clss S* ⊆# *clause* ‘# *get-learned-clss T* ∧
get-init-learned-clss S ⊆# *get-init-learned-clss T* ∧
get-all-init-clss S = *get-all-init-clss T*›
by (*use assms* **in** ‹*induction rule*: *cdcl-twl-stgy.induct*›)
(*auto simp*: *cdcl-twl-cp-all-learned-diff-learned cdcl-twl-o-all-learned-diff-learned*)

lemma *rtranclp-cdcl-twl-stgy-all-learned-diff-learned*:
assumes ‹*cdcl-twl-stgy*** *S T*›
shows
‹*clause* ‘# *get-learned-clss S* ⊆# *clause* ‘# *get-learned-clss T* ∧
get-init-learned-clss S ⊆# *get-init-learned-clss T* ∧
get-all-init-clss S = *get-all-init-clss T*›
by (*use assms* **in** ‹*induction rule*: *rtranclp-induct*›)
(*auto dest*: *cdcl-twl-stgy-all-learned-diff-learned*)

lemma *cdcl-twl-stgy-cdcl$_W$-stgy3*:
assumes ‹*cdcl-twl-stgy S T*› **and** *twl*: ‹*twl-struct-invs S*› **and**
‹*clauses-to-update S* = {#}› **and**
‹*literals-to-update S* = {#}›
shows ‹*pcdcl-tcore-stgy* (*pstate$_W$-of S*) (*pstate$_W$-of T*)›
using *cdcl-twl-stgy-cdcl$_W$-stgy2*[*OF assms*(*1*,*2*)] *assms*(*3*−)
by (*auto simp*: *lexn2-conv*)

lemma *tranclp-cdcl-twl-stgy-cdcl$_W$-stgy*:
assumes *ST*: ‹*cdcl-twl-stgy*++ *S T*› **and**
twl: ‹*twl-struct-invs S*› **and**
‹*clauses-to-update S* = {#}› **and**
‹*literals-to-update S* = {#}›
shows ‹*pcdcl-tcore-stgy*++ (*pstate$_W$-of S*) (*pstate$_W$-of T*)›
proof −
obtain *S′* **where**
SS′: ‹*cdcl-twl-stgy S S′*› **and**
S′T: ‹*cdcl-twl-stgy*** *S′ T*›
using *ST* **unfolding** *tranclp-unfold-begin* **by** *blast*

have *1*: ‹*pcdcl-tcore-stgy* (*pstate$_W$-of S*) (*pstate$_W$-of S′*)›
using *cdcl-twl-stgy-cdcl$_W$-stgy3*[*OF SS′ assms*(*2*−*4*)]
by *blast*
have *struct-S′*: ‹*twl-struct-invs S′*›
using *twl SS′* **by** (*blast intro*: *cdcl-twl-stgy-twl-struct-invs*)
have *2*: ‹*pcdcl-tcore-stgy*** (*pstate$_W$-of S′*) (*pstate$_W$-of T*)›
apply (*rule rtranclp-cdcl-twl-stgy-cdcl$_W$-stgy*)
apply (*rule S′T*)
by (*rule struct-S′*)

show *?thesis*
using *1 2* **by** *auto*
qed

definition *final-twl-state* **where**
‹*final-twl-state S* ⟷
no-step cdcl-twl-stgy S ∨ (*get-conflict S* ≠ *None* ∧ *count-decided* (*get-trail S*) = *0*)›

definition *partial-conclusive-TWL-norestart-run* :: ‹'*v twl-st* ⇒ (*bool* × '*v twl-st*) *nres*› **where**
‹*partial-conclusive-TWL-norestart-run S* = *SPEC*(λ(*b*, *T*). *b* ⟶ *cdcl-twl-stgy*** *S T* ∧ *final-twl-state T*)›

definition *conclusive-TWL-norestart-run* :: ‹'*v twl-st* ⇒ '*v twl-st nres*› **where**
‹*conclusive-TWL-norestart-run S* = *SPEC*(λ*T*. *cdcl-twl-stgy*** *S T* ∧ *final-twl-state T*)›

lemma *conflict-of-level-unsatisfiable*:
assumes
struct: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv S*› **and**
dec: ‹*count-decided* (*trail S*) = *0*› **and**
confl: ‹*conflicting S* ≠ *None*› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init S*›
shows ‹*unsatisfiable* (*set-mset* (*init-clss S*))›
proof −
obtain *M N U D* **where** *S*: ‹*S* = (*M*, *N*, *U*, *Some D*)›
by (*cases S*) (*use confl* **in** ‹*auto simp*: $cdcl_W$*-restart-mset-state*›)
have [*simp*]: ‹*get-all-ann-decomposition M* = [([], *M*)]›
by (*rule no-decision-get-all-ann-decomposition*)
(*use dec* **in** ‹*auto simp*: *count-decided-def filter-empty-conv S* $cdcl_W$*-restart-mset-state*›)
have
N-U: ‹*N* ⊨*psm U*› **and**
M-D: ‹*M* ⊨*as CNot D*› **and**
N-U-M: ‹*set-mset N* ∪ *set-mset U* ⊨*ps unmark-l M*› **and**
n-d: ‹*no-dup M*› **and**
N-U-D: ‹*set-mset N* ∪ *set-mset U* ⊨*p D*›
using *assms*
by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def all-decomposition-implies-def*
S clauses-def $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*)
have ‹*set-mset N* ∪ *set-mset U* ⊨*ps CNot D*›
by (*rule true-clss-clss-true-clss-cls-true-clss-clss*[*OF N-U-M M-D*])
then have ‹*set-mset N* ⊨*ps CNot D*› ‹*set-mset N* ⊨*p D*›
using *N-U N-U-D true-clss-clss-left-right* **by** *blast*+
then have ‹*unsatisfiable* (*set-mset N*)›
by (*rule true-clss-clss-CNot-true-clss-cls-unsatisfiable*)

then show *?thesis*
by (*auto simp*: *S clauses-def dest*: *satisfiable-decreasing*)
qed

lemma *conflict-of-level-unsatisfiable2*:
assumes
struct: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv S*› **and**
dec: ‹*count-decided* (*trail S*) = *0*› **and**

confl: ‹*conflicting S* ≠ *None*›
shows ‹*unsatisfiable* (*set-mset* (*init-clss S* + *learned-clss S*))›
proof −
obtain *M N U D* **where** *S*: ‹*S* = (*M*, *N*, *U*, *Some D*)›
by (*cases S*) (*use confl* **in** ‹*auto simp*: $cdcl_W$*-restart-mset-state*›)
have [*simp*]: ‹*get-all-ann-decomposition M* = [([], *M*)]›
by (*rule no-decision-get-all-ann-decomposition*)
(*use dec* **in** ‹*auto simp*: *count-decided-def filter-empty-conv S*›)
have
M-D: ‹*M* ⊨*as CNot D*› **and**
N-U-M: ‹*set-mset N* ∪ *set-mset U* ⊨*ps unmark-l M*› **and**
n-d: ‹*no-dup M*› **and**
N-U-D: ‹*set-mset N* ∪ *set-mset U* ⊨*p D*›
using *assms*
by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def all-decomposition-implies-def*
S clauses-def $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*)
have ‹*set-mset N* ∪ *set-mset U* ⊨*ps CNot D*›
by (*rule true-clss-clss-true-clss-cls-true-clss-clss*[*OF N-U-M M-D*])
then have ‹*set-mset N* ∪ *set-mset U* ⊨*ps CNot D*› ‹*set-mset N* ∪ *set-mset U* ⊨*p D*›
using *N-U-D true-clss-clss-left-right* **by** *blast*+
then have ‹*unsatisfiable* (*set-mset N* ∪ *set-mset U*)›
by (*rule true-clss-clss-CNot-true-clss-cls-unsatisfiable*)

then show *?thesis*
by (*auto simp*: *S clauses-def dest*: *satisfiable-decreasing*)
qed

end
theory *Watched-Literals-Clauses*
imports *More-Sepref.WB-More-Refinement-List Watched-Literals-Transition-System*
begin

type-synonym *'v clause-l* = ‹*'v literal list*›
type-synonym *'v clauses-l* = ‹(*nat*, (*'v clause-l* × *bool*)) *fmap*›

abbreviation *watched-l* :: ‹*'a clause-l* ⇒ *'a clause-l*› **where**
‹*watched-l l* ≡ *take 2 l*›

abbreviation *unwatched-l* :: ‹*'a clause-l* ⇒ *'a clause-l*› **where**
‹*unwatched-l l* ≡ *drop 2 l*›

fun *twl-clause-of* :: ‹*'a clause-l* ⇒ *'a clause twl-clause*› **where**
‹*twl-clause-of l* = *TWL-Clause* (*mset* (*watched-l l*)) (*mset* (*unwatched-l l*))›

abbreviation *clause-in* :: ‹*'v clauses-l* ⇒ *nat* ⇒ *'v clause-l*› (**infix** ∝ *101*) **where**
‹*N* ∝ *i* ≡ *fst* (*the* (*fmlookup N i*))›

abbreviation *clause-upd* :: ‹*'v clauses-l* ⇒ *nat* ⇒ *'v clause-l* ⇒ *'v clauses-l*› **where**
‹*clause-upd N i C* ≡ *fmupd i* (*C*, *snd* (*the* (*fmlookup N i*))) *N*›

Taken from *fun-upd*.

nonterminal *updclsss* **and** *updclss*

syntax

-updclss :: *'a clauses-l* ⇒ *'a* ⇒ *updclss* ((*2*- ↪/ -))
:: *updbind* ⇒ *updbinds* (-)
-updclsss:: *updclss* ⇒ *updclsss* ⇒ *updclsss* (-,/ -)
-Updateclss :: *'a* ⇒ *updclss* ⇒ *'a* (-/'((-)') [*1000*, *0*] *900*)

translations
-Updateclss f (*-updclsss b bs*) ⇌ *-Updateclss* (*-Updateclss f b*) *bs*
f(*x* ↪ *y*) ⇌ *CONST clause-upd f x y*

abbreviation *ran-mf* :: ‹*'v clauses-l* ⇒ *'v clause-l multiset*› **where**
‹*ran-mf N* ≡ *fst* ‘# *ran-m N*›

abbreviation *learned-clss-l* :: ‹*'v clauses-l* ⇒ (*'v clause-l* × *bool*) *multiset*› **where**
‹*learned-clss-l N* ≡ {#*C* ∈# *ran-m N*. ¬*snd C*#}›

abbreviation *learned-clss-lf* :: ‹*'v clauses-l* ⇒ *'v clause-l multiset*› **where**
‹*learned-clss-lf N* ≡ *fst* ‘# *learned-clss-l N*›

abbreviation *init-clss-l* :: ‹*'v clauses-l* ⇒ (*'v clause-l* × *bool*) *multiset*› **where**
‹*init-clss-l N* ≡ {#*C* ∈# *ran-m N*. *snd C*#}›

abbreviation *init-clss-lf* :: ‹*'v clauses-l* ⇒ *'v clause-l multiset*› **where**
‹*init-clss-lf N* ≡ *fst* ‘# *init-clss-l N*›

abbreviation *all-clss-l* :: ‹*'v clauses-l* ⇒ (*'v clause-l* × *bool*) *multiset*› **where**
‹*all-clss-l N* ≡ *init-clss-l N* + *learned-clss-l N*›

lemma *all-clss-l-ran-m*[*simp*]:
‹*all-clss-l N* = *ran-m N*›
by (*metis multiset-partition*)

abbreviation *all-clss-lf* :: ‹*'v clauses-l* ⇒ *'v clause-l multiset*› **where**
‹*all-clss-lf N* ≡ *init-clss-lf N* + *learned-clss-lf N*›

lemma *all-clss-lf-ran-m*: ‹*all-clss-lf N* = *fst* ‘# *ran-m N*›
by (*metis* (*no-types*) *image-mset-union multiset-partition*)

abbreviation *irred* :: ‹*'v clauses-l* ⇒ *nat* ⇒ *bool*› **where**
‹*irred N C* ≡ *snd* (*the* (*fmlookup N C*))›

definition *irred'* **where** ‹*irred'* = *irred*›

lemma *ran-m-ran*: ‹*fset-mset* (*ran-m N*) = *fmran N*›
unfolding *ran-m-def ran-def*
apply (*auto simp*: *fmlookup-ran-iff dom-m-def elim*!: *fmdomE*)
apply (*metis fmdomE notin-fset option.sel*)
by (*metis* (*no-types*, *lifting*) *fmdomI fmember.rep-eq image-iff option.sel*)

lemma *ran-m-clause-upd*:
assumes
NC: ‹*C* ∈# *dom-m N*›
shows ‹*ran-m* (*N*(*C* ↪ *C'*)) =
add-mset (*C'*, *irred N C*) (*remove1-mset* (*N* ∝ *C*, *irred N C*) (*ran-m N*))›
proof −
define *N'* **where**

```
    ‹N' = fmdrop C N›
  have N-N': ‹dom-m N = add-mset C (dom-m N')›
    using NC unfolding N'-def by auto
  have ‹C ∉# dom-m N'›
    using NC distinct-mset-dom[of N] unfolding N-N' by auto
  then show ?thesis
    by (auto simp: N-N' ran-m-def mset-set.insert-remove image-mset-remove1-mset-if
      intro!: image-mset-cong)
qed

lemma ran-m-mapsto-upd:
  assumes
    NC: ‹C ∈# dom-m N›
  shows ‹ran-m (fmupd C C' N) =
        add-mset C' (remove1-mset (N ∝ C, irred N C) (ran-m N))›
proof −
  define N' where
    ‹N' = fmdrop C N›
  have N-N': ‹dom-m N = add-mset C (dom-m N')›
    using NC unfolding N'-def by auto
  have ‹C ∉# dom-m N'›
    using NC distinct-mset-dom[of N] unfolding N-N' by auto
  then show ?thesis
    by (auto simp: N-N' ran-m-def mset-set.insert-remove image-mset-remove1-mset-if
      intro!: image-mset-cong)
qed

lemma ran-m-mapsto-upd-notin:
  assumes
    NC: ‹C ∉# dom-m N›
  shows ‹ran-m (fmupd C C' N) = add-mset C' (ran-m N)›
  using NC
  by (auto simp: ran-m-def mset-set.insert-remove image-mset-remove1-mset-if
      intro!: image-mset-cong split: if-splits)

lemma learned-clss-l-update[simp]:
  ‹bh ∈# dom-m ax ⟹ size (learned-clss-l (ax(bh ↪ C))) = size (learned-clss-l ax)›
  by (auto simp: ran-m-clause-upd size-Diff-singleton-if dest!: multi-member-split)
    (auto simp: ran-m-def)

lemma Ball-ran-m-dom:
  ‹(∀ x∈#ran-m N. P (fst x)) ⟷ (∀ x∈#dom-m N. P (N ∝ x))›
  by (auto simp: ran-m-def)

lemma Ball-ran-m-dom-struct-wf:
  ‹(∀ x∈#ran-m N. struct-wf-twl-cls (twl-clause-of (fst x))) ⟷
    (∀ x∈# dom-m N. struct-wf-twl-cls (twl-clause-of (N ∝ x)))›
  by (rule Ball-ran-m-dom)

lemma init-clss-lf-fmdrop[simp]:
  ‹irred N C ⟹ C ∈# dom-m N ⟹ init-clss-lf (fmdrop C N) = remove1-mset (N∝C) (init-clss-lf
N)›
  using distinct-mset-dom[of N]
  by (auto simp: ran-m-def image-mset-If-eq-notin[of C - the] dest!: multi-member-split)

lemma init-clss-lf-fmdrop-irrelev[simp]:
```

‹¬irred N C ⟹ init-clss-lf (fmdrop C N) = init-clss-lf N›
using *distinct-mset-dom*[*of N*]
apply (*cases* ‹C ∈# dom-m N›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *learned-clss-lf-lf-fmdrop*[*simp*]:
‹¬irred N C ⟹ C ∈# dom-m N ⟹ learned-clss-lf (fmdrop C N) = remove1-mset (N∝C) (learned-clss-lf N)›
using *distinct-mset-dom*[*of N*]
apply (*cases* ‹C ∈# dom-m N›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *learned-clss-l-l-fmdrop*: ‹¬ irred N C ⟹ C ∈# dom-m N ⟹
learned-clss-l (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*)) (*learned-clss-l N*)›
using *distinct-mset-dom*[*of N*]
apply (*cases* ‹C ∈# dom-m N›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *learned-clss-lf-lf-fmdrop-irrelev*[*simp*]:
‹irred N C ⟹ learned-clss-lf (fmdrop C N) = learned-clss-lf N›
using *distinct-mset-dom*[*of N*]
apply (*cases* ‹C ∈# dom-m N›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *ran-mf-lf-fmdrop*[*simp*]:
‹C ∈# dom-m N ⟹ ran-mf (fmdrop C N) = remove1-mset (N∝C) (ran-mf N)›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - ‹λx. fst* (*the x*)›] *dest*!: *multi-member-split*)

lemma *ran-mf-lf-fmdrop-notin*[*simp*]:
‹C ∉# dom-m N ⟹ ran-mf (fmdrop C N) = ran-mf N›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - ‹λx. fst* (*the x*)›] *dest*!: *multi-member-split*)

lemma *lookup-None-notin-dom-m*[*simp*]:
‹*fmlookup N i* = *None* ⟷ *i* ∉# *dom-m N*›
by (*auto simp*: *dom-m-def fmlookup-dom-iff fmember.rep-eq*[*symmetric*])

While it is tempting to mark the two following theorems as [simp], this would break more simplifications since *ran-mf* is only an abbreviation for *ran-m*.

lemma *ran-m-fmdrop*:
‹C ∈# dom-m N ⟹ ran-m (fmdrop C N) = remove1-mset (N ∝ C, irred N C) (ran-m N)›
using *distinct-mset-dom*[*of N*]
by (*cases* ‹*fmlookup N C*›)
(*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - ‹λx. fst* (*the x*)›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *ran-m-fmdrop-notin*:
‹C ∉# dom-m N ⟹ ran-m (fmdrop C N) = ran-m N›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - ‹λx. fst* (*the x*)›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *init-clss-l-fmdrop-irrelev*:

‹¬*irred N C* ⟹ *init-clss-l* (*fmdrop C N*) = *init-clss-l N*›
using *distinct-mset-dom*[*of N*]
apply (*cases* ‹*C* ∈# *dom-m N*›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *init-clss-l-fmdrop*:
‹*irred N C* ⟹ *C* ∈# *dom-m N* ⟹ *init-clss-l* (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*))
(*init-clss-l N*)›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *ran-m-lf-fmdrop*:
‹*C* ∈# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*)) (*ran-m N*)›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - ‹λx. fst* (*the x*)›] *dest*!: *multi-member-split*
intro!: *image-mset-cong*)
lemma *set-clauses-simp*[*simp*]:
‹*f* ‘ {*a*. *a* ∈# *ran-m N* ∧ ¬ *snd a*} ∪ *f* ‘ {*a*. *a* ∈# *ran-m N* ∧ *snd a*} ∪ *A* =
f ‘ {*a*. *a* ∈# *ran-m N*} ∪ *A*›
by *auto*

lemma *init-clss-l-clause-upd*:
‹*C* ∈# *dom-m N* ⟹ *irred N C* ⟹
init-clss-l (*N*(*C* ↪ *C'*)) =
add-mset (*C'*, *irred N C*) (*remove1-mset* (*N* ∝ *C*, *irred N C*) (*init-clss-l N*))›
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd*:
‹*C* ∈# *dom-m N* ⟹ *irred N C* ⟹
init-clss-l (*fmupd C* (*C'*, *True*) *N*) =
add-mset (*C'*, *irred N C*) (*remove1-mset* (*N* ∝ *C*, *irred N C*) (*init-clss-l N*))›
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *learned-clss-l-mapsto-upd*:
‹*C* ∈# *dom-m N* ⟹ ¬*irred N C* ⟹
learned-clss-l (*fmupd C* (*C'*, *False*) *N*) =
add-mset (*C'*, *irred N C*) (*remove1-mset* (*N* ∝ *C*, *irred N C*) (*learned-clss-l N*))›
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd-irrel*: ‹*C* ∈# *dom-m N* ⟹ ¬*irred N C* ⟹
init-clss-l (*fmupd C* (*C'*, *False*) *N*) = *init-clss-l N*›
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd-irrel-notin*: ‹*C* ∉# *dom-m N* ⟹
init-clss-l (*fmupd C* (*C'*, *False*) *N*) = *init-clss-l N*›
by (*auto simp*: *ran-m-mapsto-upd-notin*)

lemma *learned-clss-l-mapsto-upd-irrel*: ‹*C* ∈# *dom-m N* ⟹ *irred N C* ⟹
learned-clss-l (*fmupd C* (*C'*, *True*) *N*) = *learned-clss-l N*›
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *learned-clss-l-mapsto-upd-notin*: ‹*C* ∉# *dom-m N* ⟹
learned-clss-l (*fmupd C* (*C'*, *False*) *N*) = *add-mset* (*C'*, *False*) (*learned-clss-l N*)›
by (*auto simp*: *ran-m-mapsto-upd-notin*)

lemma *in-ran-mf-clause-inI*[*intro*]:

‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow i = \mathit{irred}\ N\ C \Longrightarrow (N \propto C, i) \in\# \mathit{ran\text{-}m}\ N$›
by (*auto simp*: *ran-m-def dom-m-def*)

lemma *init-clss-l-mapsto-upd-notin*:
‹$C \notin\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{init\text{-}clss\text{-}l}\ (\mathit{fmupd}\ C\ (C', \mathit{True})\ N) =$
$\mathit{add\text{-}mset}\ (C', \mathit{True})\ (\mathit{init\text{-}clss\text{-}l}\ N)$›
by (*auto simp*: *ran-m-mapsto-upd-notin*)

lemma *learned-clss-l-mapsto-upd-notin-irrelev*: ‹$C \notin\# \mathit{dom\text{-}m}\ N \Longrightarrow$
$\mathit{learned\text{-}clss\text{-}l}\ (\mathit{fmupd}\ C\ (C', \mathit{True})\ N) = \mathit{learned\text{-}clss\text{-}l}\ N$›
by (*auto simp*: *ran-m-mapsto-upd-notin*)

lemma *clause-twl-clause-of*: ‹$\mathit{clause}\ (\mathit{twl\text{-}clause\text{-}of}\ C) = \mathit{mset}\ C$› **for** C
by (*cases C*; *cases* ‹*tl C*›) *auto*

lemma *learned-clss-l-l-fmdrop-irrelev*: ‹$\mathit{irred}\ N\ C \Longrightarrow$
$\mathit{learned\text{-}clss\text{-}l}\ (\mathit{fmdrop}\ C\ N) = \mathit{learned\text{-}clss\text{-}l}\ N$›
using *distinct-mset-dom*[*of N*]
apply (*cases* ‹$C \in\# \mathit{dom\text{-}m}\ N$›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C - the*] *dest*!: *multi-member-split*)

lemma *init-clss-l-fmdrop-if*:
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{init\text{-}clss\text{-}l}\ (\mathit{fmdrop}\ C\ N) = (\mathit{if}\ \mathit{irred}\ N\ C\ \mathit{then}\ \mathit{remove1\text{-}mset}\ (\mathit{the}\ (\mathit{fmlookup}\ N\ C))\ (\mathit{init\text{-}clss\text{-}l}\ N)$
$\mathit{else}\ \mathit{init\text{-}clss\text{-}l}\ N)$›
by (*auto simp*: *init-clss-l-fmdrop init-clss-l-fmdrop-irrelev*)

lemma *init-clss-l-fmupd-if*:
‹$C' \notin\# \mathit{dom\text{-}m}\ \mathit{new} \Longrightarrow \mathit{init\text{-}clss\text{-}l}\ (\mathit{fmupd}\ C'\ D\ \mathit{new}) = (\mathit{if}\ \mathit{snd}\ D\ \mathit{then}\ \mathit{add\text{-}mset}\ D\ (\mathit{init\text{-}clss\text{-}l}\ \mathit{new})$
$\mathit{else}\ \mathit{init\text{-}clss\text{-}l}\ \mathit{new})$›
by (*cases D*) (*auto simp*: *init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-notin*)

lemma *learned-clss-l-fmdrop-if*:
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{learned\text{-}clss\text{-}l}\ (\mathit{fmdrop}\ C\ N) = (\mathit{if}\ \neg\mathit{irred}\ N\ C\ \mathit{then}\ \mathit{remove1\text{-}mset}\ (\mathit{the}\ (\mathit{fmlookup}$
$N\ C))\ (\mathit{learned\text{-}clss\text{-}l}\ N)$
$\mathit{else}\ \mathit{learned\text{-}clss\text{-}l}\ N)$›
by (*auto simp*: *learned-clss-l-l-fmdrop learned-clss-l-l-fmdrop-irrelev*)

lemma *learned-clss-l-fmupd-if*:
‹$C' \notin\# \mathit{dom\text{-}m}\ \mathit{new} \Longrightarrow \mathit{learned\text{-}clss\text{-}l}\ (\mathit{fmupd}\ C'\ D\ \mathit{new}) = (\mathit{if}\ \neg\mathit{snd}\ D\ \mathit{then}\ \mathit{add\text{-}mset}\ D\ (\mathit{learned\text{-}clss\text{-}l}$
$\mathit{new})\ \mathit{else}\ \mathit{learned\text{-}clss\text{-}l}\ \mathit{new})$›
by (*cases D*) (*auto simp*: *learned-clss-l-mapsto-upd-notin-irrelev*
learned-clss-l-mapsto-upd-notin)

definition *op-clauses-at* :: ‹$'v\ \mathit{clauses\text{-}l} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow 'v\ \mathit{literal}$› **where**
‹$\mathit{op\text{-}clauses\text{-}at}\ N\ C\ i = N \propto C\ !\ i$›

definition *mop-clauses-at* :: ‹$'v\ \mathit{clauses\text{-}l} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow 'v\ \mathit{literal}\ \mathit{nres}$› **where**
‹$\mathit{mop\text{-}clauses\text{-}at}\ N\ C\ i = \mathit{do}\ \{$
$\mathit{ASSERT}(C \in\# \mathit{dom\text{-}m}\ N);$
$\mathit{ASSERT}(i < \mathit{length}\ (N \propto C));$
$\mathit{RETURN}\ (N \propto C\ !\ i)$
$\}$›

lemma *mop-clauses-at*:
  ‹(*uncurry2 mop-clauses-at*, *uncurry2* (*RETURN ooo op-clauses-at*)) ∈
  [λ((*N*, *C*), *i*). *C* ∈# *dom-m N* ∧ *i* < *length* (*N* ∝ *C*)]$_f$
  Id ×$_f$ *Id* ×$_f$ *Id* → ⟨*Id*⟩*nres-rel*›
  by (*auto simp*: *mop-clauses-at-def op-clauses-at-def intro*!: *frefI nres-relI*)

definition *mop-clauses-swap* :: ‹'*v clauses-l* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*v clauses-l nres*› **where**
‹*mop-clauses-swap N C i j* = *do* {
  ASSERT(*C* ∈# *dom-m N*);
  ASSERT(*i* < *length* (*N* ∝ *C*));
  ASSERT(*j* < *length* (*N* ∝ *C*));
  RETURN (*N*(*C* ↪ (*swap* (*N* ∝ *C*) *i j*)))
}›

definition *op-clauses-swap* :: ‹'*v clauses-l* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*v clauses-l*› **where**
‹*op-clauses-swap N C i j* = (*N*(*C* ↪ (*swap* (*N* ∝ *C*) *i j*)))›

lemma *mop-clauses-swap*:
  ‹(*uncurry3 mop-clauses-swap*, *uncurry3* (*RETURN oooo op-clauses-swap*)) ∈
  [λ(((*N*, *C*), *i*), *j*). *C* ∈# *dom-m N* ∧ *i* < *length* (*N* ∝ *C*) ∧ *j* < *length* (*N* ∝ *C*)]$_f$
  Id ×$_f$ *Id* ×$_f$ *Id* ×$_f$ *Id* → ⟨*Id*⟩*nres-rel*›
  by (*auto simp*: *mop-clauses-swap-def op-clauses-swap-def intro*!: *frefI nres-relI*)

lemma *mop-clauses-at-itself*:
  ‹(*uncurry2 mop-clauses-at*, *uncurry2 mop-clauses-at*) ∈ *Id* ×$_f$ *Id* ×$_f$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
  by (*auto intro*!: *frefI nres-relI*)

lemma *mop-clauses-at-itself-spec*:
  ‹((*N*, *C*, *i*), (*N'*, *C'*, *i'*)) ∈ *Id* ⟹
    mop-clauses-at N C i ≤ ⇓ {(*L*, *L'*). *L* = *L'* ∧ *L* = *N* ∝ *C* ! *i*} (*mop-clauses-at N' C' i'*)›
  by (*auto intro*!: *frefI nres-relI ASSERT-refine-right simp*: *mop-clauses-at-def*)

lemma *mop-clauses-at-itself-spec2*:
  ‹((*N*, *C*, *i*), (*N'*, *C'*, *i'*)) ∈ *Id* ⟹
    mop-clauses-at N C i ≤ ⇓ {(*L*, *L'*). *L* = *L'* ∧ *L* = *N* ∝ *C* ! *i* ∧ *C* ∈# *dom-m N* ∧ *i* < *length* (*N* ∝ *C*)}
     (*mop-clauses-at N' C' i'*)›
  by (*auto intro*!: *frefI nres-relI ASSERT-refine-right simp*: *mop-clauses-at-def*)

lemma *mop-clauses-at-op-clauses-at-spec2*:
  ‹((*N*, *C*, *i*), (*N'*, *C'*, *i'*)) ∈ *Id* ⟹ *C* ∈# *dom-m N* ∧ *i* < *length* (*N* ∝ *C*) ⟹
    mop-clauses-at N C i ≤ ⇓ {(*L*, *L'*). *L* = *L'* ∧ *L* = *N* ∝ *C* ! *i*}
     (*RETURN* (*op-clauses-at N' C' i'*))›
  by (*auto intro*!: *frefI nres-relI ASSERT-refine-right*
   simp: *mop-clauses-at-def op-clauses-at-def*)

lemma *mop-clauses-swap-itself*:
  ‹(*uncurry3 mop-clauses-swap*, *uncurry3 mop-clauses-swap*) ∈ *Id* ×$_f$ *Id* ×$_f$ *Id* ×$_f$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
  by (*auto intro*!: *frefI nres-relI*)

lemma *mop-clauses-swap-itself-spec*:
  ‹((*N*, *C*, *i*, *j*), (*N'*, *C'*, *i'*, *j'*)) ∈ *Id* ⟹
    mop-clauses-swap N C i j ≤ ⇓ {(*L*, *L'*). *L* = *L'* ∧ *L* = *op-clauses-swap N' C' i' j'* ∧ *C'* ∈# *dom-m N*} (*mop-clauses-swap N' C' i' j'*)›
  by (*auto intro*!: *frefI nres-relI ASSERT-refine-right simp*: *mop-clauses-swap-def*

op-clauses-swap-def)

lemma *mop-clauses-swap-itself-spec2*:
‹((*N*, *C*, *i*, *j*), (*N'*, *C'*, *i'*, *j'*)) ∈ *Id* ⟹
mop-clauses-swap N C i j ≤ ⇓ {(*L*, *L'*). *L* = *L'* ∧ *L* = *op-clauses-swap N' C' i' j'* ∧ *C'* ∈# *dom-m N* ∧
i < *length* (*N* ∝ *C*) ∧ *j* < *length* (*N* ∝ *C*)} (*mop-clauses-swap N' C' i' j'*)›
by (*auto intro*!: *frefI nres-relI ASSERT-refine-right simp*: *mop-clauses-swap-def op-clauses-swap-def*)

end
theory *Watched-Literals-All-Literals*
imports *Watched-Literals-Clauses*
begin

Chapter 3

Set of all literals

type-synonym *ann-lits-l* = ‹(*nat*, *nat*) *ann-lits*›
type-synonym *clauses-to-update-ll* = ‹*nat list*›

3.1 Refinement

3.1.1 Set of all literals of the problem

definition *all-lits-of-mm* :: ‹'*a clauses* ⇒ '*a literal multiset*› **where**
‹*all-lits-of-mm Ls* = *Pos* ‘# (*atm-of* ‘# (∑# *Ls*)) + *Neg* ‘# (*atm-of* ‘# (∑# *Ls*))›

lemma *all-lits-of-mm-empty*[*simp*]: ‹*all-lits-of-mm* {#} = {#}›
by (*auto simp*: *all-lits-of-mm-def*)
lemma *in-all-lits-of-mm-ain-atms-of-iff*:
‹*L* ∈# *all-lits-of-mm N* ⟷ *atm-of L* ∈ *atms-of-mm N*›
by (*cases L*) (*auto simp*: *all-lits-of-mm-def atms-of-ms-def atms-of-def*)

lemma *all-lits-of-mm-union*:
‹*all-lits-of-mm* (*M* + *N*) = *all-lits-of-mm M* + *all-lits-of-mm N*›
unfolding *all-lits-of-mm-def* **by** *auto*

definition *all-lits-of-m* :: ‹'*a clause* ⇒ '*a literal multiset*› **where**
‹*all-lits-of-m Ls* = *Pos* ‘# (*atm-of* ‘# *Ls*) + *Neg* ‘# (*atm-of* ‘# *Ls*)›

lemma *all-lits-of-m-empty*[*simp*]: ‹*all-lits-of-m* {#} = {#}›
by (*auto simp*: *all-lits-of-m-def*)

lemma *all-lits-of-m-empty-iff*[*iff*]: ‹*all-lits-of-m A* = {#} ⟷ *A* = {#}›
by (*cases A*) (*auto simp*: *all-lits-of-m-def*)

lemma *in-all-lits-of-m-ain-atms-of-iff*: ‹*L* ∈# *all-lits-of-m N* ⟷ *atm-of L* ∈ *atms-of N*›
by (*cases L*) (*auto simp*: *all-lits-of-m-def atms-of-ms-def atms-of-def*)

lemma *in-clause-in-all-lits-of-m*: ‹*x* ∈# *C* ⟹ *x* ∈# *all-lits-of-m C*›
using *atm-of-lit-in-atms-of in-all-lits-of-m-ain-atms-of-iff* **by** *blast*

lemma *all-lits-of-mm-add-mset*:
‹*all-lits-of-mm* (*add-mset C N*) = (*all-lits-of-m C*) + (*all-lits-of-mm N*)›
by (*auto simp*: *all-lits-of-mm-def all-lits-of-m-def*)

lemma *all-lits-of-m-add-mset*:
‹*all-lits-of-m* (*add-mset L C*) = *add-mset L* (*add-mset* (−*L*) (*all-lits-of-m C*))›

by (*cases L*) (*auto simp*: *all-lits-of-m-def*)

lemma *all-lits-of-m-union*:
‹*all-lits-of-m* (*A* + *B*) = *all-lits-of-m A* + *all-lits-of-m B*›
by (*auto simp*: *all-lits-of-m-def*)

lemma *all-lits-of-m-mono*:
‹*D* ⊆# *D'* ⟹ *all-lits-of-m D* ⊆# *all-lits-of-m D'*›
by (*auto elim*!: *mset-le-addE simp*: *all-lits-of-m-union*)

lemma *all-lits-of-m-diffD*: ‹*x* ∈# *all-lits-of-m* (*D* − *D'*) ⟹ *x* ∈# *all-lits-of-m D*›
by (*auto simp*: *all-lits-of-m-def dest*: *in-diffD*)

lemma *in-all-lits-of-mm-uminusD*: ‹*x2* ∈# *all-lits-of-mm N* ⟹ −*x2* ∈# *all-lits-of-mm N*›
by (*auto simp*: *all-lits-of-mm-def*)

lemma *in-all-lits-of-mm-uminus-iff*: ‹−*x2* ∈# *all-lits-of-mm N* ⟷ *x2* ∈# *all-lits-of-mm N*›
by (*cases x2*) (*auto simp*: *all-lits-of-mm-def*)

lemma *all-lits-of-mm-diffD*:
‹*L* ∈# *all-lits-of-mm* (*A* − *B*) ⟹ *L* ∈# *all-lits-of-mm A*›
apply (*induction A arbitrary*: *B*)
subgoal by *auto*
subgoal for *a A' B*
by (*cases* ‹*a* ∈# *B*›)
(*fastforce dest*!: *multi-member-split*[*of a B*] *simp*: *all-lits-of-mm-add-mset*)+
done

lemma *all-lits-of-mm-mono*:
‹*set-mset A* ⊆ *set-mset B* ⟹ *set-mset* (*all-lits-of-mm A*) ⊆ *set-mset* (*all-lits-of-mm B*)›
by (*auto simp*: *all-lits-of-mm-def*)

3.2 Conversion from set of atoms to set of literals

We start in a context where we have an initial set of atoms. We later extend the locale to include a bound on the largest atom (in order to generate more efficient code).

context
fixes $\mathcal{A}_{in}$ :: ‹'*v multiset*›
begin

This is the *completion* of $\mathcal{A}_{in}$, containing the positive and the negation of every literal of $\mathcal{A}_{in}$:

definition $\mathcal{L}_{all}$ **where** ‹$\mathcal{L}_{all}$ = *poss* $\mathcal{A}_{in}$ + *negs* $\mathcal{A}_{in}$›

lemma *atms-of-*$\mathcal{L}_{all}$*-*$\mathcal{A}_{in}$: ‹*atms-of* $\mathcal{L}_{all}$ = *set-mset* $\mathcal{A}_{in}$›
unfolding $\mathcal{L}_{all}$*-def* **by** (*auto simp*: *atms-of-def image-Un image-image*)

definition *is-*$\mathcal{L}_{all}$ :: ‹'*v literal multiset* ⇒ *bool*› **where**
‹*is-*$\mathcal{L}_{all}$ *S* ⟷ *set-mset* $\mathcal{L}_{all}$ = *set-mset S*›

definition *literals-are-in-*$\mathcal{L}_{in}$ :: ‹'*v clause* ⇒ *bool*› **where**
‹*literals-are-in-*$\mathcal{L}_{in}$ *C* ⟷ *set-mset* (*all-lits-of-m C*) ⊆ *set-mset* $\mathcal{L}_{all}$›

lemma *literals-are-in-*$\mathcal{L}_{in}$*-empty*[*simp*]: ‹*literals-are-in-*$\mathcal{L}_{in}$ {#}›

by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-def*)

lemma *in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*: ‹*x* ∈# $\mathcal{L}_{all}$ ⟷ *atm-of x* ∈ *atms-of* $\mathcal{L}_{all}$›
by (*cases x*) (*auto simp*: $\mathcal{L}_{all}$*-def atms-of-def atm-of-eq-atm-of image-Un image-image*)

lemma *literals-are-in-$\mathcal{L}_{in}$-add-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$* (*add-mset L A*) ⟷ *literals-are-in-$\mathcal{L}_{in}$ A* ∧ *L* ∈# $\mathcal{L}_{all}$›
by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-def all-lits-of-m-add-mset in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*)

lemma *literals-are-in-$\mathcal{L}_{in}$-mono*:
assumes *N*: ‹*literals-are-in-$\mathcal{L}_{in}$ D'*› **and** *D*: ‹*D* ⊆# *D'*›
shows ‹*literals-are-in-$\mathcal{L}_{in}$ D*›
proof −
have ‹*set-mset* (*all-lits-of-m D*) ⊆ *set-mset* (*all-lits-of-m D'*)›
using *D* **by** (*auto simp*: *in-all-lits-of-m-ain-atms-of-iff atm-iff-pos-or-neg-lit*)
then show *?thesis*
using *N* **unfolding** *literals-are-in-$\mathcal{L}_{in}$-def* **by** *fast*
qed

lemma *literals-are-in-$\mathcal{L}_{in}$-sub*:
‹*literals-are-in-$\mathcal{L}_{in}$ y* ⟹ *literals-are-in-$\mathcal{L}_{in}$* (*y* − *z*)›
using *literals-are-in-$\mathcal{L}_{in}$-mono*[*of y* ‹*y* − *z*›] **by** *auto*

lemma *all-lits-of-m-subset-all-lits-of-mmD*:
‹*a* ∈# *b* ⟹ *set-mset* (*all-lits-of-m a*) ⊆ *set-mset* (*all-lits-of-mm b*)›
by (*auto simp*: *all-lits-of-m-def all-lits-of-mm-def*)

lemma *all-lits-of-m-remdups-mset*:
‹*set-mset* (*all-lits-of-m* (*remdups-mset N*)) = *set-mset* (*all-lits-of-m N*)›
by (*auto simp*: *all-lits-of-m-def*)

lemma *literals-are-in-$\mathcal{L}_{in}$-remdups*[*simp*]:
‹*literals-are-in-$\mathcal{L}_{in}$* (*remdups-mset N*) = *literals-are-in-$\mathcal{L}_{in}$ N*›
by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-def all-lits-of-m-remdups-mset*)

lemma *uminus-$\mathcal{A}_{in}$-iff*: ‹− *L* ∈# $\mathcal{L}_{all}$ ⟷ *L* ∈# $\mathcal{L}_{all}$›
by (*simp add*: *in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*)

definition *literals-are-in-$\mathcal{L}_{in}$-mm* :: ‹*'v clauses* ⇒ *bool*› **where**
‹*literals-are-in-$\mathcal{L}_{in}$-mm C* ⟷ *set-mset* (*all-lits-of-mm C*) ⊆ *set-mset* $\mathcal{L}_{all}$›

lemma *literals-are-in-$\mathcal{L}_{in}$-mm-add-msetD*:
‹*literals-are-in-$\mathcal{L}_{in}$-mm* (*add-mset C N*) ⟹ *L* ∈# *C* ⟹ *L* ∈# $\mathcal{L}_{all}$›
by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-mm-def all-lits-of-mm-add-mset*
all-lits-of-m-add-mset
dest!: *multi-member-split*)

lemma *literals-are-in-$\mathcal{L}_{in}$-mm-add-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$-mm* (*add-mset C N*) ⟷
literals-are-in-$\mathcal{L}_{in}$-mm N ∧ *literals-are-in-$\mathcal{L}_{in}$ C*›
unfolding *literals-are-in-$\mathcal{L}_{in}$-mm-def literals-are-in-$\mathcal{L}_{in}$-def*
by (*auto simp*: *all-lits-of-mm-add-mset*)

definition *literals-are-in-$\mathcal{L}_{in}$-trail* :: ‹(*'v*, *'mark*) *ann-lits* ⇒ *bool*› **where**
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* ⟷ *set-mset* (*lit-of* ‘# *mset M*) ⊆ *set-mset* $\mathcal{L}_{all}$›

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l*:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail M* $\Longrightarrow$ *a* $\in$ *lits-of-l M* $\Longrightarrow$ *a* $\in\#$ $\mathcal{L}_{all}$›
  by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-trail-def lits-of-def*)

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-uminus-in-lits-of-l*:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail M* $\Longrightarrow$ $-a$ $\in$ *lits-of-l M* $\Longrightarrow$ *a* $\in\#$ $\mathcal{L}_{all}$›
  by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-trail-def lits-of-def uminus-lit-swap uminus-$\mathcal{A}_{in}$-iff*)

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-uminus-in-lits-of-l-atms*:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail M* $\Longrightarrow$ $-a$ $\in$ *lits-of-l M* $\Longrightarrow$ *atm-of a* $\in\#$ $\mathcal{A}_{in}$›
  using *literals-are-in-$\mathcal{L}_{in}$-trail-uminus-in-lits-of-l*[*of M a*]
  unfolding *in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*[*symmetric*] *atms-of-$\mathcal{L}_{all}$-$\mathcal{A}_{in}$*[*symmetric*]
  .
end

lemma *isasat-input-ops-$\mathcal{L}_{all}$-empty*[*simp*]:
  ‹$\mathcal{L}_{all}$ {#} = {#}›
  unfolding *$\mathcal{L}_{all}$-def*
  by *auto*

lemma *$\mathcal{L}_{all}$-atm-of-all-lits-of-mm*: ‹*set-mset* (*$\mathcal{L}_{all}$* (*atm-of* ‘# *all-lits-of-mm A*)) = *set-mset* (*all-lits-of-mm A*)›
  apply (*auto simp*: *$\mathcal{L}_{all}$-def in-all-lits-of-mm-ain-atms-of-iff*)
  by (*metis* (*no-types*, *lifting*) *image-iff in-all-lits-of-mm-ain-atms-of-iff literal.exhaust-sel*)

definition *all-lits* :: ‹('*a*, '*v literal list* × '*b*) *fmap* ⇒ '*v literal multiset multiset* ⇒
   '*v literal multiset*› **where**
  ‹*all-lits S NUE* = *all-lits-of-mm* ((λ*C*. *mset* (*fst C*)) ‘# *ran-m S* + *NUE*)›

lemma *all-lits-alt-def*:
  ‹*all-lits S NUE* = *all-lits-of-mm* (*mset* ‘# *ran-mf S* + *NUE*)›
  unfolding *all-lits-def*
  by *auto*

lemma *all-lits-alt-def2*:
  ‹*all-lits S* (*NUE* + *NUS* + *N0S*) = *all-lits-of-mm* (*mset* ‘# *ran-mf S* + *NUE* + *NUS* + *N0S*)›
  ‹*all-lits S* (*NUE* + *NUS* + *N0S*) = *all-lits-of-mm* ((λ*C*. *mset* (*fst C*)) ‘# *ran-m S* + *NUE* + *NUS* + *N0S*)›
  unfolding *all-lits-def*
  by (*auto simp*: *ac-simps*)

lemma *literals-are-in-$\mathcal{L}_{in}$-mm-in-$\mathcal{L}_{all}$*:
  assumes
    N1: ‹*literals-are-in-$\mathcal{L}_{in}$-mm* $\mathcal{A}$ (*mset* ‘# *ran-mf xs*)› **and**
    i-xs: ‹*i* $\in\#$ *dom-m xs*› **and** *j-xs*: ‹*j* < *length* (*xs* ∝ *i*)›
  shows ‹*xs* ∝ *i* ! *j* $\in\#$ $\mathcal{L}_{all}$ $\mathcal{A}$›
proof −
  have ‹*xs* ∝ *i* $\in\#$ *ran-mf xs*›
    using *i-xs* **by** *auto*
  then have ‹*xs* ∝ *i* ! *j* ∈ *set-mset* (*all-lits-of-mm* (*mset* ‘# *ran-mf xs*))›
    using *j-xs* **by** (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff atms-of-ms-def Bex-def*
      intro!: *exI*[*of* - ‹*xs* ∝ *i*›])
  then show *?thesis*
    using *N1* **unfolding** *literals-are-in-$\mathcal{L}_{in}$-mm-def* **by** *blast*
qed

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l-atms*:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}_{in}$ $M \Longrightarrow a \in$ *lits-of-l* $M \Longrightarrow$ *atm-of* $a \in\#$ $\mathcal{A}_{in}$›
  using *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l*[*of* $\mathcal{A}_{in}$ M a]
  unfolding *in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*[*symmetric*] *atms-of-$\mathcal{L}_{all}$-$\mathcal{A}_{in}$*[*symmetric*]
  .

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-Cons*:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}_{in}$ $(L \# M) \longleftrightarrow$
    literals-are-in-$\mathcal{L}_{in}$-trail $\mathcal{A}_{in}$ $M \wedge$ *lit-of* $L \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}_{in}$›
  by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-trail-def*)

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-empty*[*simp*]:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}$ []›
  by (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-trail-def*)

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-lit-of-mset*:
  ‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}$ $M =$ *literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}$ (*lit-of* ‘# *mset* M)›
  by (*induction* M) (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-add-mset literals-are-in-$\mathcal{L}_{in}$-trail-Cons*)

lemma *literals-are-in-$\mathcal{L}_{in}$-in-mset-$\mathcal{L}_{all}$*:
  ‹*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}$ $C \Longrightarrow L \in\# C \Longrightarrow L \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}$›
  unfolding *literals-are-in-$\mathcal{L}_{in}$-def*
  by (*auto dest*!: *multi-member-split simp*: *all-lits-of-m-add-mset*)

lemma *literals-are-in-$\mathcal{L}_{in}$-in-$\mathcal{L}_{all}$*:
  assumes
    N1: ‹*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}$ (*mset xs*)› **and**
    i-xs: ‹$i <$ *length xs*›
  shows ‹*xs* ! $i \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}$›
  using *literals-are-in-$\mathcal{L}_{in}$-in-mset-$\mathcal{L}_{all}$*[*of* $\mathcal{A}$ ‹*mset xs*› ‹*xs*!*i*›] *assms* **by** *auto*

lemma *is-$\mathcal{L}_{all}$-$\mathcal{L}_{all}$-rewrite*[*simp*]:
  ‹*is-$\mathcal{L}_{all}$* $\mathcal{A}$ (*all-lits-of-mm* $\mathcal{A}'$) $\Longrightarrow$
    set-mset ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits-of-mm* $\mathcal{A}'$)) = *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$)›
  using *in-all-lits-of-mm-ain-atms-of-iff*
  unfolding *set-mset-set-mset-eq-iff is-$\mathcal{L}_{all}$-def Ball-def in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*
    in-all-lits-of-mm-ain-atms-of-iff atms-of-$\mathcal{L}_{all}$-$\mathcal{A}_{in}$
  by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff*)

lemma *is-$\mathcal{L}_{all}$-alt-def*: ‹*is-$\mathcal{L}_{all}$* $\mathcal{A}$ (*all-lits-of-mm* A) $\longleftrightarrow$ *atms-of* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *atms-of-mm* A›
  unfolding *set-mset-set-mset-eq-iff is-$\mathcal{L}_{all}$-def Ball-def in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff*
    in-all-lits-of-mm-ain-atms-of-iff
  by *auto* (*metis literal.sel*(*2*))+

lemma *in-$\mathcal{L}_{all}$-atm-of-$\mathcal{A}_{in}$*: ‹$L \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}_{in} \longleftrightarrow$ *atm-of* $L \in\#$ $\mathcal{A}_{in}$›
  by (*cases* L) (*auto simp*: $\mathcal{L}_{all}$*-def*)

lemma *literals-are-in-$\mathcal{L}_{in}$-alt-def*:
  ‹*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}$ $S \longleftrightarrow$ *atms-of* $S \subseteq$ *atms-of* ($\mathcal{L}_{all}$ $\mathcal{A}$)›
  apply (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-def all-lits-of-mm-union lits-of-def*
      in-all-lits-of-m-ain-atms-of-iff in-all-lits-of-mm-ain-atms-of-iff atms-of-$\mathcal{L}_{all}$-$\mathcal{A}_{in}$
      atm-of-eq-atm-of uminus-$\mathcal{A}_{in}$-iff subset-iff in-$\mathcal{L}_{all}$-atm-of-$\mathcal{A}_{in}$)
  apply (*auto simp*: *atms-of-def*)
  done

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-atm-of*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}_{in}$ $M \longleftrightarrow$ *atm-of ‘ lits-of-l* $M \subseteq$ *set-mset* $\mathcal{A}_{in}$›
apply (*rule iffI*)
subgoal by (*auto dest*: *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l-atms*)
subgoal by (*fastforce simp*: *literals-are-in-$\mathcal{L}_{in}$-trail-def lits-of-def in-$\mathcal{L}_{all}$-atm-of-$\mathcal{A}_{in}$*)
done

lemma *literals-are-in-$\mathcal{L}_{in}$-poss-remdups-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}_{in}$ (*poss* (*remdups-mset* (*atm-of ‘# C*))) $\longleftrightarrow$ *literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}_{in}$ *C*›
by (*induction C*)
(*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-add-mset in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff atm-of-eq-atm-of*
dest!: *multi-member-split*)

lemma *literals-are-in-$\mathcal{L}_{in}$-negs-remdups-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}_{in}$ (*negs* (*remdups-mset* (*atm-of ‘# C*))) $\longleftrightarrow$ *literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}_{in}$ *C*›
by (*induction C*)
(*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-add-mset in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff atm-of-eq-atm-of*
dest!: *multi-member-split*)

lemma *$\mathcal{L}_{all}$-atm-of-all-lits-of-m*:
‹*set-mset* (*$\mathcal{L}_{all}$* (*atm-of ‘# all-lits-of-m C*)) = *set-mset C* ∪ *uminus ‘ set-mset C*›
supply *lit-eq-Neg-Pos-iff*[*iff*]
by (*auto simp*: *$\mathcal{L}_{all}$-def all-lits-of-m-def image-iff dest*!: *multi-member-split*)

lemma *atm-of-all-lits-of-mm*:
‹*set-mset* (*atm-of ‘# all-lits-of-mm bw*) = *atms-of-mm bw*›
‹*atm-of ‘ set-mset* (*all-lits-of-mm bw*) = *atms-of-mm bw*›
using *in-all-lits-of-mm-ain-atms-of-iff* **apply** (*auto simp*: *image-iff*)
by (*metis* (*full-types*) *image-eqI literal.sel*(*1*))+

lemma *$\mathcal{L}_{all}$-union*:
‹*set-mset* (*$\mathcal{L}_{all}$* (*A* + *B*)) = *set-mset* (*$\mathcal{L}_{all}$ A*) ∪ *set-mset* (*$\mathcal{L}_{all}$ B*)›
by (*auto simp*: *$\mathcal{L}_{all}$-def*)

lemma *$\mathcal{L}_{all}$-cong*:
‹*set-mset A* = *set-mset B* ⟹ *set-mset* (*$\mathcal{L}_{all}$ A*) = *set-mset* (*$\mathcal{L}_{all}$ B*)›
by (*auto simp*: *$\mathcal{L}_{all}$-def*)

lemma *atms-of-$\mathcal{L}_{all}$-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *atms-of* (*$\mathcal{L}_{all}$* $\mathcal{A}$) = *atms-of* (*$\mathcal{L}_{all}$* $\mathcal{B}$)›
unfolding *$\mathcal{L}_{all}$-def*
by *auto*

lemma *is-$\mathcal{L}_{all}$-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *is-$\mathcal{L}_{all}$* $\mathcal{A}$ = *is-$\mathcal{L}_{all}$* $\mathcal{B}$›
unfolding *$\mathcal{L}_{all}$-def is-$\mathcal{L}_{all}$-def*
by *auto*

The definition is here to be shared later.

definition *get-propagation-reason* :: ‹('*v*, '*mark*) *ann-lits* ⇒ '*v literal* ⇒ '*mark option nres*› **where**
‹*get-propagation-reason M L* = *SPEC*(λ*C*. *C* ≠ *None* ⟶ *Propagated L* (*the C*) ∈ *set M*)›

end
theory *Watched-Literals-Algorithm*

imports
More-Sepref.WB-More-Refinement
Watched-Literals-Transition-System
Watched-Literals-All-Literals
begin

Chapter 4

First Refinement: Deterministic Rule Application

4.1 Unit Propagation Loops

definition *set-conflict-pre* :: ‹*'v twl-cls ⇒ 'v twl-st ⇒ bool*› **where**
‹*set-conflict-pre C S ⟷*
(∃ *L C'*. *let S'* = (*set-clauses-to-update* (*add-mset* (*L*, *C'*) (*clauses-to-update S*)) *S*) *in*
twl-struct-invs S' ∧ *twl-stgy-invs S'* ∧ *get-trail S'* ⊨*as CNot* (*clause C*))›

definition *set-conflicting* :: ‹*'v twl-cls ⇒ 'v twl-st ⇒ 'v twl-st*› **where**
‹*set-conflicting* = (λ*C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*).
(*M*, *N*, *U*, *Some* (*clause C*), *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›

definition *mop-set-conflicting* :: ‹*'v twl-cls ⇒ 'v twl-st ⇒ 'v twl-st nres*› **where**
‹*mop-set-conflicting* = (λ*C S*. *do* {
ASSERT (*set-conflict-pre C S*);
RETURN (*set-conflicting C S*)})›

definition *propagate-lit-pre* :: ‹*'v literal ⇒ 'v twl-cls ⇒ 'v twl-st ⇒ bool*› **where**
‹*propagate-lit-pre L' C S ⟷*
(∃ *L C'*. *let S'* = (*set-clauses-to-update* (*add-mset* (*L*, *C'*) (*clauses-to-update S*)) *S*) *in*
twl-struct-invs S' ∧ *twl-stgy-invs S'* ∧ *undefined-lit* (*get-trail S*) *L'* ∧
L' ∈# *all-lits-of-mm* (*clauses* (*get-clauses S*) + *unit-clss S*))›

definition *propagate-lit* :: ‹*'v literal ⇒ 'v twl-cls ⇒ 'v twl-st ⇒ 'v twl-st*› **where**
‹*propagate-lit* = (λ*L' C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
(*Propagated L'* (*clause C*) # *M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* (−*L'*) *Q*)})›

definition *mop-propagate-lit* :: ‹*'v literal ⇒ 'v twl-cls ⇒ 'v twl-st ⇒ 'v twl-st nres*› **where**
‹*mop-propagate-lit* = (λ*L' C S*. *do* {
ASSERT(*propagate-lit-pre L' C S*);
RETURN (*propagate-lit L' C S*)})›

definition *update-clauseS-pre* :: ‹*'v literal ⇒ 'v twl-cls ⇒ 'v twl-st ⇒ bool*› **where**
‹*update-clauseS-pre L C S ⟷*
(*let S* = (*set-clauses-to-update* (*add-mset* (*L*, *C*) (*clauses-to-update S*)) *S*) *in*
L ∈# *watched C* ∧ *C* ∈# *get-clauses S* ∧ *twl-struct-invs S* ∧ *twl-stgy-invs S*)›

definition *update-clauseS* :: ‹*'v literal ⇒ 'v twl-cls ⇒ 'v twl-st ⇒ 'v twl-st nres*› **where**
‹*update-clauseS* = (λ*L C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {

```
      ASSERT(update-clauseS-pre L C (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q));
      K ← SPEC (λL. L ∈# unwatched C ∧ −L ∉ lits-of-l M);
      if K ∈ lits-of-l M
      then RETURN (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q)
      else do {
        (N', U') ← SPEC (λ(N', U'). update-clauses (N, U) C L K (N', U'));
        RETURN (M, N', U', D, NE, UE, NS, US, N0, U0, WS, Q)
      }
  })›

definition all-lits-of-st :: ‹'v twl-st ⇒ 'v literal multiset› where
‹all-lits-of-st S ≡ all-lits-of-mm (clauses (get-clauses S) + unit-clss S + subsumed-clauses S +
    get-all-clauses0 S)›

definition mop-lit-is-pos where
  ‹mop-lit-is-pos L S = do {
     ASSERT(L ∈# all-lits-of-st S ∧
       no-dup (get-trail S));
     RETURN (L ∈ lits-of-l (get-trail S))
  }›

definition unit-propagation-inner-loop-body :: ‹'v literal ⇒ 'v twl-cls ⇒
  'v twl-st ⇒ 'v twl-st nres› where
  ‹unit-propagation-inner-loop-body = (λL C S. do {
    do {
      bL' ← SPEC (λK. K ∈# clause C);
      val-bL' ← mop-lit-is-pos bL' S;
      if val-bL'
      then RETURN S
      else do {
        L' ← SPEC (λK. K ∈# watched C − {#L#});
        ASSERT (watched C = {#L, L'#});
        val-L' ← mop-lit-is-pos L' S;
        if val-L'
        then RETURN S
        else
          if ∀ L ∈# unwatched C. −L ∈ lits-of-l (get-trail S)
          then
            if −L' ∈ lits-of-l (get-trail S)
            then do {mop-set-conflicting C S}
            else do {mop-propagate-lit L' C S}
          else do {
            update-clauseS L C S
          }
      }
    }
  })
›

definition unit-propagation-inner-loop :: ‹'v twl-st ⇒ 'v twl-st nres› where
  ‹unit-propagation-inner-loop S₀ = do {
    n ← SPEC(λ-::nat. True);
   (S, -) ← WHILE_T^λ(S, n). twl-struct-invs S ∧ twl-stgy-invs S ∧ cdcl-twl-cp** S₀ S ∧          (clauses-to-update S ≠ {#} ∨ n
      (λ(S, n). clauses-to-update S ≠ {#} ∨ n > 0)
      (λ(S, n). do {
        b ← SPEC(λb. (b ⟶ n > 0) ∧ (¬b ⟶ clauses-to-update S ≠ {#}));
```

```
      if ¬b then do {
        ASSERT(clauses-to-update S ≠ {#});
        (L, C) ← SPEC (λC. C ∈# clauses-to-update S);
        let S' = set-clauses-to-update (clauses-to-update S − {#(L, C)#}) S;
        T ← unit-propagation-inner-loop-body L C S';
        RETURN (T, if get-conflict T = None then n else 0)
      } else do { This branch allows us to do skip some clauses.
        RETURN (S, n − 1)
      }
    })
    (S0, n);
  RETURN S
 }
›

lemma unit-propagation-inner-loop-body:
 fixes S :: ‹'v twl-st›
 assumes
  ‹clauses-to-update S ≠ {#}› and
  x-WS: ‹(L, C) ∈# clauses-to-update S› and
  inv: ‹twl-struct-invs S› and
  inv-s: ‹twl-stgy-invs S› and
  confl: ‹get-conflict S = None›
 shows
  ‹unit-propagation-inner-loop-body L C
      (set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S)) S)
    ≤ (SPEC (λT'.  twl-struct-invs T' ∧ twl-stgy-invs T' ∧ cdcl-twl-cp** S T' ∧
      (T', S) ∈ measure (size ∘ clauses-to-update)))› (is ?spec is ‹- ≤ ?R›) and
  ‹nofail (unit-propagation-inner-loop-body L C
    (set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S)) S))› (is ?fail)
proof −
 obtain M N U D NE UE NS US N0 U0 WS Q where
  S: ‹S = (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q)›
  by (cases S) auto

 have ‹C ∈# N + U› and struct: ‹struct-wf-twl-cls C› and L-C: ‹L ∈# watched C› and
  uL-M: ‹−L ∈ lits-of-l M›
  using inv multi-member-split[OF x-WS]
  unfolding twl-struct-invs-def twl-st-inv.simps S
  by force+

 have uL: ‹− L ∈ lits-of-l M› and
  all-struct: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S)›
  using inv x-WS unfolding twl-struct-invs-def S pcdcl-all-struct-invs-def by auto

 have ‹cdcl_W-restart-mset.no-strange-atm (state_W-of S)› and
  ‹cdcl_W-restart-mset.cdcl_W-M-level-inv (state_W-of S)›
  using all-struct unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def by fast+
 then have alien-L: ‹L ∈# all-lits-of-mm (clauses N + clauses U + (NE + NS + UE + US + N0
+ U0))› and
  n-d: ‹no-dup M›
  using uL unfolding cdcl_W-restart-mset.no-strange-atm-def
   cdcl_W-restart-mset.cdcl_W-M-level-inv-def
  by (auto simp: S cdcl_W-restart-mset-state in-all-lits-of-mm-ain-atms-of-iff)
 have ‹C ∈# N + U ⟹ b ∈# clause C ⟹
  b ∈# all-lits-of-mm (clauses N + clauses U + (NE + NS + UE + US + N0 + U0))› for b C
```

```
  by (auto dest!: multi-member-split simp: all-lits-of-mm-add-mset all-lits-of-m-add-mset)

note [[goals-limit=15]]
show ?spec
  using assms unfolding unit-propagation-inner-loop-body-def update-clause.simps
    mop-lit-is-pos-def nres-monad3
proof (refine-vcg; (unfold prod.inject clauses-to-update.simps set-clauses-to-update.simps
      ball-simps)?;  clarify?; (unfold triv-forall-equality)?)
  fix L' :: ‹'v literal›
  assume
    ‹clauses-to-update S ≠ {#}› and
    twl-inv: ‹twl-struct-invs S›
  have ‹C ∈# N + U› and struct: ‹struct-wf-twl-cls C› and L-C: ‹L ∈# watched C›
    using twl-inv x-WS unfolding twl-struct-invs-def twl-st-inv.simps S by (auto; fail)+

  define WS' where ‹WS' = WS - {#(L, C)#}›
  have WS-WS': ‹WS = add-mset (L, C) WS'›
    using x-WS unfolding WS'-def S by auto

  have D: ‹D = None›
    using confl S by auto

  let ?S' = ‹(M, N, U, None, NE, UE, NS, US, N0, U0, add-mset (L, C) WS', Q)›
  let ?T = ‹(set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S)) S)›
  let ?T' = ‹(M, N, U, None, NE, UE, NS, US, N0, U0, WS', Q)›

  { — blocking literal
    fix K'
    assume
        K': ‹K' ∈# clause C›
    show ‹K' ∈# all-lits-of-st (set-clauses-to-update
                  (remove1-mset (L, C) (clauses-to-update S)) S)›
      using K' ‹C ∈# N + U› by (auto dest!: multi-member-split
          simp: all-lits-of-mm-add-mset all-lits-of-m-add-mset clauses-def S all-lits-of-st-def)

    show ‹no-dup (get-trail
                (set-clauses-to-update
                  (remove1-mset (L, C)
                    (clauses-to-update S))
                  S))›
      using n-d by (auto simp: S)
    assume
        L': ‹K' ∈ lits-of-l (get-trail ?T)›

    have ‹cdcl-twl-cp ?S' ?T'›
      by (rule cdcl-twl-cp.delete-from-working) (use L' K' S in simp-all)

    then have cdcl: ‹cdcl-twl-cp S ?T›
      using L' D by (simp add: S WS-WS')
    show ‹twl-struct-invs ?T›
      using cdcl inv D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-struct-invs)

    show ‹twl-stgy-invs ?T›
      using cdcl inv-s inv D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-stgy-invs)

    show ‹cdcl-twl-cp** S ?T›
```

```
      using D WS-WS' cdcl by auto

    show ‹(?T, S) ∈ measure (size ∘ clauses-to-update)›
      by (simp add: WS'-def[symmetric] WS-WS' S)

  }

  assume L': ‹L' ∈# remove1-mset L (watched C)›
  show watched: ‹watched C = {#L, L'#}›
    by (cases C) (use struct L-C L' in ‹auto simp: size-2-iff›)
  then have L-C': ‹L ∈# clause C› and L'-C': ‹L' ∈# clause C›
    by (cases C; auto; fail)+

    show ‹L' ∈# all-lits-of-st (set-clauses-to-update
                (remove1-mset (L, C) (clauses-to-update S)) S)›
      using L'-C' ‹C ∈# N + U› by (auto dest!: multi-member-split
          simp: all-lits-of-mm-add-mset all-lits-of-m-add-mset clauses-def S all-lits-of-st-def)


  { — if L' ∈ lits-of-l M, then:
    assume L': ‹L' ∈ lits-of-l (get-trail ?T)›

    have ‹cdcl-twl-cp ?S' ?T'›
      by (rule cdcl-twl-cp.delete-from-working) (use L' L'-C' watched S in simp-all)

    then have cdcl: ‹cdcl-twl-cp S ?T›
      using L' watched D by (simp add: S WS-WS')
    show ‹twl-struct-invs ?T›
      using cdcl inv D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-struct-invs)

    show ‹twl-stgy-invs ?T›
      using cdcl inv-s inv D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-stgy-invs)

    show ‹cdcl-twl-cp** S ?T›
      using D WS-WS' cdcl by auto

    show ‹(?T, S) ∈ measure (size ∘ clauses-to-update)›
      by (simp add: WS'-def[symmetric] WS-WS' S)

  }
    — if L' ∈ lits-of-l M, else:
  let ?M = ‹get-trail ?T›
  assume L': ‹L' ∉ lits-of-l ?M›
  {
   { — if ∀ La∈#unwatched C. − La ∈ lits-of-l (get-trail (set-clauses-to-update (remove1-mset (L, C)
(clauses-to-update S)) S)), then
      assume unwatched: ‹∀ L∈#unwatched C. − L ∈ lits-of-l ?M›

      { — if − L' ∈ lits-of-l (get-trail (set-clauses-to-update (remove1-mset (L, C) (clauses-to-update
S)) S)) then
        let ?T' = ‹(M, N, U, Some (clause C), NE, UE, NS, US, N0, U0, {#}, {#})›
        let ?T = ‹set-conflicting C (set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S))
S)›
        assume uL': ‹−L' ∈ lits-of-l ?M›
        have cdcl: ‹cdcl-twl-cp ?S' ?T'›
          by (rule cdcl-twl-cp.conflict) (use uL' L' watched unwatched S in simp-all)
```

then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
using *uL′ L′ watched unwatched* **by** (*simp add*: *set-conflicting-def WS-WS′ S D*)

have ‹*twl-struct-invs ?T*›
using *cdcl inv D* **unfolding** *WS-WS′*
by (*force intro*: *cdcl-twl-cp-twl-struct-invs*)
moreover have ‹*twl-stgy-invs ?T*›
using *cdcl inv inv-s D* **unfolding** *WS-WS′*
by (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)
moreover have ‹*cdcl-twl-cp*** *S ?T*›
using *D WS-WS′ cdcl S* **by** *auto*
moreover have ‹(*?T*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*)›
by (*simp add*: *S WS′-def*[*symmetric*] *WS-WS′ set-conflicting-def*)

moreover have ‹*get-trail* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*))
S) ⊨*as CNot* (*clause C*)›
using *unwatched uL′ watched L′ uL-M* **by** (*cases C*; *auto simp*: *true-annots-true-cls-def-iff-negation-in-model S*)
ultimately show ‹*mop-set-conflicting C* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)
≤ *?R*›
using *assms*
by (*auto simp*: *mop-set-conflicting-def set-conflict-pre-def Let-def S*
intro!: *ASSERT-leI exI*[*of - L*] *exI*[*of - C*])

}

{ — if − *L′* ∈ *lits-of-l M* else
let *?S* = ‹(*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
let *?T′* = ‹(*Propagated L′* (*clause C*) # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS′*, *add-mset* (− *L′*) *Q*)›
let *?S′* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *C*) *WS′*, *Q*)›
let *?T* = ‹*propagate-lit L′ C* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)›
assume *uL′*: ‹− *L′* ∉ *lits-of-l ?M*›

have *undef*: ‹*undefined-lit M L′*›
using *uL′ L′* **by** (*auto simp*: *S defined-lit-map lits-of-def atm-of-eq-atm-of*)

have *cdcl*: ‹*cdcl-twl-cp ?S′ ?T′*›
by (*rule cdcl-twl-cp.propagate*) (*use uL′ L′ undef watched unwatched D S* **in** *simp-all*)
then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
using *uL′ L′ undef watched unwatched D S WS-WS′* **by** (*simp add*: *propagate-lit-def*)

moreover have ‹*twl-struct-invs ?T*›
using *cdcl inv D* **unfolding** *S WS-WS′* **by** (*force intro*: *cdcl-twl-cp-twl-struct-invs*)

moreover have ‹*cdcl-twl-cp*** *S ?T*›
using *cdcl D WS-WS′* **by** *force*
moreover have ‹*twl-stgy-invs ?T*›
using *cdcl inv inv-s D* **unfolding** *S WS-WS′* **by** (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)
moreover have ‹*L′* ∈# *all-lits-of-mm* (*clauses N* + *clauses U* + (*NE* + *UE*))›
using *L′-C′* ‹*C* ∈# *N* + *U*› **by** (*auto simp*: *all-lits-of-mm-add-mset all-lits-of-m-add-mset*
dest!: *multi-member-split*)
ultimately show ‹*mop-propagate-lit L′ C* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*)

```
(clauses-to-update S)) S)
        ≤ ?R›
        using x-WS inv inv-s undef
        by (auto simp: mop-propagate-lit-def propagate-lit-pre-def propagate-lit-def Let-def S
          intro!: ASSERT-leI exI[of - L] exI[of - C] dest: multi-member-split)
    }
  }

  fix La
— if ∀ L∈#unwatched C. − L ∈ lits-of-l M, else
  {
    let ?S = ‹(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q)›
    let ?S' = ‹(M, N, U, None, NE, UE, NS, US, N0, U0, add-mset (L, C) WS', Q)›
    let ?T = ‹set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S)) S›
    fix K M' N' U' D' WS'' NE' UE' Q' N'' U''
    have ‹update-clauseS L C (set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S)) S)
        ≤ SPEC (λS'. twl-struct-invs S' ∧ twl-stgy-invs S' ∧ cdcl-twl-cp** S S' ∧
        (S', S) ∈ measure (size ∘ clauses-to-update))› (is ?upd)
      apply (rewrite at ‹set-clauses-to-update - ⌑› S)
      apply (rewrite at ‹clauses-to-update ⌑› S)
      unfolding update-clauseS-def clauses-to-update.simps set-clauses-to-update.simps
      apply clarify
    proof refine-vcg
      fix x xa a b
      show ‹update-clauseS-pre L C (M, N, U, D, NE, UE, NS, US, N0, U0, remove1-mset (L, C)
WS, Q)›
        using assms L-C ‹C ∈# N + U› unfolding update-clauseS-pre-def S Let-def
        by auto
      assume K: ‹x ∈# unwatched C ∧ − x ∉ lits-of-l M›
      have uL: ‹− L ∈ lits-of-l M›
        using inv unfolding twl-struct-invs-def S WS-WS' by auto
      { — BLIT
        let ?T = ‹(M, N, U, D, NE, UE, NS, US, N0, U0, remove1-mset (L, C) WS, Q)›
        let ?T' = ‹(M, N, U, None, NE, UE, NS, US, N0, U0, WS', Q)›

        assume ‹x ∈ lits-of-l M›
        have uL: ‹− L ∈ lits-of-l M›
          using inv unfolding twl-struct-invs-def S WS-WS' by auto
        have ‹L ∈# clause C› ‹x ∈# clause C›
          using watched K by (cases C; simp; fail)+
        have ‹cdcl-twl-cp ?S' ?T'›
          by (rule cdcl-twl-cp.delete-from-working[OF ‹x ∈# clause C› ‹x ∈ lits-of-l M›])
        then have cdcl: ‹cdcl-twl-cp S ?T›
          by (auto simp: S D WS-WS')

        show ‹twl-struct-invs ?T›
          using cdcl inv D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-struct-invs)


        show ‹twl-stgy-invs ?T›
          using cdcl inv inv-s D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-stgy-invs)
        show ‹cdcl-twl-cp** S ?T›
          using D WS-WS' cdcl by auto
        show ‹(?T, S) ∈ measure (size ∘ clauses-to-update)›
          by (simp add: WS'-def[symmetric] WS-WS' S)
      }
```

```
      assume
        update: ‹case xa of (N', U') ⇒ update-clauses (N, U) C L x (N', U')› and
        [simp]: ‹xa = (a, b)›
      let ?T' = ‹(M, a, b, None, NE, UE, NS, US, N0, U0, WS', Q)›
      let ?T = ‹(M, a, b, D, NE, UE, NS, US, N0, U0, remove1-mset (L, C) WS, Q)›
      have ‹cdcl-twl-cp ?S' ?T'›
        by (rule cdcl-twl-cp.update-clause)
          (use uL L' K update watched S in ‹simp-all add: true-annot-iff-decided-or-true-lit›)
      then have cdcl: ‹cdcl-twl-cp S ?T›
        by (auto simp: S D WS-WS')

      show ‹twl-struct-invs ?T›
        using cdcl inv D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-struct-invs)

      have uL: ‹- L ∈ lits-of-l M›
        using inv unfolding twl-struct-invs-def S WS-WS' by auto

      show ‹twl-stgy-invs ?T›
        using cdcl inv inv-s D unfolding S WS-WS' by (force intro: cdcl-twl-cp-twl-stgy-invs)
      show ‹cdcl-twl-cp** S ?T›
        using D WS-WS' cdcl by auto
      show ‹(?T, S) ∈ measure (size ∘ clauses-to-update)›
        by (simp add: WS'-def[symmetric] WS-WS' S)
    qed
    moreover assume ‹¬?upd›
    ultimately show ‹- La ∈
      lits-of-l (get-trail (set-clauses-to-update (remove1-mset (L, C) (clauses-to-update S)) S))›
      by fast
  }
 }
 qed
 from SPEC-nofail[OF this] show ?fail .
qed

declare unit-propagation-inner-loop-body(1)[THEN order-trans, refine-vcg]

lemma unit-propagation-inner-loop:
 assumes ‹twl-struct-invs S› and inv: ‹twl-stgy-invs S› and ‹get-conflict S = None›
 shows ‹unit-propagation-inner-loop S ≤ SPEC (λS'. twl-struct-invs S' ∧ twl-stgy-invs S' ∧
   cdcl-twl-cp** S S' ∧ clauses-to-update S' = {#})›
 unfolding unit-propagation-inner-loop-def
 apply (refine-vcg WHILEIT-rule[where R = ‹measure (λ(S, n). (size o clauses-to-update) S + n)›])
 subgoal by auto
 subgoal using assms by auto
 subgoal using assms by auto
 subgoal using assms by auto
 subgoal using assms by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by (auto simp add: twl-struct-invs-def)
```

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

declare *unit-propagation-inner-loop*[*THEN order-trans*, *refine-vcg*]

definition *pop-literal-to-update-pre* **where**
‹*pop-literal-to-update-pre S* ⟷
twl-struct-invs S ∧ *twl-stgy-invs S* ∧ *clauses-to-update S* = {#} ∧
literals-to-update S ≠ {#}›

definition *mop-pop-literal-to-update* :: ‹'*v twl-st* ⇒ ('*v literal* × '*v twl-st*) *nres*› **where**
‹*mop-pop-literal-to-update S* = *do* {
ASSERT(*pop-literal-to-update-pre S*);
L ← *SPEC* (λ*L*. *L* ∈# *literals-to-update S*);
let S' = *set-clauses-to-update* {#(*L*, *C*)|*C* ∈# *get-clauses S*. *L* ∈# *watched C*#}
(*set-literals-to-update* (*literals-to-update S* − {#*L*#}) *S*);
RETURN (*L*, *S'*)
}›

definition *unit-propagation-outer-loop* :: ‹'*v twl-st* ⇒ '*v twl-st nres*› **where**
‹*unit-propagation-outer-loop S*$_0$ =
WHILE$_T$λ*S*. *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧ *cdcl-twl-cp*** *S*$_0$ *S* ∧ *clauses-to-update S* = {#}
(λ*S*. *literals-to-update S* ≠ {#})
(λ*S*. *do* {
(*L*, *S'*) ← *mop-pop-literal-to-update S*;
ASSERT(*cdcl-twl-cp S S'*);
unit-propagation-inner-loop S'
})
S$_0$
›

abbreviation *unit-propagation-outer-loop-spec* **where**
‹*unit-propagation-outer-loop-spec S S'* ≡ *twl-struct-invs S'* ∧ *cdcl-twl-cp*** *S S'* ∧
literals-to-update S' = {#} ∧ (∀ *S'a*. ¬ *cdcl-twl-cp S' S'a*) ∧ *twl-stgy-invs S'*›

lemma *unit-propagation-outer-loop*:
assumes ‹*twl-struct-invs S*› **and** ‹*clauses-to-update S* = {#}› **and** *confl*: ‹*get-conflict S* = *None*› **and**
‹*twl-stgy-invs S*›
shows ‹*unit-propagation-outer-loop S* ≤ *SPEC* (λ*S'*. *twl-struct-invs S'* ∧ *cdcl-twl-cp*** *S S'* ∧
literals-to-update S' = {#} ∧ *no-step cdcl-twl-cp S'* ∧ *twl-stgy-invs S'*)›
proof −
have *assert-twl-cp*: ‹*cdcl-twl-cp T T'*› (**is** *?twl*) **and**
assert-twl-struct-invs:
‹*twl-struct-invs T'*›
(**is** ‹*twl-struct-invs ?T'*›) **and**
assert-stgy-invs:

```
    ‹twl-stgy-invs T'› (is ?stgy)
   if
    p: ‹literals-to-update T ≠ {#}› and
    L-T: ‹L ∈# literals-to-update T› and
    invs: ‹twl-struct-invs T ∧ twl-stgy-invs T ∧cdcl-twl-cp** S T ∧ clauses-to-update T = {#}› and
    eq: ‹(L, set-clauses-to-update (Pair L '# {#C ∈# get-clauses T. L ∈# watched C#})
        (set-literals-to-update (remove1-mset L (literals-to-update T)) T)) =
     (L', T')›
    for L T L' T'
 proof −
  from that have
    p: ‹literals-to-update T ≠ {#}› and
    L-T: ‹L ∈# literals-to-update T› and
    struct-invs: ‹twl-struct-invs T› and
    ‹cdcl-twl-cp** S T› and
    w-q: ‹clauses-to-update T = {#}› and
   eq[simp]: ‹L' = L› ‹T' = (set-clauses-to-update (Pair L '# {#Ca ∈# get-clauses T. L ∈# watched
Ca#})
     (set-literals-to-update (remove1-mset L (literals-to-update T)) T))›
    by fast+
  have ‹get-conflict T = None›
    using w-q p invs unfolding twl-struct-invs-def by auto
  then obtain M N U NE UE NS US N0 U0 Q where
    T: ‹T = (M, N, U, None, NE, UE, NS, US, N0, U0, {#}, Q)›
    using w-q p by (cases T) auto
  define Q' where ‹Q' = remove1-mset L Q›
  have Q: ‹Q = add-mset L Q'›
    using L-T unfolding Q'-def T by auto

   — Show assertion that one step has been done
  show twl: ?twl
     unfolding eq T set-clauses-to-update.simps set-literals-to-update.simps literals-to-update.simps
Q'-def[symmetric]
    unfolding Q get-clauses.simps
    by (rule cdcl-twl-cp.pop)
  then show ‹twl-struct-invs ?T'›
    using cdcl-twl-cp-twl-struct-invs struct-invs by blast


  then show ?stgy
    using twl cdcl-twl-cp-twl-stgy-invs[OF twl] invs by blast
 qed

 show ?thesis
  unfolding unit-propagation-outer-loop-def mop-pop-literal-to-update-def nres-monad3
  apply (refine-vcg WHILEIT-rule[where R = ‹{(T, S). twl-struct-invs S ∧ cdcl-twl-cp++ S T}›])
          apply ((simp-all add: assms tranclp-wf-cdcl-twl-cp; fail)+)[6]
  subgoal unfolding pop-literal-to-update-pre-def by blast
  subgoal by (rule assert-twl-cp) — Assertion
  subgoal by (rule assert-twl-struct-invs) — WHILE-loop invariants
  subgoal by (rule assert-stgy-invs)
  subgoal for S L
    by (cases S)
     (auto simp: twl-st twl-struct-invs-def)
  subgoal by (simp; fail)
  subgoal by auto
```

subgoal by *auto*
subgoal by *simp*
subgoal by *auto* — Termination
subgoal — Final invariants
by *simp*
subgoal by *simp*
subgoal by *auto*
subgoal by (*auto simp*: *cdcl-twl-cp.simps*)
subgoal by *simp*
done
qed

declare *unit-propagation-outer-loop*[*THEN order-trans*, *refine-vcg*]

4.2 Other Rules

4.2.1 Decide

definition *find-unassigned-lit* :: ‹*'v twl-st* ⇒ *'v literal option nres*› **where**
‹*find-unassigned-lit* = (λ*S*.
SPEC (λ*L*.
(*L* ≠ *None* ⟶ *undefined-lit* (*get-trail S*) (*the L*) ∧
(*the L*) ∈# *all-lits-of-st S*) ∧
(*L* = *None* ⟶ (∄*L*. *undefined-lit* (*get-trail S*) *L* ∧
L ∈# *all-lits-of-st S*))))›

definition *propagate-dec* **where**
‹*propagate-dec* = (λ*L* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*).
(*Decided L # M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, {#−*L*#}))›

definition *decide-or-skip-pre* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*decide-or-skip-pre S* ⟷
clauses-to-update S = {#} ∧ *literals-to-update S* = {#} ∧ *get-conflict S* = *None* ∧
twl-struct-invs S ∧ *twl-stgy-invs S*›

definition *decide-or-skip* :: ‹*'v twl-st* ⇒ (*bool* × *'v twl-st*) *nres*› **where**
‹*decide-or-skip S* = *do* {
ASSERT(*decide-or-skip-pre S*);
L ← *find-unassigned-lit S*;
case L of
None ⇒ *RETURN* (*True*, *S*)
| *Some L* ⇒ *RETURN* (*False*, *propagate-dec L S*)
}
›

lemma *decide-or-skip-spec*:
assumes ‹*clauses-to-update S* = {#}› **and** ‹*literals-to-update S* = {#}› **and** ‹*get-conflict S* = *None*›
and
twl: ‹*twl-struct-invs S*› **and** *twl-s*: ‹*twl-stgy-invs S*›
shows ‹*decide-or-skip S* ≤ *SPEC*(λ(*brk*, *T*). *cdcl-twl-o*** *S T* ∧
get-conflict T = *None* ∧
no-step cdcl-twl-o T ∧ (*brk* ⟶ *no-step cdcl-twl-stgy T*) ∧ *twl-struct-invs T* ∧
twl-stgy-invs T ∧ *clauses-to-update T* = {#} ∧
(¬*brk* ⟶ *literals-to-update T* ≠ {#}) ∧
(¬*no-step cdcl-twl-o S* ⟶ *cdcl-twl-o*$^{++}$ *S T*))›

proof −
obtain *M N U NE UE NS US N0 U0* **where** *S*: ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *assms* **by** (*cases S*) *auto*
have *atm-N-U*:
‹*atm-of L* ∈ *atms-of-ms* (*clause* ‘ *set-mset U*) ⟹
atm-of L ∈ *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)›
‹*atms-of-mm* (*get-all-clss* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})) = *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)›
for *L*
proof −
have ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of S*)› **and** *unit*: ‹*entailed-clss-inv* (*pstate*$_W$*-of S*)›
using *twl* **unfolding** *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
pcdcl-all-struct-invs-def
by *auto*
then show
‹*atm-of L* ∈ *atms-of-ms* (*clause* ‘ *set-mset U*) ⟹ *atm-of L* ∈ *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)›
‹*atms-of-mm* (*get-all-clss* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})) = *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)›
by (*auto simp*: *cdcl*$_W$*-restart-mset.no-strange-atm-def S cdcl*$_W$*-restart-mset-state image-Un*)
qed
{
fix *L*
assume *undef*: ‹*undefined-lit M L*› **and** *L*: ‹*atm-of L* ∈ *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)›
let *?T* = ‹(*Decided L* # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#− *L*#})›
have *o*: ‹*cdcl-twl-o* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *?T*›
by (*rule cdcl-twl-o.decide*) (*use undef L* **in** *auto*)
have *twl*′: ‹*twl-struct-invs ?T*›
using *S cdcl-twl-o-twl-struct-invs o twl* **by** *blast*
have *twl-s*′: ‹*twl-stgy-invs ?T*›
using *S cdcl-twl-o-twl-stgy-invs o twl twl-s* **by** *blast*
note *o twl*′ *twl-s*′
} **note** *H* = *this*
have *H*′[*unfolded S*, *simp*]:
‹*L* ∈# *all-lits-of-st S* ⟷ *atm-of L* ∈ *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)› **for** *L*
using *atm-N-U*(*2*)
by (*auto simp*: *S all-lits-of-st-def all-lits-def in-all-lits-of-mm-ain-atms-of-iff*)
show *?thesis*
using *assms* **unfolding** *S find-unassigned-lit-def propagate-dec-def decide-or-skip-def*
atm-N-U
apply (*refine-vcg*)
subgoal unfolding *decide-or-skip-pre-def* **by** *blast*
subgoal by *fast*
subgoal by *blast*
subgoal by (*force simp*: *H elim*!: *cdcl-twl-oE cdcl-twl-stgyE cdcl-twl-cpE dest*!: *atm-N-U*(*1*))
subgoal by (*force elim*!: *cdcl-twl-oE cdcl-twl-stgyE cdcl-twl-cpE*)
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by (*auto elim*!: *cdcl-twl-oE*)
subgoal using *atm-N-U*(*1*) *H*′ **by** (*auto simp*: *cdcl-twl-o.simps decide*)
subgoal by *auto*
subgoal by (*auto elim*!: *cdcl-twl-oE*)
subgoal using *atm-N-U H* **by** *auto*

subgoal using *H H' atm-N-U* **by** *auto*
subgoal using *H H' atm-N-U* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *H H' atm-N-U* **by** *auto*
done
qed

declare *decide-or-skip-spec*[*THEN order-trans*, *refine-vcg*]

4.2.2 Skip and Resolve Loop

definition *skip-and-resolve-loop-inv* **where**
‹*skip-and-resolve-loop-inv* S_0 =
(λ(*brk*, *S*). *cdcl-twl-o*** S_0 *S* ∧ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧
clauses-to-update S = {#} ∧ *literals-to-update S* = {#} ∧
get-conflict S ≠ *None* ∧
count-decided (*get-trail S*) ≠ *0* ∧
get-trail S ≠ [] ∧
get-conflict S ≠ *Some* {#} ∧
(*brk* ⟶ *no-step* $cdcl_W$*-restart-mset.skip* ($state_W$*-of S*) ∧
no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of S*)))›

definition *tl-state* :: ‹'*v twl-st* ⇒ (*bool* × '*v twl-st*)› **where**
‹*tl-state* = (λ(*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
(*False*, (*tl M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*))})›

definition *mop-tl-state-pre* :: ‹'*v twl-st* ⇒ *bool*› **where**
‹*mop-tl-state-pre S* ⟷ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧
clauses-to-update S = {#} ∧ *literals-to-update S* = {#} ∧
get-conflict S ≠ *None* ∧
count-decided (*get-trail S*) ≠ *0* ∧
get-trail S ≠ [] ∧
get-conflict S ≠ *Some* {#} ∧
is-proped (*hd* (*get-trail S*)) ∧
lit-of (*hd* (*get-trail S*)) ∈# *all-lits-of-st S* ∧
lit-of (*hd* (*get-trail S*)) ∉# *the* (*get-conflict S*) ∧
−*lit-of* (*hd* (*get-trail S*)) ∉# *the* (*get-conflict S*)›

definition *mop-tl-state* :: ‹'*v twl-st* ⇒ (*bool* × '*v twl-st*) *nres*› **where**
‹*mop-tl-state* = (λ*S*. *do* {
ASSERT(*mop-tl-state-pre S*);
RETURN(*tl-state S*)})›

definition *update-confl-tl-pre* :: ‹'*v literal* ⇒ '*v clause* ⇒ '*v twl-st* ⇒ *bool*› **where**
‹*update-confl-tl-pre L D S* ⟷ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧
clauses-to-update S = {#} ∧ *literals-to-update S* = {#} ∧
get-conflict S ≠ *None* ∧
count-decided (*get-trail S*) ≠ *0* ∧
get-trail S ≠ [] ∧
get-conflict S ≠ *Some* {#} ∧
L ∈# *D* ∧
−*L* ∈# *the* (*get-conflict S*) ∧
hd (*get-trail S*) = *Propagated L D* ∧
L ∈# *all-lits-of-st S*›

definition *update-confl-tl* :: ‹'v literal ⇒ 'v clause ⇒ 'v twl-st ⇒ (bool × 'v twl-st)› **where**
‹*update-confl-tl* = (λL C (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q).
(False, (tl M, N, U, Some (remove1-mset L (remove1-mset (−L) ((the D) ∪# C))), NE, UE, NS, US, N0, U0, WS, Q)))›

definition *mop-update-confl-tl* :: ‹'v literal ⇒ 'v clause ⇒ 'v twl-st ⇒ (bool × 'v twl-st) nres› **where**
‹*mop-update-confl-tl* = (λL C S. *do* {
ASSERT(*update-confl-tl-pre* L C S);
RETURN (*update-confl-tl* L C S)})›

definition *mop-lit-notin-conflict* :: ‹'v literal ⇒ 'v twl-st ⇒ bool nres› **where**
‹*mop-lit-notin-conflict* L S = *do* {
ASSERT(*get-conflict* S ≠ *None* ∧ −L ∉# *the* (*get-conflict* S) ∧ L ∈# *all-lits-of-st* S);
RETURN (L ∉# *the* (*get-conflict* S))
}›

definition *mop-maximum-level-removed-pre* :: ‹'v literal ⇒ 'v twl-st ⇒ bool› **where**
‹*mop-maximum-level-removed-pre* L S ⟷ *twl-struct-invs* S ∧ *twl-stgy-invs* S ∧
clauses-to-update S = {#} ∧ *literals-to-update* S = {#} ∧
get-conflict S ≠ *None* ∧
count-decided (*get-trail* S) ≠ 0 ∧
get-trail S ≠ [] ∧
get-conflict S ≠ *Some* {#} ∧
−L ∈# *the* (*get-conflict* S) ∧
L = *lit-of* (*hd* (*get-trail* S))›

definition *mop-maximum-level-removed* :: ‹'v literal ⇒ 'v twl-st ⇒ bool nres› **where**
‹*mop-maximum-level-removed* L S = *do* {
ASSERT (*mop-maximum-level-removed-pre* L S);
RETURN (*get-maximum-level* (*get-trail* S) (*remove1-mset* (−L) (*the* (*get-conflict* S))) = *count-decided* (*get-trail* S))
}›

definition *mop-hd-trail-pre* :: ‹'v twl-st ⇒ bool› **where**
‹*mop-hd-trail-pre* S ⟷ *twl-struct-invs* S ∧ *twl-stgy-invs* S ∧
clauses-to-update S = {#} ∧ *literals-to-update* S = {#} ∧
get-conflict S ≠ *None* ∧
get-trail S ≠ [] ∧ *is-proped* (*hd* (*get-trail* S)) ∧
get-conflict S ≠ *Some* {#}›

definition *mop-hd-trail* :: ‹'v twl-st ⇒ ('v literal × 'v clause) nres› **where**
‹*mop-hd-trail* S = *do*{
ASSERT(*mop-hd-trail-pre* S);
SPEC(λ(L, C). *Propagated* L C = *hd* (*get-trail* S))
}›

definition *skip-and-resolve-loop* :: ‹'v twl-st ⇒ 'v twl-st nres› **where**
‹*skip-and-resolve-loop* S_0 =
do {
(-, S) ←
$WHILE_T$*skip-and-resolve-loop-inv* S_0
(λ(uip, S). ¬uip ∧ ¬*is-decided* (*hd* (*get-trail* S)))
(λ(-, S).
do {
(L, C) ← *mop-hd-trail* S;
b ← *mop-lit-notin-conflict* (−L) S;

```
        if b then
          do {mop-tl-state S}
        else do {
            b ← mop-maximum-level-removed L S;
            if b
            then
              do {mop-update-confl-tl L C S}
            else
              do {RETURN (True, S)}}
      }
    )
    (False, S_0);
   RETURN S
  }
›

lemma skip-and-resolve-loop-spec:
  assumes struct-S: ‹twl-struct-invs S› and stgy-S: ‹twl-stgy-invs S› and
    ‹clauses-to-update S = {#}› and ‹literals-to-update S = {#}› and
    ‹get-conflict S ≠ None› and count-dec: ‹count-decided (get-trail S) > 0›
  shows ‹skip-and-resolve-loop S ≤ SPEC(λT. cdcl-twl-o** S T ∧ twl-struct-invs T ∧ twl-stgy-invs T
∧
      no-step cdcl_W-restart-mset.skip (state_W-of T) ∧
      no-step cdcl_W-restart-mset.resolve (state_W-of T) ∧
      get-conflict T ≠ None ∧ clauses-to-update T = {#} ∧ literals-to-update T = {#})›
     (is ‹- ≤ ?R›)
  unfolding skip-and-resolve-loop-def mop-hd-trail-def nres-monad3 mop-lit-notin-conflict-def
    mop-maximum-level-removed-def
proof (refine-vcg WHILEIT-rule[where R = ‹measure (λ(brk, S). Suc (length (get-trail S) − If brk 1
0))›];
      remove-dummy-vars)
  let ?R = ‹(λa b. SPEC
        (λs'. skip-and-resolve-loop-inv S s' ∧
              (s', a, b)
              ∈ measure
                (λ(brk, S).
                    Suc (length (get-trail S) − (if brk then 1 else 0)))))›
  show ‹wf (measure (λ(brk, S). Suc (length (get-trail S) − (if brk then 1 else 0))))›
    by auto

  have neg: ‹get-trail S ⊨as CNot (the (get-conflict S))› if ‹get-conflict S ≠ None›
      using assms that unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
        cdcl_W-restart-mset.cdcl_W-conflicting-def pcdcl-all-struct-invs-def
      by (cases S, auto simp add: cdcl_W-restart-mset-state)
  then have ‹get-trail S ≠ []› if ‹get-conflict S ≠ Some {#}›
    using that assms by auto
  then show ‹skip-and-resolve-loop-inv S (False, S)›
    using assms by (cases S) (auto simp: skip-and-resolve-loop-inv-def cdcl_W-restart-mset.skip.simps
        cdcl_W-restart-mset.resolve.simps cdcl_W-restart-mset-state
        twl-stgy-invs-def cdcl_W-restart-mset.conflict-non-zero-unless-level-0-def)

  fix brk :: bool and T :: ‹'a twl-st›
  assume
    inv: ‹skip-and-resolve-loop-inv S (brk, T)› and
    brk: ‹case (brk, T) of (brk, S) ⇒ ¬ brk ∧ ¬ is-decided (hd (get-trail S))›
  have [simp]: ‹brk = False›
```

```
    using brk by auto
  have M-not-empty: ‹get-trail T ≠ []›
    using brk inv unfolding skip-and-resolve-loop-inv-def by auto
  then show ‹mop-hd-trail-pre T›
    using brk inv unfolding skip-and-resolve-loop-inv-def mop-hd-trail-pre-def
    by (cases ‹get-trail T›; cases ‹hd (get-trail T)›)  auto

  fix L :: ‹'a literal› and C
  assume
    LC: ‹case (L, C) of (L, C) ⇒ Propagated L C = hd (get-trail T)›

  obtain M N U D NE UE NS US N0 U0 WS Q where
    T: ‹T = (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q)›
    by (cases T)


  obtain M' :: ‹('a, 'a clause) ann-lits› and D' where
    M: ‹get-trail T = Propagated L C # M'› and WS: ‹WS = {#}› and Q: ‹Q = {#}› and D: ‹D =
Some D'› and
    st: ‹cdcl-twl-o** S T› and twl: ‹twl-struct-invs T› and D': ‹D' ≠ {#}› and
    twl-stgy-S: ‹twl-stgy-invs T› and
    count-dec[simp]: ‹count-decided (tl M) > 0› ‹count-decided (tl M) ≠ 0› ‹count-decided M ≠ 0›
    using brk inv LC unfolding skip-and-resolve-loop-inv-def
    by (cases ‹get-trail T›; cases ‹hd (get-trail T)›) (auto simp: T)

  show ‹get-conflict T ≠ None›
     by (auto simp: T D)

  have ‹cdcl_W-restart-mset.cdcl_W-conflicting (state_W-of T)› and
       ‹cdcl_W-restart-mset.cdcl_W-M-level-inv (state_W-of T)› and
       alien: ‹cdcl_W-restart-mset.no-strange-atm (state_W-of T)›
    using twl D D' M unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      pcdcl-all-struct-invs-def by auto
  then have ‹M ⊨as CNot D'› and n-d: ‹no-dup M› and
    confl-proped: ‹⋀L mark a b.
       a @ Propagated L mark # b = trail (state_W-of T) ⟶
       b ⊨as CNot (remove1-mset L mark) ∧ L ∈# mark›
   using M unfolding cdcl_W-restart-mset.cdcl_W-conflicting-def cdcl_W-restart-mset.cdcl_W-M-level-inv-def
    by (auto simp add: cdcl_W-restart-mset-state T D)
  then show ‹- (- L) ∉# the (get-conflict T)›
    using M by (auto simp: T D dest!: multi-member-split uminus-lits-of-l-definedD)
  show alien-L: ‹- L  ∈# all-lits-of-st T›
    using alien M
    by (cases T)
      (auto simp: cdcl_W-restart-mset.no-strange-atm-def all-lits-of-st-def
        cdcl_W-restart-mset-state in-all-lits-of-mm-ain-atms-of-iff)

  have LD': ‹L ∉# D'›
    using ‹M ⊨as CNot D'› M n-d
    by (auto dest!: multi-member-split dest: uminus-lits-of-l-definedD simp: T)

  { — skip
    assume LD: ‹- L ∉# the (get-conflict T)›
    let ?T = ‹snd (tl-state T)›
    have o-S-T: ‹cdcl-twl-o T ?T›
      using cdcl-twl-o.skip[of L ‹the D› C M' N U NE UE NS US N0 U0]
```

using *LD D inv M* **unfolding** *skip-and-resolve-loop-inv-def T WS Q D* **by** (*auto simp*: *tl-state-def*)
have *st-T*: ‹*cdcl-twl-o*** *S ?T*›
using *st o-S-T* **by** *auto*
moreover have *twl-T*: ‹*twl-struct-invs ?T*›
using *struct-S twl o-S-T cdcl-twl-o-twl-struct-invs* **by** *blast*
moreover have *twl-stgy-T*: ‹*twl-stgy-invs ?T*›
using *twl o-S-T stgy-S twl-stgy-S cdcl-twl-o-twl-stgy-invs* **by** *blast*
moreover have ‹*tl M* ≠ []›
using *twl-T D D'* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
cdcl$_W$-restart-mset.cdcl$_W$-conflicting-def pcdcl-all-struct-invs-def
by (*auto simp*: *cdcl$_W$-restart-mset-state T tl-state-def*)
moreover have ‹− *lit-of* (*hd M*) ∈# *all-lits-of-st T*› ‹*is-proped* (*hd M*)›
‹− *lit-of* (*hd M*) ∉# *the* (*get-conflict T*)› ‹*lit-of* (*hd M*) ∉# *the* (*get-conflict T*)›
using *M alien-L LD LD' D*
by (*auto intro*!: *ASSERT-leI simp*: *T tl-state-def in-all-lits-of-mm-uminus-iff*)
ultimately show ‹*mop-tl-state T* ≤ *?R brk T*›
using *WS Q D D' twl-stgy-S twl alien-L LD*
unfolding *skip-and-resolve-loop-inv-def mop-tl-state-pre-def mop-tl-state-def all-lits-of-st-def*
by (*auto intro*!: *ASSERT-leI simp*: *T tl-state-def in-all-lits-of-mm-uminus-iff*)
}

{ — resolve
assume
LD: ‹¬− *L* ∉# *the* (*get-conflict T*)›

have *count-dec'*: ‹*count-decided M'* = *count-decided M*›
using *M* **unfolding** *T* **by** *auto*
then show *mop-maximum-level-removed-pre*: ‹*mop-maximum-level-removed-pre L T*›
using *count-dec twl-stgy-S twl D' LD M* **unfolding** *mop-maximum-level-removed-pre-def*
by (*auto simp*: *T D WS Q simp del*: *count-dec*)

assume
max: ‹*get-maximum-level* (*get-trail T*) (*remove1-mset* (−*L*) (*the* (*get-conflict T*)))
= *count-decided* (*get-trail T*)›

let *?D* = ‹*remove1-mset* (− *L*) (*the* (*get-conflict T*)) ∪# *remove1-mset L C*›
let *?T* = ‹*snd* (*update-confl-tl L C T*)›

have ‹*cdcl$_W$-restart-mset.cdcl$_W$-conflicting* (*state$_W$-of T*)›
using *twl D D' M* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
pcdcl-all-struct-invs-def **by** *auto*
then have ‹*L* ∈# *C*› ‹*M'* ⊨*as CNot* (*remove1-mset L C*)› ‹*M* ⊨*as CNot D'*›
using *M*
unfolding *cdcl$_W$-restart-mset.cdcl$_W$-conflicting-def*
by (*force simp*: *T cdcl$_W$-restart-mset-state D*)+
then have ‹−*L* ∉# *C*›
using *n-d M* **by** (*auto simp*: *T add-mset-eq-add-mset Decided-Propagated-in-iff-in-lits-of-l*
dest!: *multi-member-split*)
have ‹*L* ∉# *D'*›
using ‹*M* ⊨*as CNot D'*› *M n-d*
by (*auto simp*: *T Decided-Propagated-in-iff-in-lits-of-l*
dest!: *multi-member-split*)
then have *D-alt*[*simp*]: ‹*D'* ∪# *C* − {#*L*, − *L*#} = *?D*›
‹*the D* ∪# *C* − {#*L*, − *L*#} = *?D*›
using *LD* ‹*L* ∈# *C*› ‹−*L* ∉# *C*›
by (*auto simp*: *T D sup-union-right1 dest*!: *multi-member-split*)

```
    have o-S-T: ‹cdcl-twl-o T ?T›
      using cdcl-twl-o.resolve[of L ‹the D› C M' N U NE UE] LD D max M WS Q D
      by (auto simp: T D update-confl-tl-def)
    then have st-T: ‹cdcl-twl-o** S ?T›
      using st by auto
    moreover have twl-T: ‹twl-struct-invs ?T›
      using st-T twl o-S-T cdcl-twl-o-twl-struct-invs by blast
    moreover have twl-stgy-T: ‹twl-stgy-invs ?T›
      using twl o-S-T twl-stgy-S cdcl-twl-o-twl-stgy-invs by blast
    moreover {
      have ‹cdcl_W-restart-mset.cdcl_W-conflicting (state_W-of ?T)›
      using twl-T D D' M unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      pcdcl-all-struct-invs-def by auto
      then have ‹tl M ⊨as CNot ?D›
        using M unfolding cdcl_W-restart-mset.cdcl_W-conflicting-def
        by (auto simp add: cdcl_W-restart-mset-state T update-confl-tl-def)
    }
    moreover have ‹get-conflict ?T ≠ Some {#}›
      using twl-stgy-T count-dec unfolding twl-stgy-invs-def update-confl-tl-def
        cdcl_W-restart-mset.conflict-non-zero-unless-level-0-def T
        by (auto simp: trail.simps conflicting.simps)
    moreover have ‹update-confl-tl-pre L C T›
      using WS Q D D' mop-maximum-level-removed-pre twl-stgy-T twl twl-stgy-S M LD count-dec
        confl-proped[of ‹[]› L C ‹M'›] alien-L
      unfolding skip-and-resolve-loop-inv-def
     by (auto simp add: cdcl_W-restart-mset.skip.simps cdcl_W-restart-mset.resolve.simps all-lits-of-st-def
        cdcl_W-restart-mset-state update-confl-tl-def T update-confl-tl-pre-def in-all-lits-of-mm-uminus-iff)
    ultimately show ‹mop-update-confl-tl L C T ≤ ?R brk T›
    using WS Q D D' mop-maximum-level-removed-pre M count-dec unfolding skip-and-resolve-loop-inv-def
      by (auto simp add: cdcl_W-restart-mset.skip.simps cdcl_W-restart-mset.resolve.simps
          cdcl_W-restart-mset-state update-confl-tl-def T mop-update-confl-tl-def)
  }
  { — No step
    assume
      LD: ‹¬- L ∉# the (get-conflict T)› and
      max: ‹get-maximum-level (get-trail T) (remove1-mset (- L) (the (get-conflict T)))
        ≠ count-decided (get-trail T)›

    show ‹skip-and-resolve-loop-inv S (True, T)›
      using inv max LD D M unfolding skip-and-resolve-loop-inv-def
      by (auto simp add: cdcl_W-restart-mset.skip.simps cdcl_W-restart-mset.resolve.simps
          cdcl_W-restart-mset-state T)
    show ‹((True, T), (brk, T)) ∈ measure (λ(brk, S). Suc (length (get-trail S) - (if brk then 1 else
0)))›
      using M-not-empty by simp
  }
next — Final properties
  fix brk T U
  assume
    inv: ‹skip-and-resolve-loop-inv S (brk, T)› and
    brk: ‹¬(case (brk, T) of (brk, S) ⇒ ¬ brk ∧ ¬ is-decided (hd (get-trail S)))›
  show ‹cdcl-twl-o** S T›
    using inv by (auto simp add: skip-and-resolve-loop-inv-def)

  { assume ‹is-decided (hd (get-trail T))›
    then have ‹no-step cdcl_W-restart-mset.skip (state_W-of T)› and
```

```
      ‹no-step cdcl_W-restart-mset.resolve (state_W-of T)›
    by (cases T;  auto simp add: cdcl_W-restart-mset.skip.simps
      cdcl_W-restart-mset.resolve.simps cdcl_W-restart-mset-state)+
  }
  moreover
  { assume ‹brk›
    then have ‹no-step cdcl_W-restart-mset.skip (state_W-of T)› and
      ‹no-step cdcl_W-restart-mset.resolve (state_W-of T)›
      using inv by (auto simp: skip-and-resolve-loop-inv-def)
  }
  ultimately show ‹¬ cdcl_W-restart-mset.skip (state_W-of T) U› and
    ‹¬ cdcl_W-restart-mset.resolve (state_W-of T) U›
    using brk unfolding prod.case by blast+

  show ‹twl-struct-invs T›
    using inv unfolding skip-and-resolve-loop-inv-def by auto
  show ‹twl-stgy-invs T›
    using inv unfolding skip-and-resolve-loop-inv-def by auto

  show ‹get-conflict T ≠ None›
    using inv by (auto simp: skip-and-resolve-loop-inv-def)

  show ‹clauses-to-update T = {#}›
    using inv by (auto simp: skip-and-resolve-loop-inv-def)

  show ‹literals-to-update T = {#}›
    using inv by (auto simp: skip-and-resolve-loop-inv-def)
qed

declare skip-and-resolve-loop-spec[THEN order-trans, refine-vcg]
```

4.2.3 Backtrack

```
definition extract-shorter-conflict-pre :: ‹'v twl-st ⇒ bool› where
  ‹extract-shorter-conflict-pre S ⟷ twl-struct-invs S ∧ twl-stgy-invs S ∧
      clauses-to-update S = {#} ∧ literals-to-update S = {#} ∧ get-trail S ≠ []›

definition extract-shorter-conflict :: ‹'v twl-st ⇒ 'v twl-st nres› where
  ‹extract-shorter-conflict = (λ(M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q). do {
    ASSERT(extract-shorter-conflict-pre (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q));
    SPEC(λS'. ∃D'. S' = (M, N, U, Some D', NE, UE, NS, US, N0, U0, WS, Q) ∧
      D' ⊆# the D ∧ clause `# (N + U) + NE + NS + UE + US + N0 + U0 ⊨pm D' ∧ -lit-of (hd
M) ∈# D')
  })›

fun equality-except-conflict :: ‹'v twl-st ⇒ 'v twl-st ⇒ bool› where
‹equality-except-conflict (M, N, U, D, NE, UE, NS, US, N0, U0, WS, Q)
    (M', N', U', D', NE', UE', NS', US', N0', U0', WS', Q') ⟷
  M = M' ∧ N = N' ∧ U = U' ∧ NE = NE' ∧ UE = UE' ∧ NS = NS' ∧ US = US' ∧ N0 = N0' ∧
U0 = U0' ∧
      WS = WS' ∧ Q = Q'›

lemma extract-shorter-conflict-alt-def:
  ‹extract-shorter-conflict S = do {
   ASSERT(extract-shorter-conflict-pre S);
    SPEC(λS'. ∃D'. equality-except-conflict S S' ∧ Some D' = get-conflict S' ∧
```

$D' \subseteq\# \textit{ the } (\textit{get-conflict } S) \wedge \textit{ clause } `\# (\textit{get-clauses } S) + \textit{unit-clss } S + \textit{subsumed-clauses } S + \textit{get-all-clauses0 } S \models pm\ D' \wedge$
$-\textit{lit-of } (\textit{hd } (\textit{get-trail } S)) \in\# D')\}\rangle$
unfolding *extract-shorter-conflict-def*
by (*cases S*) (*auto simp*: *ac-simps intro*!: *bind-cong*[*OF refl*])

definition *reduce-trail-bt* :: ‹'*v literal* ⇒ '*v twl-st* ⇒ '*v twl-st nres*› **where**
‹*reduce-trail-bt* = (λ*L* (*M*, *N*, *U*, *D'*, *NE*, *UE*, *WS*, *Q*). *do* {
M1 ← *SPEC*(λ*M1*. ∃ *K M2*. (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
get-level M K = *get-maximum-level M* (*the D'* − {#−*L*#}) + *1*);
RETURN (*M1*, *N*, *U*, *D'*, *NE*, *UE*, *WS*, *Q*)
})›

definition *propagate-bt-pre* :: ‹'*v literal* ⇒ '*v literal* ⇒ '*v twl-st* ⇒ *bool*› **where**
‹*propagate-bt-pre L L' S* ⟷ (∃ *M N U D NE UE NS US WS Q M' D'*.
S = (*M*, *N*, *U*, *Some D*, *NE*, *UE*,*NS*, *US*, *WS*, *Q*) ∧
twl-stgy-invs (*M'* @ *M*, *N*, *U*, *Some D'*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) ∧
twl-struct-invs (*M'* @ *M*, *N*, *U*, *Some D'*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) ∧
D ⊆# *D'* ∧
−*L* ∈# *D* ∧
get-conflict S ≠ *None* ∧
L' ∈# *D* − {#−*L*#} ∧
L ≠ −*L'* ∧
get-level (*M*) *L'* = *get-maximum-level* (*M*) (*D* − {#−*L*#}) ∧
undefined-lit M L ∧
L ∈# *all-lits-of-st* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*)∧
L' ∈# *all-lits-of-st* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) ∧
(*set-mset* (*all-lits-of-m D*) ⊆
set-mset (*all-lits-of-st* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*))))›

definition *propagate-bt* :: ‹'*v literal* ⇒ '*v literal* ⇒ '*v twl-st* ⇒ '*v twl-st*› **where**
‹*propagate-bt* = (λ*L L'* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
(*Propagated* (−*L*) (*the D*) # *M*, *N*, *add-mset* (*TWL-Clause* {#−*L*, *L'*#} (*the D* − {#−*L*, *L'*#}))
U,
None, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, {#*L*#})
})›

definition *mop-propagate-bt* :: ‹'*v literal* ⇒ '*v literal* ⇒ '*v twl-st* ⇒ '*v twl-st nres*› **where**
‹*mop-propagate-bt* = (λ*L L' S*. *do* {
ASSERT(*propagate-bt-pre L L' S*);
RETURN (*propagate-bt L L' S*)
})›

definition *propagate-unit-bt-pre* :: ‹'*v literal* ⇒ '*v twl-st* ⇒ *bool*› **where**
‹*propagate-unit-bt-pre L S* ⟷ (∃ *M N U NE UE NS US N0 U0 WS Q M' D'*.
S = (*M*, *N*, *U*, *Some* {#−*L*#}, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ∧
twl-stgy-invs (*M'* @ *M*, *N*, *U*, *Some D'*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ∧
twl-struct-invs (*M'* @ *M*, *N*, *U*, *Some D'*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ∧
{#−*L*#} ⊆# *D'* ∧
undefined-lit M L ∧
L ∈# *all-lits-of-st* (*M*, *N*, *U*, *Some D'*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*))›

definition *propagate-unit-bt* :: ‹'*v literal* ⇒ '*v twl-st* ⇒ '*v twl-st*› **where**
‹*propagate-unit-bt* = (λ*L* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*).
(*Propagated* (−*L*) (*the D*) # *M*, *N*, *U*, *None*, *NE*, *add-mset* (*the D*) *UE*, *NS*, *US*, *N0*, *U0*, *WS*,

{#L#}))›

definition *mop-propagate-unit-bt* :: ‹'v literal ⇒ 'v twl-st ⇒ 'v twl-st nres› **where**
‹*mop-propagate-unit-bt* = (λL S. *do* {
ASSERT (*propagate-unit-bt-pre* L S);
RETURN (*propagate-unit-bt* L S)
})›

definition *mop-lit-hd-trail-pre* :: ‹'v twl-st ⇒ bool› **where**
‹*mop-lit-hd-trail-pre* S ⟷ *twl-struct-invs* S ∧ *twl-stgy-invs* S ∧
clauses-to-update S = {#} ∧ *literals-to-update* S = {#} ∧
get-conflict S ≠ *None* ∧
get-trail S ≠ [] ∧
get-conflict S ≠ *Some* {#}›

definition *mop-lit-hd-trail* :: ‹'v twl-st ⇒ ('v literal) nres› **where**
‹*mop-lit-hd-trail* S = *do* {
ASSERT(*mop-lit-hd-trail-pre* S);
SPEC(λL. L = *lit-of* (*hd* (*get-trail* S)))
}›

definition *backtrack-inv* **where**
‹*backtrack-inv* S ⟷ *twl-struct-invs* S ∧ *twl-stgy-invs* S ∧ *get-trail* S ≠ [] ∧
get-conflict S ≠ *None* ∧ *get-conflict* S ≠ *Some* {#} ∧ −*lit-of* (*hd* (*get-trail* S)) ∈# *the* (*get-conflict* S) ∧
no-step $cdcl_W$-*restart-mset.skip* ($state_W$-*of* S) ∧ *no-step* $cdcl_W$-*restart-mset.resolve* ($state_W$-*of* S)›

definition *find-lit-of-max-level* **where**
‹*find-lit-of-max-level* = (λS L. *do* {
ASSERT(*distinct-mset* (*the* (*get-conflict* S)) ∧ −L ∈# *the* (*get-conflict* S));
SPEC(λL'. L' ∈# *the* (*get-conflict* S) − {#−L#} ∧
get-level (*get-trail* S) L' = *get-maximum-level* (*get-trail* S) (*the* (*get-conflict* S) − {#−L#}))
})›

definition *backtrack* :: ‹'v twl-st ⇒ 'v twl-st nres› **where**
‹*backtrack* S =
do {
ASSERT(*backtrack-inv* S);
L ← *mop-lit-hd-trail* S;
S ← *extract-shorter-conflict* S;
S ← *reduce-trail-bt* L S;

if size (*the* (*get-conflict* S)) > 1
then do {
L' ← *find-lit-of-max-level* S L;
mop-propagate-bt L L' S
}
else do {
mop-propagate-unit-bt L S
}
}
›

lemma
assumes *confl*: ‹*get-conflict S ≠ None*› ‹*get-conflict S ≠ Some {#}*› **and**
w-q: ‹*clauses-to-update S = {#}*› **and** *p*: ‹*literals-to-update S = {#}*› **and**
ns-s: ‹*no-step cdcl$_W$-restart-mset.skip* (*state$_W$-of S*)› **and**
ns-r: ‹*no-step cdcl$_W$-restart-mset.resolve* (*state$_W$-of S*)› **and**
twl-struct: ‹*twl-struct-invs S*› **and** *twl-stgy*: ‹*twl-stgy-invs S*›
shows
backtrack-spec:
‹*backtrack S ≤ SPEC* (λ*T*. *cdcl-twl-o S T* ∧ *get-conflict T = None* ∧ *no-step cdcl-twl-o T* ∧
twl-struct-invs T ∧ *twl-stgy-invs T* ∧ *clauses-to-update T = {#}* ∧
literals-to-update T ≠ {#})› (**is** *?spec*) **and**
backtrack-nofail:
‹*nofail* (*backtrack S*)› (**is** *?fail*)
proof −
let *?S* = ‹*state$_W$-of S*›
have *inv-s*: ‹*cdcl$_W$-restart-mset.cdcl$_W$-stgy-invariant ?S*› **and**
inv: ‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv ?S*›
using *twl-struct twl-stgy* **unfolding** *pcdcl-all-struct-invs-def twl-struct-invs-def*
twl-stgy-invs-def **by** *auto*
let *?D′* = ‹*the* (*conflicting ?S*)›
have *M-CNot-D′*: ‹*trail ?S* ⊨*as CNot ?D′*› **and**
dist: ‹*distinct-mset ?D′*›
using *inv confl* **unfolding** *cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
cdcl$_W$-restart-mset.cdcl$_W$-conflicting-def
cdcl$_W$-restart-mset.distinct-cdcl$_W$-state-def
by (*cases* ‹*conflicting ?S*›; *cases S*; *auto simp*: *cdcl$_W$-restart-mset-state*; *fail*)+
then have *trail*: ‹*get-trail S ≠* []›
using *confl* **unfolding** *true-annots-true-cls-def-iff-negation-in-model*
by (*cases S*) (*auto simp*: *cdcl$_W$-restart-mset-state*)
have *all-struct*:
‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv* (*state$_W$-of S*)›
using *twl-struct*
by (*auto simp*: *twl-struct-invs-def pcdcl-all-struct-invs-def*)
then have *uL-D*: ‹− *lit-of* (*hd* (*get-trail S*)) ∈# *the* (*get-conflict S*)›
using *cdcl$_W$-restart-mset.no-step-skip-hd-in-conflicting*[*of*
‹*state$_W$-of S*›] *ns-s ns-r confl twl-stgy*
by (*auto simp*: *twl-st twl-stgy-invs-def pcdcl-all-struct-invs-def*)

show *?spec*
unfolding *backtrack-def extract-shorter-conflict-def reduce-trail-bt-def*
mop-lit-hd-trail-def mop-propagate-bt-def mop-propagate-unit-bt-def
find-lit-of-max-level-def
proof (*refine-vcg*; *remove-dummy-vars*; *clarify?*)
show ‹*backtrack-inv S*›
using *uL-D trail confl assms* **unfolding** *backtrack-inv-def* **by** *fast*
show ‹*mop-lit-hd-trail-pre S*›
using *trail confl assms* **unfolding** *mop-lit-hd-trail-pre-def*
by *auto*
fix *M M1 M2* :: ‹('*a*, '*a clause*) *ann-lits*› **and**
N U :: ‹'*a twl-clss*› **and**
D :: ‹'*a clause option*› **and** *D′* :: ‹'*a clause*› **and** *NE UE NS US N0 U0* :: ‹'*a clauses*› **and**
WS :: ‹'*a clauses-to-update*› **and** *Q* :: ‹'*a lit-queue*› **and** *K K′* :: ‹'*a literal*›
let *?S* = ‹(*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
let *?T* = ‹(*M*, *N*, *U*, *Some D′*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
let *?U* = ‹(*M1*, *N*, *U*, *Some D′*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›

let *?MS* = ‹*get-trail ?S*›
let *?MT* = ‹*get-trail ?T*›
assume
S: ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
show ‹*extract-shorter-conflict-pre ?S*›
using *assms trail* **unfolding** *S extract-shorter-conflict-pre-def* **by** *auto*
have *alien*: ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of S*)› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv* (*state*$_W$*-of S*)›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def* **by** *fast+*
then have *alien-L*: ‹*lit-of* (*hd M*) ∈# *all-lits-of-st ?S*›
using *trail* **unfolding** *cdcl*$_W$*-restart-mset.no-strange-atm-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def all-lits-of-st-def*
by (*cases M*) (*auto simp*: *S cdcl*$_W$*-restart-mset-state in-all-lits-of-mm-ain-atms-of-iff*)

assume
D′-D: ‹*D′* ⊆# *the D*› **and**
L-D′: ‹−*lit-of* (*hd M*) ∈# *D′*›
show ‹*distinct-mset* (*the* (*get-conflict ?U*))›
using *dist distinct-mset-mono*[*OF D′-D*] **by** (*auto simp*: *S cdcl*$_W$*-restart-mset-state*)
show ‹− *lit-of* (*hd* (*get-trail ?S*)) ∈# *the* (*get-conflict ?U*)›
using *L-D′* **by** (*auto simp*: *S*)

assume
N-U-NE-UE-D′: ‹*clauses* (*N* + *U*) + *NE* + *NS* + *UE* + *US* + *N0* + *U0* ⊨*pm D′*› **and**
decomp: ‹(*Decided K′* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
lev-K′: ‹*get-level M K′* = *get-maximum-level M* (*remove1-mset* (− *lit-of* (*hd ?MS*))
(*the* (*Some D′*))) + *1*›
have *WS*: ‹*WS* = {#}› **and** *Q*: ‹*Q* = {#}›
using *w-q p* **unfolding** *S* **by** *auto*

have *uL-D*: ‹− *lit-of* (*hd M*) ∈# *the D*›
using *decomp N-U-NE-UE-D′ D′-D L-D′ lev-K′*
unfolding *WS Q*
by *auto*

have *D-Some-the*: ‹*D* = *Some* (*the D*)›
using *confl S* **by** *auto*
let *?S′* = ‹*state*$_W$*-of S*›
have *inv-s*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant ?S′*› **and**
inv: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv ?S′*›
using *twl-struct twl-stgy* **unfolding** *twl-struct-invs-def twl-stgy-invs-def*
pcdcl-all-struct-invs-def **by** *auto*
have *Q*: ‹*Q* = {#}› **and** *WS*: ‹*WS* = {#}›
using *w-q p* **unfolding** *S* **by** *auto*
have *M-CNot-D′*: ‹*M* ⊨*as CNot D′*›
using *M-CNot-D′ S D′-D*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state true-annots-true-cls-def-iff-negation-in-model*)
obtain *L″ M′* **where** *M*: ‹*M* = *L″* # *M′*›
using *trail S* **by** (*cases M*) *auto*
have *D′-empty*: ‹*D′* ≠ {#}›
using *L-D′* **by** *auto*
have *L′-D*: ‹−*lit-of L″* ∈# *D′*›
using *L-D′* **by** (*auto simp*: *cdcl*$_W$*-restart-mset-state M*)
have *lev-inv*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv ?S′*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def* **by** *fast*
then have *n-d*: ‹*no-dup M*› **and** *dec*: ‹*backtrack-lvl ?S′* = *count-decided M*›

using S **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: $cdcl_W$*-restart-mset-state*)
then have *uL''-M*: ‹$-$*lit-of* L'' $\notin$ *lits-of-l* M›
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l* M)
have ‹*get-maximum-level* M (*remove1-mset* ($-$*lit-of* (*hd* M)) D') $<$ *count-decided* M›
proof (*cases* L'')
case (*Decided x1*) **note** $L'' =$ *this*(*1*)
have ‹*distinct-mset* (*the* D)›
using *inv S confl* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def*
by (*auto simp*: $cdcl_W$*-restart-mset-state*)
then have ‹*distinct-mset* D'›
using D'*-D* **by** (*blast intro*: *distinct-mset-mono*)
then have ‹$-$ *x1* $\notin\#$ *remove1-mset* ($-$ *x1*) D'›
using L'*-D* L'' D'*-D* **by** (*auto dest*: *distinct-mem-diff-mset*)
then have *H*: ‹$\forall$ $x\in\#$*remove1-mset* ($-$ *lit-of* (*hd* M)) D'. *undefined-lit* [L''] x›
using L'' *M-CNot-D'* *uL''-M*
by (*fastforce simp*: *atms-of-def atm-of-eq-atm-of M true-annots-true-cls-def-iff-negation-in-model*
dest: *in-diffD*)
have ‹*get-maximum-level* M (*remove1-mset* ($-$ *lit-of* (*hd* M)) D') =
get-maximum-level M' (*remove1-mset* ($-$ *lit-of* (*hd* M)) D')›
using *get-maximum-level-skip-beginning*[*OF H*, *of M'*] M
by *auto*
then show *?thesis*
using *count-decided-ge-get-maximum-level*[*of M'* ‹*remove1-mset* ($-$*lit-of* (*hd* M)) D'›] M L''
by *simp*
next
case (*Propagated L C*) **note** $L'' =$ *this*(*1*)
moreover {
have ‹$\forall$ *L mark a b*. *a* @ *Propagated L mark* # *b* = *trail* (*state*$_W$*-of S*) $\longrightarrow$
b $\models$*as CNot* (*remove1-mset L mark*) $\wedge$ *L* $\in\#$ *mark*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
by *blast*
then have ‹*L* $\in\#$ *C*›
by (*force simp*: *S M* $cdcl_W$*-restart-mset-state* L'') }
moreover have *D-empty*: ‹*the D* $\neq$ {$\#$}›
using D'*-D* D'*-empty* **by** *auto*
moreover have ‹$-$*L* $\in\#$ *the D*›
using *ns-s* L'' *confl D-empty*
by (*force simp*: $cdcl_W$*-restart-mset.skip.simps S M* $cdcl_W$*-restart-mset-state*)
ultimately have ‹*get-maximum-level M* (*remove1-mset* ($-$ *lit-of* (*hd M*)) (*the D*))
$<$ *count-decided M*›
using *ns-r confl count-decided-ge-get-maximum-level*[*of M*
‹*remove1-mset* ($-$*lit-of* (*hd M*)) (*the D*)›]
by (*fastforce simp add*: $cdcl_W$*-restart-mset.resolve.simps S M*
$cdcl_W$*-restart-mset-state*)

moreover have ‹*get-maximum-level M* (*remove1-mset* ($-$ *lit-of* (*hd M*)) D') $\leq$
get-maximum-level M (*remove1-mset* ($-$ *lit-of* (*hd M*)) (*the D*))›
by (*rule get-maximum-level-mono*) (*use* D'*-D* **in** ‹*auto intro*: *mset-le-subtract*›)
ultimately show *?thesis*
by *simp*
qed

then have ‹$\exists$ *K M1 M2*. (*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition M*) $\wedge$

get-level M K = get-maximum-level M (remove1-mset (−lit-of (hd M)) D′) + 1›
using *cdcl$_W$-restart-mset.backtrack-ex-decomp n-d*
by (*auto simp*: *cdcl$_W$-restart-mset-state S*)

define *i* **where** ‹*i = get-maximum-level M (remove1-mset (− lit-of (hd M)) D′)*›

let *?T =* ‹(*Propagated (−lit-of (hd M)) D′ # M1, N*,
add-mset (TWL-Clause {#−lit-of (hd M), K#} (D′ − {#−lit-of (hd M), K#})) U,
None, NE, UE, NS, US, N0, U0, WS, {#lit-of (hd M)#})›
let *?T′ =* ‹(*Propagated (−lit-of (hd M)) D′ # M1, N*,
add-mset (TWL-Clause {#−lit-of (hd M), K#} (D′ − {#−lit-of (hd M), K#})) U,
None, NE, UE, NS, US, N0, U0, WS, {#− (−lit-of (hd M))#})›

have *lev-D′*: ‹*count-decided M = get-maximum-level (L″ # M′) D′*›
using *count-decided-ge-get-maximum-level*[*of M D′*] *L′-D*
get-maximum-level-ge-get-level[*of* ‹*−lit-of L″*› *D′ M*] **unfolding** *M*
by (*auto split*: *if-splits*)

have ‹*the D ≠ {#}*›
using *D′-D D′-empty L′-D* **by** (*auto*)
have *uL′-D*: ‹*lit-of L″ ∉# D′*›
using *L-D′ n-d M-CNot-D′* **by** (*auto simp*: *cdcl$_W$-restart-mset-state M add-mset-eq-add-mset*
Decided-Propagated-in-iff-in-lits-of-l
dest!: *multi-member-split*)

{ — conflict clause > 1 literal
assume *size-D*: ‹*1 < size (the (get-conflict ?U))*› **and**
K-D: ‹*K ∈# remove1-mset (− lit-of (hd ?MS)) (the (get-conflict ?U))*› **and**
lev-K: ‹*get-level (get-trail ?U) K = get-maximum-level (get-trail ?U)*
(remove1-mset (− lit-of (hd (get-trail ?S))) (the (get-conflict ?U)))›
have *neq*: ‹*lit-of (hd M) ≠ − K*›
using *dist K-D D′-D L-D′ distinct-mset-mono*[*OF D′-D*]
by (*auto simp*: *S conflicting.simps M distinct-mset-remove1-All*)
moreover have ‹*undefined-lit M1 (lit-of (hd (get-trail ?S)))*›
using *decomp n-d M*
apply (*auto simp*: *dest*!: *get-all-ann-decomposition-exists-prepend*)
apply (*case-tac c*; *case-tac M2*)
apply *auto*
done
moreover have ‹*lit-of (hd (get-trail ?S)) ∈# all-lits-of-st ?S*›
using *alien-L* **by** (*auto simp*: *M*)
moreover have ‹*K ∈# all-lits-of-st ?S*›
using *alien neq mset-subset-eqD*[*OF D′-D in-diffD*[*OF K-D*,
unfolded get-conflict.simps option.sel]] *assms*(*1*)
unfolding *cdcl$_W$-restart-mset.no-strange-atm-def all-lits-of-st-def all-lits-of-st-def*
by (*auto simp*: *S cdcl$_W$-restart-mset-state in-all-lits-of-mm-ain-atms-of-iff*
dest!: *multi-member-split*)
moreover have ‹*set-mset (all-lits-of-m D′) ⊆ set-mset (all-lits-of-st ?S)*›
using *alien assms*(*1*) *atms-of-subset-mset-mono*[*OF D′-D*]
unfolding *cdcl$_W$-restart-mset.no-strange-atm-def all-lits-of-st-def*
by (*fastforce simp*: *S cdcl$_W$-restart-mset-state in-all-lits-of-mm-ain-atms-of-iff*
in-all-lits-of-m-ain-atms-of-iff)
ultimately show ‹*propagate-bt-pre (lit-of (hd (get-trail ?S))) K ?U*›
using ‹*the D ≠ {#}*› *assms D′-D D′-empty L′-D K-D uL′-D lev-K*
get-all-ann-decomposition-exists-prepend[*OF decomp*] *uL-D*
unfolding *propagate-bt-pre-def S all-lits-of-st-def*

by (*auto simp*: *L-D′ intro*!: *exI*[*of* - ‹*take* (*length M* − *length M1*) *M*›]
exI[*of* - ‹*the D*›])
have ‹∀ *L′* ∈# *D′*. −*L′* ∈ *lits-of-l M*›
using *M-CNot-D′ uL′′-M*
by (*fastforce simp*: *atms-of-def atm-of-eq-atm-of M true-annots-true-cls-def-iff-negation-in-model*
dest: *in-diffD*)
obtain *c* **where** *c*: ‹*M* = *c* @ *M2* @ *Decided K′* # *M1*›
using *get-all-ann-decomposition-exists-prepend*[*OF decomp*] **by** *blast*
have ‹*get-level M K′* = *Suc* (*count-decided M1*)›
using *n-d* **unfolding** *c* **by** *auto*
then have *i*: ‹*i* = *count-decided M1*›
using *lev-K′* **unfolding** *i-def* **by** *auto*
have *lev-M-M1*: ‹∀ *L′* ∈# *D′* − {#−*lit-of* (*hd M*)#}. *get-level M L′* = *get-level M1 L′*›
proof
fix *L′*
assume *L′*: ‹*L′* ∈# *D′* − {#−*lit-of* (*hd M*)#}›
have ‹*get-level M L′* > *count-decided M1*› **if** ‹*defined-lit* (*c* @ *M2* @ *Decided K′* # []) *L′*›
using *get-level-skip-end*[*OF that*, *of M1*] *n-d that get-level-last-decided-ge*[*of* ‹*c* @ *M2*›]
by (*auto simp*: *c*)
moreover have ‹*get-level M L′* ≤ *i*›
using *get-maximum-level-ge-get-level*[*OF L′*, *of M*] **unfolding** *i-def* **by** *auto*
ultimately show ‹*get-level M L′* = *get-level M1 L′*›
using *n-d c L′ i* **by** (*cases* ‹*defined-lit* (*c* @ *M2* @ *Decided K′* # []) *L′*›) *auto*
qed
have ‹*get-level M1* ‘# *remove1-mset* (− *lit-of* (*hd M*)) *D′* = *get-level M* ‘# *remove1-mset* (− *lit-of* (*hd M*)) *D′*›
by (*rule image-mset-cong*) (*use lev-M-M1* **in** *auto*)
then have *max-M1-M1-D*: ‹*get-maximum-level M1* (*remove1-mset* (− *lit-of* (*hd M*)) *D′*) =
get-maximum-level M (*remove1-mset* (− *lit-of* (*hd M*)) *D′*)›
unfolding *get-maximum-level-def* **by** *argo*

have ‹∃ *L′* ∈# *remove1-mset* (−*lit-of* (*hd M*)) *D′*.
get-level M L′ = *get-maximum-level M* (*remove1-mset* (− *lit-of* (*hd M*)) *D′*)›
by (*rule get-maximum-level-exists-lit-of-max-level*)
(*use size-D* **in** ‹*auto simp*: *remove1-mset-empty-iff*›)
have *D′-ne-single*: ‹*D′* ≠ {#− *lit-of* (*hd M*)#}›
using *size-D* **apply** (*cases D′*, *simp*)
apply (*rename-tac L D′′*)
apply (*case-tac D′′*)
by *simp-all*
have *cdcl*: ‹*cdcl-twl-o* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) *?T′*›
unfolding *Q WS option.sel list.sel*
apply (*subst D-Some-the*)
apply (*rule cdcl-twl-o.backtrack-nonunit-clause*[*of* ‹−*lit-of* (*hd M*)› - *K′ M1 M2* - - *i*])
subgoal using *D′-D L-D′* **by** *blast*
subgoal using *L′-D decomp M* **by** *auto*
subgoal using *L′-D decomp M* **by** *auto*
subgoal using *L′-D M lev-D′* **by** *auto*
subgoal using *i lev-D′ i-def* **by** *auto*
subgoal using *lev-K′ i-def* **by** *auto*
subgoal using *D′-ne-single* **.**
subgoal using *D′-D* **.**
subgoal using *N-U-NE-UE-D′* **by** (*auto simp*: *ac-simps*)
subgoal using *L-D′* **.**
subgoal using *K-D* **by** (*auto dest*: *in-diffD*)

subgoal using *lev-K lev-M-M1 K-D* **by** (*simp add*: *i-def max-M1-M1-D*)
done
moreover have ‹*get-conflict ?T' = None ∧ clauses-to-update ?T' = {#} ∧ literals-to-update ?T' ≠ {#}*›
unfolding *WS Q*
using *S cdcl-twl-o-twl-struct-invs twl-struct* **by** (*auto simp*: *propagate-bt-def*)
moreover have ‹(∀ *S'*. ¬ *cdcl-twl-o ?T' S'*)› **by** (*auto simp*: *cdcl-twl-o.simps*)
moreover have ‹*twl-struct-invs ?T'*›
using *S cdcl cdcl-twl-o-twl-struct-invs*[*OF cdcl*] *twl-struct twl-stgy* **by** *auto*
moreover have ‹*twl-stgy-invs ?T'*›
using *S cdcl cdcl-twl-o-twl-stgy-invs*[*OF cdcl*] *twl-struct twl-stgy* **by** *auto*
moreover have ‹*clauses-to-update ?T' = {#}*›
using *WS* **by** (*auto simp*: *propagate-bt-def*)

moreover have *False* **if** ‹*cdcl-twl-o ?T'* (*an*, *ao*, *ap*, *aq*, *ar*, *as*, *at'*, *b*)›
for *an ao ap aq ar as at' b*
using *that* **by** (*auto simp*: *cdcl-twl-o.simps propagate-bt-def*)
ultimately show *cdcl*:
‹*cdcl-twl-o ?S* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*)›
‹*get-conflict* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) = *None*›
‹*twl-struct-invs* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*)›
‹*twl-stgy-invs* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*)›
‹*clauses-to-update* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) = {#}›
‹⋀*S'*. *cdcl-twl-o* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) *S'* ⟹ *False*›
‹*literals-to-update* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) = {#} ⟹ *False*›
unfolding *propagate-bt-def* **by** *auto*
}

{ — conflict clause has 1 literal
assume ‹¬ *1 < size* (*the* (*get-conflict ?U*))›
then have *D'*: ‹*D'* = {#−*lit-of* (*hd M*)#}›
using *L'-D* **by** (*cases D'*) (*auto simp*: *M*)
let *?T* = ‹(*Propagated* (− *lit-of* (*hd M*)) *D'* # *M1*, *N*, *U*, *None*, *NE*, *add-mset D' UE*, *NS*, *US*, *N0*, *U0*, *WS*,
unmark (*hd M*))›
let *?T'* = ‹(*Propagated* (− *lit-of* (*hd M*)) *D'* # *M1*, *N*, *U*, *None*, *NE*, *add-mset D' UE*, *NS*, *US*, *N0*, *U0*, *WS*,
{#− (−*lit-of* (*hd M*))#})›

have *i-0*: ‹*i = 0*›
using *i-def* **by** (*auto simp*: *D'*)

have *cdcl*: ‹*cdcl-twl-o* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) *?T'*›
unfolding *D' option.sel WS Q* **apply** (*subst D-Some-the*)
apply (*rule cdcl-twl-o.backtrack-unit-clause*[*of* - ‹*the D*› *K' M1 M2* - *D' i*])
subgoal using *D'-D D'* **by** *auto*
subgoal using *decomp* **by** *simp*
subgoal by (*simp add*: *M*)
subgoal using *D'* **by** (*auto simp*: *get-maximum-level-add-mset*)
subgoal using *i-def* **by** *simp*
subgoal using *lev-K' i-def*[*symmetric*] **by** *auto*
subgoal using *D'* **.**
subgoal using *D'-D* **.**
subgoal using *N-U-NE-UE-D'* **by** (*auto simp*: *ac-simps*)
done
moreover have ‹*get-conflict ?T' = None*›

```
      by (auto simp add: propagate-unit-bt-def)

    moreover have ‹twl-struct-invs ?T'›
      using S cdcl cdcl-twl-o-twl-struct-invs twl-struct by blast

    moreover have ‹twl-stgy-invs ?T'›
      using S cdcl cdcl-twl-o-twl-stgy-invs twl-struct twl-stgy by blast
    moreover have ‹clauses-to-update ?T' = {#}›
      using WS by (auto simp add: propagate-unit-bt-def)
    ultimately show cdcl:
     ‹cdcl-twl-o ?S (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U)›
     ‹get-conflict (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U) = None›
     ‹twl-struct-invs (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U)›
     ‹twl-stgy-invs (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U)›
     ‹clauses-to-update (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U) = {#}›
    unfolding propagate-unit-bt-def by auto
    show False if ‹literals-to-update (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U) = {#}›
      using that by (auto simp add: propagate-unit-bt-def)
    fix an ao ap aq ar as at' b
    show False if ‹cdcl-twl-o (propagate-unit-bt (lit-of (hd (get-trail ?S))) ?U) (an, ao, ap, aq, ar, as,
at', b)›
      using that by (auto simp: cdcl-twl-o.simps propagate-unit-bt-def)
    have ‹undefined-lit M1 (lit-of (hd (get-trail ?S)))›
      using decomp n-d M
      apply (auto simp: dest!: get-all-ann-decomposition-exists-prepend)
      apply (case-tac c; case-tac M2)
      apply auto
      done
    then show ‹propagate-unit-bt-pre (lit-of (hd (get-trail ?S))) ?U›
      using ‹the D ≠ {#}› assms D'-D D'-empty L'-D uL'-D alien-L cdcl
        get-all-ann-decomposition-exists-prepend[OF decomp] uL-D
      unfolding propagate-unit-bt-pre-def S all-lits-of-st-def
      by (auto simp: L-D' D' intro!: exI[of - ‹take (length M − length M1) M›]
        exI[of - ‹the D›])
  }
 qed
 then show ?fail
  using nofail-simps(2) pwD1 by blast
qed

declare backtrack-spec[THEN order-trans, refine-vcg]
```

4.2.4 Full loop

```
definition cdcl-twl-o-prog-pre :: ‹'v twl-st ⇒ bool› where
 ‹cdcl-twl-o-prog-pre S' ⟷
    no-step cdcl-twl-cp S' ∧ twl-struct-invs S' ∧ twl-stgy-invs S' ∧
     clauses-to-update S' = {#} ∧ literals-to-update S' = {#}›

definition cdcl-twl-o-prog :: ‹'v twl-st ⇒ (bool × 'v twl-st) nres› where
 ‹cdcl-twl-o-prog S =
   do {
     ASSERT(cdcl-twl-o-prog-pre S);
     if get-conflict S = None
     then decide-or-skip S
     else do {
```

```
      if count-decided (get-trail S) > 0
      then do {
        T ← skip-and-resolve-loop S;
        ASSERT(get-conflict T ≠ None ∧ get-conflict T ≠ Some {#});
        U ← backtrack T;
        RETURN (False, U)
      }
      else
        RETURN (True, S)
    }
  }
›
```

setup ‹*map-theory-claset* (*fn ctxt => ctxt delSWrapper* (*split-all-tac*))›
declare *split-paired-All*[*simp del*]

lemma *skip-and-resolve-same-decision-level*:
  assumes ‹*cdcl-twl-o S T*› ‹*get-conflict T ≠ None*›
  shows ‹*count-decided* (*get-trail T*) = *count-decided* (*get-trail S*)›
  using *assms* **by** (*induction rule*: *cdcl-twl-o.induct*) *auto*

lemma *skip-and-resolve-conflict-before*:
  assumes ‹*cdcl-twl-o S T*› ‹*get-conflict T ≠ None*›
  shows ‹*get-conflict S ≠ None*›
  using *assms* **by** (*induction rule*: *cdcl-twl-o.induct*) *auto*

lemma *rtranclp-skip-and-resolve-same-decision-level*:
  ‹*cdcl-twl-o*** *S T* ⟹ *get-conflict S ≠ None* ⟹ *get-conflict T ≠ None* ⟹
    count-decided (*get-trail T*) = *count-decided* (*get-trail S*)›
  apply (*induction rule*: *rtranclp-induct*)
  subgoal by *auto*
  subgoal for *T U*
    using *skip-and-resolve-conflict-before*[*of T U*]
    by (*auto simp*: *skip-and-resolve-same-decision-level*)
  done

lemma *empty-conflict-lvl0*:
  ‹*twl-stgy-invs T* ⟹ *get-conflict T = Some {#}* ⟹ *count-decided* (*get-trail T*) = *0*›
  by (*cases T*) (*auto simp*: *twl-stgy-invs-def cdcl*$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
    trail.simps conflicting.simps)

abbreviation *cdcl-twl-o-prog-spec* **where**
  ‹*cdcl-twl-o-prog-spec S* ≡ λ(*brk*, *T*).
      *cdcl-twl-o*** *S T* ∧
      (*get-conflict T ≠ None* ⟶ *count-decided* (*get-trail T*) = *0*) ∧
      (¬ *brk* ⟶ *get-conflict T = None* ∧ (∀ *S'*. ¬ *cdcl-twl-o T S'*)) ∧
      (*brk* ⟶ *get-conflict T ≠ None* ∨ (∀ *S'*. ¬ *cdcl-twl-stgy T S'*)) ∧
      twl-struct-invs T ∧ *twl-stgy-invs T* ∧ *clauses-to-update T* = {#} ∧
      (¬ *brk* ⟶ *literals-to-update T ≠* {#}) ∧
      (¬*brk* ⟶ ¬ (∀ *S'*. ¬ *cdcl-twl-o S S'*) ⟶ *cdcl-twl-o*$^{++}$ *S T*) ∧
      cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)›

lemma *cdcl-twl-o-prog-spec*:
  assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and**
    ‹*literals-to-update S* = {#}› **and**

ns-cp: ‹*no-step cdcl-twl-cp S*› **and**
ent: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)›
shows
‹*cdcl-twl-o-prog S* ≤ *SPEC*(*cdcl-twl-o-prog-spec S*)›
(**is** ‹- ≤ *?S*›)
proof −
have [*iff*]: ‹¬ *cdcl-twl-cp S T*› **for** *T*
using *ns-cp* **by** *fast*

show *?thesis*
unfolding *cdcl-twl-o-prog-def*
apply (*refine-vcg decide-or-skip-spec*[*THEN order-trans*]; *remove-dummy-vars*)
— initial invariants
subgoal using *assms* **unfolding** *cdcl-twl-o-prog-pre-def* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *simp*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal for *T* **using** *assms empty-conflict-lvl0*[*of T*]
rtranclp-skip-and-resolve-same-decision-level[*of S T*] **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** (*auto elim*!: *cdcl-twl-oE simp*: *image-Un*)
subgoal by (*auto elim*!: *cdcl-twl-stgyE cdcl-twl-oE cdcl-twl-cpE*)
subgoal by (*auto simp*: *rtranclp-unfold elim*!: *cdcl-twl-oE*)
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
done
qed

declare *cdcl-twl-o-prog-spec*[*THEN order-trans*, *refine-vcg*]

4.3 Full Strategy

abbreviation *cdcl-twl-stgy-prog-inv* **where**

```
  ‹cdcl-twl-stgy-prog-inv S₀ ≡ λ(brk, T). twl-struct-invs T ∧ twl-stgy-invs T ∧
      (brk ⟶ final-twl-state T) ∧ cdcl-twl-stgy** S₀ T ∧ clauses-to-update T = {#} ∧
      (¬brk ⟶ get-conflict T = None)›

definition cdcl-twl-stgy-prog :: ‹'v twl-st ⇒ 'v twl-st nres› where
  ‹cdcl-twl-stgy-prog S₀ =
  do {
    do {
      (brk, T) ← WHILE_T^cdcl-twl-stgy-prog-inv S₀
        (λ(brk, -). ¬brk)
        (λ(brk, S).
        do {
          T ← unit-propagation-outer-loop S;
          cdcl-twl-o-prog T
        })
        (False, S₀);
      RETURN T
    }
  }
  ›

lemma wf-cdcl-twl-stgy-measure:
   ‹wf ({((brkT, T), (brkS, S)). twl-struct-invs S ∧ cdcl-twl-stgy^++ S T}
       ∪ {((brkT, T), (brkS, S)). S = T ∧ brkT ∧ ¬brkS})›
  (is ‹wf (?TWL ∪ ?BOOL)›)
proof (rule wf-union-compatible)
  show ‹wf ?TWL›
    using tranclp-wf-cdcl-twl-stgy wf-snd-wf-pair by blast
  show ‹?TWL O ?BOOL ⊆ ?TWL›
    by auto

  show ‹wf ?BOOL›
    unfolding wf-iff-no-infinite-down-chain
  proof clarify
    fix f :: ‹nat ⇒ bool × 'b›
    assume H: ‹∀ i. (f (Suc i), f i) ∈ {((brkT, T), brkS, S). S = T ∧ brkT ∧ ¬ brkS}›
    then have ‹(f (Suc 0), f 0) ∈ {((brkT, T), brkS, S). S = T ∧ brkT ∧ ¬ brkS}› and
      ‹(f (Suc 1), f 1) ∈ {((brkT, T), brkS, S). S = T ∧ brkT ∧ ¬ brkS}›
      by presburger+
    then show False
      by auto
  qed
qed

lemma cdcl-twl-o-final-twl-state:
  assumes
    ‹cdcl-twl-stgy-prog-inv S (brk, T)› and
    ‹case (brk, T) of (brk, -) ⇒ ¬ brk› and
    twl-o: ‹cdcl-twl-o-prog-spec U (True, V)›
  shows ‹final-twl-state V›
proof −
  have ‹cdcl-twl-o** U V› and
    confl-lev: ‹get-conflict V ≠ None ⟶ count-decided (get-trail V) = 0› and
    final: ‹get-conflict V ≠ None ∨ (∀ S'. ¬ cdcl-twl-stgy V S')›
    ‹twl-struct-invs V›
    ‹twl-stgy-invs V›
```

```
    ‹clauses-to-update V = {#}›
    using twl-o
    by force+

  show ?thesis
    unfolding final-twl-state-def
    using confl-lev final
    by auto
qed

lemma cdcl-twl-stgy-in-measure:
  assumes
    twl-stgy: ‹cdcl-twl-stgy-prog-inv S (brk0, T)› and
    brk0: ‹case (brk0, T) of (brk, uu-) ⇒ ¬ brk› and
    twl-o: ‹cdcl-twl-o-prog-spec U V› and
    [simp]: ‹twl-struct-invs U› and
    TU: ‹cdcl-twl-cp** T U› and
    ‹literals-to-update U = {#}›
  shows ‹(V, brk0, T)
        ∈ {((brkT, T), brkS, S). twl-struct-invs S ∧ cdcl-twl-stgy++ S T} ∪
          {((brkT, T), brkS, S). S = T ∧ brkT ∧ ¬ brkS}›
proof −
  have [simp]: ‹twl-struct-invs T›
    using twl-stgy by fast+
  obtain brk' V' where
    V: ‹V = (brk', V')›
    by (cases V)
  have
    UV: ‹cdcl-twl-o** U V'› and
    ‹(get-conflict V' ≠ None ⟶ count-decided (get-trail V') = 0)› and
    not-brk': ‹(¬ brk' ⟶ get-conflict V' = None ∧ (∀ S'. ¬ cdcl-twl-o V' S'))› and
    brk': ‹(brk' ⟶ get-conflict V' ≠ None ∨ (∀ S'. ¬ cdcl-twl-stgy V' S'))› and
    [simp]: ‹twl-struct-invs V'›
    ‹twl-stgy-invs V'›
    ‹clauses-to-update V' = {#}› and
    no-lits-to-upd: ‹(0 < count-decided (get-trail V') ⟶ ¬ brk' ⟶ literals-to-update V' ≠ {#})›
    ‹(¬brk' ⟶ ¬ (∀ S'. ¬ cdcl-twl-o U S') ⟶ cdcl-twl-o++ U V')›
    using twl-o unfolding V
    by fast+
    have ‹cdcl-twl-stgy** T V'›
      using TU UV by (auto dest!: rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD)
    then have TV-or-tranclp-TV: ‹T = V' ∨ cdcl-twl-stgy++ T V'›
      unfolding rtranclp-unfold by auto
    have [simp]: ‹¬ cdcl-twl-stgy++ V' V'›
      using wf-not-refl[OF tranclp-wf-cdcl-twl-stgy, of V'] by auto
    have [simp]: ‹brk0 = False›
      using brk0 by auto

    have ‹brk'› if ‹T = V'›
    proof −
      have ns-TV: ‹¬cdcl-twl-stgy++ T V'›
        using that[symmetric] wf-not-refl[OF tranclp-wf-cdcl-twl-stgy, of T] by auto

      have ns-T-T: ‹¬cdcl-twl-o++ T T›
        using wf-not-refl[OF tranclp-wf-cdcl-twl-o, of T] by auto
      have ‹T = U›
```

```
      by (metis (no-types, opaque-lifting) TU UV ns-TV rtranclp-cdcl-twl-cp-stgyD
          rtranclp-cdcl-twl-o-stgyD rtranclp-tranclp-tranclp rtranclp-unfold)
    show ?thesis
      using assms ‹literals-to-update U = {#}› unfolding V that[symmetric] ‹T = U›[symmetric]
      by (auto simp: ns-T-T)
  qed

  then show ?thesis
    using TV-or-tranclp-TV
    unfolding V
    by auto
qed

lemma cdcl-twl-o-prog-cdcl-twl-stgy:
  assumes
    twl-stgy: ‹cdcl-twl-stgy-prog-inv S (brk, S')› and
    ‹case (brk, S') of (brk, uu-) ⇒ ¬ brk› and
    twl-o: ‹cdcl-twl-o-prog-spec T (brk', U)› and
    ‹twl-struct-invs T› and
    cp: ‹cdcl-twl-cp** S' T› and
    ‹literals-to-update T = {#}› and
    ‹∀ S'. ¬ cdcl-twl-cp T S'› and
    ‹twl-stgy-invs T›
  shows ‹cdcl-twl-stgy** S U›
proof −
  have ‹cdcl-twl-stgy** S S'›
    using twl-stgy by fast
  moreover {
    have ‹cdcl-twl-o** T U›
      using twl-o by fast
    then have ‹cdcl-twl-stgy** S' U›
      using cp by (auto dest!: rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD)
  }
  ultimately show ?thesis by auto
qed

lemma cdcl-twl-stgy-prog-spec:
  assumes ‹twl-struct-invs S› and ‹twl-stgy-invs S› and ‹clauses-to-update S = {#}› and
    ‹get-conflict S = None› and
    ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S)›
  shows
    ‹cdcl-twl-stgy-prog S ≤ conclusive-TWL-norestart-run S›
  unfolding cdcl-twl-stgy-prog-def full-def conclusive-TWL-norestart-run-def
  apply (refine-vcg WHILEIT-rule[where
      R = ‹{((brkT, T), (brkS, S)). twl-struct-invs S ∧ cdcl-twl-stgy++ S T} ∪
          {((brkT, T), (brkS, S)). S = T ∧ brkT ∧ ¬brkS}›];
      remove-dummy-vars)
  — Well foundedness of the relation
  subgoal using wf-cdcl-twl-stgy-measure .

  — initial invariants:
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
```

subgoal using *assms* **by** *simp*

— loop invariants:
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl$_W$-cp*)
subgoal by *simp*
subgoal for *brk b x*
apply (*subgoal-tac* ‹*pcdcl*** (*pstate$_W$-of S*) (*pstate$_W$-of x*)›)
subgoal
using *assms*
rtranclp-pcdcl-entailed-by-init[*of* ‹*pstate$_W$-of S*› ‹*pstate$_W$-of x*›]
by (*auto simp*: *twl-struct-invs-def*)
subgoal
apply (*auto simp*: *twl-struct-invs-def*)
apply (*meson assms*(*1*) *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-stgy-cdcl$_W$-stgy rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-tcore-stgy-pcdcl-stgy′ rtranclp-trans*)+
done
done
subgoal by *simp*
subgoal by *simp*
subgoal by (*rule cdcl-twl-o-final-twl-state*)
subgoal by (*rule cdcl-twl-o-prog-cdcl-twl-stgy*)
subgoal by *simp*
subgoal for *brk0 T U brl V*
by *clarsimp*

— Final properties
subgoal for *brk0 T U V* — termination
by (*rule cdcl-twl-stgy-in-measure*)
subgoal by *simp*
subgoal by *fast*
done

definition *cdcl-twl-stgy-prog-break* :: ‹*′v twl-st* ⇒ *′v twl-st nres*› **where**
‹*cdcl-twl-stgy-prog-break* S_0 =
do {
$b \leftarrow$ *SPEC*(λ-. *True*);
(b, *brk*, T) ← *WHILE$_T$*$^{\lambda(b,\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}inv\ S_0\ S}$
(λ(b, *brk*, -). $b \wedge \neg brk$)
(λ(-, *brk*, S). *do* {
$T \leftarrow$ *unit-propagation-outer-loop* S;
$T \leftarrow$ *cdcl-twl-o-prog* T;
$b \leftarrow$ *SPEC*(λ-. *True*);
RETURN (b, T)
})
(b, *False*, S_0);
if brk then RETURN T
else — finish iteration is required only
cdcl-twl-stgy-prog T
}
›

lemma *wf-cdcl-twl-stgy-measure-break*:

```
  ‹wf ({((bT, brkT, T), (bS, brkS, S)). twl-struct-invs S ∧ cdcl-twl-stgy⁺⁺ S T} ∪
        {((bT, brkT, T), (bS, brkS, S)). S = T ∧ brkT ∧ ¬brkS}
        )›
    (is ‹?wf ?R›)
proof −
  have 1: ‹wf ({((brkT, T), brkS, S). twl-struct-invs S ∧ cdcl-twl-stgy⁺⁺ S T} ∪
    {((brkT, T), brkS, S). S = T ∧ brkT ∧ ¬ brkS})›
    (is ‹wf ?S›)
    by (rule wf-cdcl-twl-stgy-measure)
  have ‹wf {((bT, T), (bS, S)). (T, S) ∈ ?S}›
    apply (rule wf-snd-wf-pair)
    apply (rule wf-subset)
    apply (rule 1)
    apply auto
    done
  then show ?thesis
    apply (rule wf-subset)
    apply auto
    done
qed

lemma cdcl-twl-stgy-prog-break-spec:
  assumes ‹twl-struct-invs S› and ‹twl-stgy-invs S› and ‹clauses-to-update S = {#}› and
    ‹get-conflict S = None› and
    ent: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S)›
  shows
    ‹cdcl-twl-stgy-prog-break S ≤ conclusive-TWL-norestart-run S›
  unfolding cdcl-twl-stgy-prog-break-def full-def conclusive-TWL-norestart-run-def
  apply (refine-vcg cdcl-twl-stgy-prog-spec[unfolded conclusive-TWL-norestart-run-def]
      WHILEIT-rule[where
    R = ‹{((bT, brkT, T), (bS, brkS, S)). twl-struct-invs S ∧ cdcl-twl-stgy⁺⁺ S T} ∪
        {((bT, brkT, T), (bS, brkS, S)). S = T ∧ brkT ∧ ¬brkS}›];
     remove-dummy-vars)
  — Well foundedness of the relation
  subgoal using wf-cdcl-twl-stgy-measure-break .

  — initial invariants:
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp

  — loop invariants:
  subgoal by simp
  subgoal by simp
  subgoal by simp
  subgoal by (simp add: no-step-cdcl-twl-cp-no-step-cdcl_W-cp)
  subgoal by simp+
  subgoal for brk b x xa
    apply (subgoal-tac ‹pcdcl** (pstate_W-of S) (pstate_W-of xa)›)
    subgoal
      using assms
        rtranclp-pcdcl-entailed-by-init[of ‹pstate_W-of S› ‹pstate_W-of xa›]
      by (auto simp: twl-struct-invs-def)
```

subgoal
apply (*auto simp*: *twl-struct-invs-def*)
apply (*meson assms*(*1*) *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-tcore-stgy-pcdcl-stgy*$'$ *rtranclp-trans*)+
done
done

subgoal by *simp*
subgoal by *simp*
subgoal for *x a aa ba xa x1a*
by (*rule cdcl-twl-o-final-twl-state*[*of S a aa ba*]) *simp-all*
subgoal for *x a aa ba xa x1a*
by (*rule cdcl-twl-o-prog-cdcl-twl-stgy*[*of S a aa ba xa x1a*]) *fast*+
subgoal by *simp*
subgoal for *brk0 T U brl V*
by *clarsimp*

— Final properties
subgoal for *x a aa ba xa xb* — termination
using *cdcl-twl-stgy-in-measure*[*of S a aa ba xa*] **by** *fast*
subgoal by *simp*
subgoal by *fast*

— second loop
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal for *brk b x*
apply (*subgoal-tac* ‹*pcdcl*** (*pstate*$_W$*-of S*) (*pstate*$_W$*-of x*)›)
subgoal
using *assms*
rtranclp-pcdcl-entailed-by-init[*of* ‹*pstate*$_W$*-of S*› ‹*pstate*$_W$*-of x*›]
by (*auto simp*: *twl-struct-invs-def*)
subgoal
apply (*auto simp*: *twl-struct-invs-def*)
apply (*meson assms*(*1*) *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-tcore-stgy-pcdcl-stgy*$'$ *rtranclp-trans*)+
done
done
subgoal using *assms* **by** *auto*
done

definition *cdcl-twl-stgy-prog-early* :: ‹$'v$ *twl-st* ⇒ (*bool* × $'v$ *twl-st*) *nres*› **where**
‹*cdcl-twl-stgy-prog-early* S_0 =
do {
b ← *SPEC*(λ-. *True*);
(b, *brk*, T) ← *WHILE*$_T$λ(b, S). *cdcl-twl-stgy-prog-inv* S_0 S
(λ(b, *brk*, -). b ∧ ¬*brk*)
(λ(-, *brk*, S). *do* {
T ← *unit-propagation-outer-loop* S;
T ← *cdcl-twl-o-prog* T;
b ← *SPEC*(λ-. *True*);
RETURN (b, T)
})
(b, *False*, S_0);

```
    RETURN (brk , T)
  }
  ›

lemma cdcl-twl-stgy-prog-early-spec:
  assumes ‹twl-struct-invs S› and ‹twl-stgy-invs S› and ‹clauses-to-update S = {#}› and
    ‹get-conflict S = None› and
    ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S)›
  shows
    ‹cdcl-twl-stgy-prog-early S ≤ partial-conclusive-TWL-norestart-run S›
  unfolding cdcl-twl-stgy-prog-early-def full-def partial-conclusive-TWL-norestart-run-def
  apply (refine-vcg
      WHILEIT-rule[where
     R = ‹{((bT, brkT, T), (bS, brkS, S)). twl-struct-invs S ∧ cdcl-twl-stgy^++ S T} ∪
         {((bT, brkT, T), (bS, brkS, S)). S = T ∧ brkT ∧ ¬brkS}›];
     remove-dummy-vars)
  — Well foundedness of the relation
  subgoal using wf-cdcl-twl-stgy-measure-break .

  — initial invariants:
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp
  subgoal using assms by simp

  — loop invariants:
  subgoal by simp
  subgoal by simp
  subgoal by simp
  subgoal by (simp add: no-step-cdcl-twl-cp-no-step-cdcl_W-cp)
  subgoal by simp
  subgoal for brk b x xa
    apply (subgoal-tac ‹pcdcl** (pstate_W-of S) (pstate_W-of xa)›)
    subgoal
      using assms
        rtranclp-pcdcl-entailed-by-init[of ‹pstate_W-of S› ‹pstate_W-of xa›]
      by (auto simp: twl-struct-invs-def)
    subgoal
      apply (auto simp: twl-struct-invs-def)
    apply (meson assms(1) rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-stgy-cdcl_W-stgy rtranclp-pcdcl-stgy-pcdcl
rtranclp-pcdcl-tcore-stgy-pcdcl-stgy' rtranclp-trans)+
      done
    done
  subgoal by simp
  subgoal by simp
  subgoal for x a aa ba xa x1a
    by (rule cdcl-twl-o-final-twl-state[of S a aa ba]) simp-all
  subgoal for x a aa ba xa x1a
    by (rule cdcl-twl-o-prog-cdcl-twl-stgy[of S a aa ba xa x1a]) fast+
  subgoal by simp
  subgoal for brk0 T U brl V
    by clarsimp

  — Final properties
```

subgoal for *x a aa ba xa xb* — termination
using *cdcl-twl-stgy-in-measure*[*of S a aa ba xa*] **by** *fast*
subgoal by *simp*
subgoal by *fast*
done

end
theory *Watched-Literals-Transition-System-Inprocessing*
imports *Watched-Literals-Transition-System*
begin

The subsumption is very similar to the PCDCL case.

inductive *cdcl-twl-subsumed* :: ‹*'v twl-st* ⇒ *'v twl-st* ⇒ *bool*› **where**
subsumed-II:
‹*cdcl-twl-subsumed* (*M*, *N* + {#*C*, *C'*#}, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *add-mset* (*clause C'*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*clause C* ⊆# *clause C'*› |
subsumed-RR:
‹*cdcl-twl-subsumed* (*M*, *N*, *U* + {#*C*, *C'*#}, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N*, *add-mset C U*, *D*, *NE*, *UE*, *NS*, *add-mset* (*clause C'*) *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*clause C* ⊆# *clause C'*› |
subsumed-IR:
‹*cdcl-twl-subsumed* (*M*, *add-mset C N*, *add-mset C' U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset* (*clause C'*) *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*clause C* ⊆# *clause C'*› |
subsumed-RI:
‹*cdcl-twl-subsumed* (*M*, *add-mset C' N*, *add-mset C U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *add-mset* (*clause C'*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*clause C* ⊆# *clause C'*› ‹¬*tautology* (*clause C*)› ‹*distinct-mset* (*clause C*)›

lemma *cdcl-twl-subsumed-cdcl-subsumed*:
‹*cdcl-twl-subsumed S T* ⟹ *cdcl-subsumed* (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*) ∨ *cdcl-subsumed-RI* (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*)›
apply (*induction rule*: *cdcl-twl-subsumed.induct*)
subgoal by (*auto simp*: *cdcl-subsumed.simps pstate*$_W$ *-of.simps*)
subgoal by (*auto simp*: *cdcl-subsumed.simps*)
subgoal by (*auto simp*: *cdcl-subsumed.simps*)
subgoal by (*auto simp*: *cdcl-subsumed-RI.simps*)
done

lemma *cdcl-twl-subsumed-II-simp*:
‹*cdcl-twl-subsumed S S'*›
if ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*S'* = (*M*, *remove1-mset C' N*, *U*, *D*, *NE*, *UE*, *add-mset* (*clause C'*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›*e*
‹*clause C* ⊆# *clause C'*›
‹*C* ∈# *N*›
‹*C'* ∈# *remove1-mset C N*›
using *that subsumed-II*[*of C C'*] **by** (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-subsumed-RR-simp*:
‹*cdcl-twl-subsumed S S'*›
if ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*S'* = (*M*, *N*, *remove1-mset C' U*, *D*, *NE*, *UE*, *NS*, *add-mset* (*clause C'*) *US*, *N0*, *U0*, {#}, *Q*)›
‹*clause C* ⊆# *clause C'*›
‹*C* ∈# *U*›
‹*C'* ∈# *remove1-mset C U*›

using *that subsumed-RR*[*of C C′ M N ‹U −{#C,C′#}› D NE UE NS US N0 U0 Q*]
by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-subsumed-IR-simp*:
‹*cdcl-twl-subsumed S S′*›
if ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*S′* = (*M*, *N*, *remove1-mset C′ U*, *D*, *NE*, *UE*, *NS*, *add-mset* (*clause C′*) *US*, *N0*, *U0*, {#}, *Q*)›
‹*clause C* ⊆# *clause C′*›
‹*C* ∈# *N*›
‹*C′* ∈# *U*›
using *that subsumed-IR*[*of C C′ M ‹N − {#C#}› ‹U −{#C′#}› D NE UE NS US N0 U0 Q*]
by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-subsumed-RI-simp*:
‹*cdcl-twl-subsumed S T*›
if ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)› ‹*clause C* ⊆# *clause C′*›
‹*T* = (*M*, *add-mset C* (*remove1-mset C′ N*), *remove1-mset C U*, *D*, *NE*, *UE*, *add-mset* (*clause C′*) *NS*,
US, *N0*, *U0*, {#}, *Q*)›
‹¬*tautology* (*clause C*)› ‹*distinct-mset* (*clause C*)›
‹*C′* ∈# *N*› ‹*C* ∈# *U*›
using *that subsumed-RI*[*of C C′ M ‹N − {#C′#}› ‹U − {#C#}› D NE UE NS US N0 U0 Q*]
by (*auto dest*!: *multi-member-split*)

The lifting from *cdcl-subresolution* is a lot more complicated due to the handling of unit and nonunit clauses. Basically, we have to split every rule in two. Hence we don't have a one-to-one mapping anymore, but need to use *cdcl-flush-unit* or rule of that kind.

We don't support (yet) generation of the empty clause. This is very tricky because we entirely leave the CDCL calculus.

The condition ∀ *L*∈#*clause E*. *undefined-lit M L* is not necessary from the point of view of CDCL, but it makes it much easier to fulfill the conditions of the watched literals. It should be possible to do so, but we would need to add conditions on it. However, this makes the inprocessing harder to do.

inductive *cdcl-twl-subresolution* :: ‹*'v twl-st* ⇒ *'v twl-st* ⇒ *bool*› **where**
twl-subresolution-II-nonunit:
‹*cdcl-twl-subresolution* (*M*, *N* + {#*C*, *C′*#}, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N* + {#*C*, *E*#}, *U*, *None*, *NE*, *UE*, *add-mset* (*clause C′*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if
‹*clause C* = *add-mset L D*›
‹*clause C′* = *add-mset* (−*L*) *D′*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D′*› ‹¬*tautology* (*D* + *D′*)›
‹*clause E* = *remdups-mset D′*› ‹*size* (*watched E*) = *2*› ‹*struct-wf-twl-cls E*› ‹∀ *L* ∈# *clause E*. *undefined-lit M L*› |
twl-subresolution-II-unit:
‹*cdcl-twl-subresolution* (*M*, *N* + {#*C*, *C′*#}, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K* {#*K*#} # *M*, *N* + {#*C*#}, *U*, *None*, *add-mset* {#*K*#} *NE*, *UE*,
add-mset (*clause C′*) *NS*, *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
if
‹*clause C* = *add-mset L D*›
‹*clause C′* = *add-mset* (−*L*) *D′*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D′*› ‹¬*tautology* (*D* + *D′*)›
‹*remdups-mset D′* = {#*K*#}› ‹*undefined-lit M K*›|

twl-subresolution-LL-nonunit:

$\langle$*cdcl-twl-subresolution* $(M,\ N,\ U + \{\#C,\ C'\#\},$ *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)$
$(M,\ N,\ U + \{\#C,\ E\#\},$ *None*, *NE*, *UE*, *NS*, *add-mset* (*clause* C') *US*, *N0*, *U0*, $\{\#\},\ Q)\rangle$
if
$\langle$*clause* $C =$ *add-mset* $L\ D\rangle$
$\langle$*clause* $C' =$ *add-mset* $(-L)\ D'\rangle$
$\langle$*count-decided* $M = 0\rangle$ $\langle D \subseteq\# D'\rangle$
$\langle$*clause* $E =$ *remdups-mset* $D'\rangle$ $\langle\neg$*tautology* $(D + D')\rangle$ $\langle$*size* (*watched* E) $= 2\rangle$ $\langle\forall L \in\#$ *clause* E. *undefined-lit* $M\ L\rangle$ |
twl-subresolution-LL-unit:
$\langle$*cdcl-twl-subresolution* $(M,\ N,\ U + \{\#C,\ C'\#\},$ *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)$
(*Propagated* $K\ \{\#K\#\}\ \#\ M$, $N,\ U + \{\#C\#\},$ *None*, *NE*, *add-mset* $\{\#K\#\}$ *UE*, *NS*,
add-mset (*clause* C') *US*, *N0*, *U0*, $\{\#\},$ *add-mset* $(-K)\ Q)\rangle$
if
$\langle$*clause* $C =$ *add-mset* $L\ D\rangle$
$\langle$*clause* $C' =$ *add-mset* $(-L)\ D'\rangle$
$\langle$*count-decided* $M = 0\rangle$ $\langle D \subseteq\# D'\rangle$
$\langle$*remdups-mset* $D' = \{\#K\#\}\rangle$ $\langle\neg$*tautology* $(D + D')\rangle$ $\langle$*undefined-lit* $M\ K\rangle$|

twl-subresolution-LI-nonunit:
$\langle$*cdcl-twl-subresolution* $(M,\ N + \{\#C\#\},\ U + \{\#C'\#\},$ *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)$
$(M,\ N + \{\#C\#\},\ U + \{\#E\#\},$ *None*, *NE*, *UE*, *NS*, *add-mset* (*clause* C') *US*, *N0*, *U0*, $\{\#\},\ Q)\rangle$
if
$\langle$*clause* $C =$ *add-mset* $L\ D\rangle$
$\langle$*clause* $C' =$ *add-mset* $(-L)\ D'\rangle$
$\langle$*count-decided* $M = 0\rangle$ $\langle D \subseteq\# D'\rangle$
$\langle$*clause* $E =$ *remdups-mset* $D'\rangle$ $\langle\neg$*tautology* $(D + D')\rangle$ $\langle$*size* (*watched* E) $= 2\rangle$ $\langle\forall L \in\#$ *clause* E. *undefined-lit* $M\ L\rangle$|
twl-subresolution-LI-unit:
$\langle$*cdcl-twl-subresolution* $(M,\ N + \{\#C\#\},\ U + \{\#C'\#\},$ *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)$
(*Propagated* $K\ \{\#K\#\}\ \#\ M$, $N + \{\#C\#\},\ U,$ *None*, *NE*, *add-mset* $\{\#K\#\}$ *UE*, *NS*,
add-mset (*clause* C') *US*, *N0*, *U0*, $\{\#\},$ *add-mset* $(-K)\ Q)\rangle$
if
$\langle$*clause* $C =$ *add-mset* $L\ D\rangle$
$\langle$*clause* $C' =$ *add-mset* $(-L)\ D'\rangle$
$\langle$*count-decided* $M = 0\rangle$ $\langle D \subseteq\# D'\rangle$
$\langle$*remdups-mset* $D' = \{\#K\#\}\rangle$ $\langle\neg$*tautology* $(D + D')\rangle$ $\langle$*undefined-lit* $M\ K\rangle$|

twl-subresolution-IL-nonunit:
$\langle$*cdcl-twl-subresolution* $(M,\ N + \{\#C'\#\},\ U + \{\#C\#\},$ *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)$
$(M,\ N + \{\#E\#\},\ U + \{\#C\#\},$ *None*, *NE*, *UE*, *add-mset* (*clause* C') *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)\rangle$
if
$\langle$*clause* $C =$ *add-mset* $L\ D\rangle$
$\langle$*clause* $C' =$ *add-mset* $(-L)\ D'\rangle$
$\langle$*count-decided* $M = 0\rangle$ $\langle D \subseteq\# D'\rangle$
$\langle$*clause* $E =$ *remdups-mset* $D'\rangle$ $\langle\neg$*tautology* $(D + D')\rangle$ $\langle$*size* (*watched* E) $= 2\rangle$ $\langle\forall L \in\#$ *clause* E. *undefined-lit* $M\ L\rangle$ |
twl-subresolution-IL-unit:
$\langle$*cdcl-twl-subresolution* $(M,\ N + \{\#C'\#\},\ U + \{\#C\#\},$ *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, $\{\#\},\ Q)$
(*Propagated* $K\ \{\#K\#\}\ \#\ M$, $N,\ U + \{\#C\#\},$ *None*, *add-mset* $\{\#K\#\}$ *NE*, *UE*,
add-mset (*clause* C') *NS*, *US*, *N0*, *U0*, $\{\#\},$ *add-mset* $(-K)\ Q)\rangle$
if
$\langle$*clause* $C =$ *add-mset* $L\ D\rangle$
$\langle$*clause* $C' =$ *add-mset* $(-L)\ D'\rangle$
$\langle$*count-decided* $M = 0\rangle$ $\langle D \subseteq\# D'\rangle$
$\langle$*remdups-mset* $D' = \{\#K\#\}\rangle$ $\langle\neg$*tautology* $(D + D')\rangle$ $\langle$*undefined-lit* $M\ K\rangle$

lemma *past-invs-count-decided0*: ‹*count-decided* (*get-trail S*) = *0* ⟹ *past-invs S*›
by (*cases S*) (*auto simp*: *past-invs.simps*)

lemma *cdcl-twl-subresolution-past-invs*: ‹*cdcl-twl-subresolution S T* ⟹ *past-invs T*›
by (*induction rule*: *cdcl-twl-subresolution.induct*)
(*auto simp*: *past-invs-count-decided0*)

lemma *twl-lazy-update-subresII*: ‹*count-decided M* = *0* ⟹ *struct-wf-twl-cls C* ⟹
¬ *twl-is-an-exception* (*C*) (*Q*) {#} ⟹ − *K* ∉# *watched C* ⟹
twl-lazy-update (*M*) *C* ⟹
twl-lazy-update (*Propagated K* {#*K*#} # *M*) *C*›
using *count-decided-ge-get-level*[*of M*]
by (*cases C*)
(*fastforce simp*: *get-level-cons-if has-blit-def uminus-lit-swap twl-is-an-exception-def*
dest!: *multi-member-split*
elim!: *size-mset-SucE*)

lemma *watched-literals-false-of-max-level-prop-lvl0*: ‹*count-decided M* = *0* ⟹ *watched-literals-false-of-max-level*
(*Propagated K* {#*K*#} # *M*) *C*›
using *count-decided-ge-get-level*[*of M*]
by (*cases C*)
(*fastforce simp*: *get-level-cons-if has-blit-def uminus-lit-swap twl-is-an-exception-def*
dest!: *multi-member-split*
elim!: *size-mset-SucE*)

lemma *watched-literals-false-of-max-level-lvl0*: ‹*count-decided M* = *0* ⟹ *watched-literals-false-of-max-level*
(*M*) *C*›
using *count-decided-ge-get-level*[*of M*]
by (*cases C*)
(*fastforce simp*: *get-level-cons-if has-blit-def uminus-lit-swap twl-is-an-exception-def*
dest!: *multi-member-split*
elim!: *size-mset-SucE*)

lemma *twl-lazy-update-undefined*: ‹∀ *L* ∈# *clause E*. *undefined-lit M L* ⟹ *twl-lazy-update M E*›
by (*cases E*)
(*auto simp*: *has-blit-def Decided-Propagated-in-iff-in-lits-of-l*
dest!: *multi-member-split*)

lemma *struct-wf-twl-cls-remdupsI*:
‹*clause E* = *remdups-mset D'* ⟹ *size* (*watched E*) = *2* ⟹ *struct-wf-twl-cls E*›
by (*cases E*) *auto*

lemma *cdcl-twl-subresolution-twl-st-inv*:
‹*cdcl-twl-subresolution S T* ⟹ *twl-st-inv S* ⟹ *twl-st-inv T*›
by (*induction rule*: *cdcl-twl-subresolution.induct*)
(*auto simp*: *twl-st-inv.simps twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0*
twl-lazy-update-subresII twl-is-an-exception-add-mset-to-queue
watched-literals-false-of-max-level-prop-lvl0
intro: *struct-wf-twl-cls-remdupsI*)

lemma *cdcl-twl-subresolution-valid-enqueued*:
‹*cdcl-twl-subresolution S T* ⟹ *valid-enqueued S* ⟹ *valid-enqueued T*›
by (*induction rule*: *cdcl-twl-subresolution.induct*)
(*auto simp*: *get-level-cons-if*)

lemma *cdcl-twl-subresolution-decompE*:
assumes
‹*cdcl-twl-subresolution S T*› **and** ‹*Multiset.Ball* (*get-clauses S*) (*distinct-mset o clause*)› **and**
subres: ‹*cdcl-subresolution* (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*) ⟹ *thesis*› **and**
unit: ‹⋀*S' T'*. *cdcl-subresolution* (*pstate*$_W$ *-of S*) *S'* ⟹
cdcl-propagate S' (*T'*) ⟹ *cdcl-flush-unit* (*T'*) (*pstate*$_W$ *-of T*) ⟹ *thesis*›
shows *thesis*
using *assms*(*1*,*2*,*3*)
apply (*cases rule*: *cdcl-twl-subresolution.cases*)
subgoal for *C L D C' D' M E N U NE UE NS US Q*
apply (*rule subres*)
using *cdcl-subresolution.intros*(*1*)[*of M D D'* ‹*clauses N*› *L* ‹*clauses U*› *None NE UE NS US*]
by *auto*
subgoal for *C L D C' D' M E N U NE UE NS US N0 U0 Q*
apply (*rule unit*[*of* ‹(*M*, *clauses N* + {#*clause C*, *D'*#}, *clauses U*, *None*, *NE*, *UE*,
add-mset (*clause C'*) *NS*, *US*, *N0*, *U0*)›
‹(*Propagated E D'* # *M*, *clauses N* + {#*clause C*, *D'*#}, *clauses U*, *None*, *NE*, *UE*,
add-mset (*clause C'*) *NS*, *US*, *N0*, *U0*)›])
subgoal
by (*auto simp*: *cdcl-subresolution.simps distinct-mset-remdups-mset-id*)
subgoal
by (*auto simp*: *cdcl-propagate.simps distinct-mset-remdups-mset-id*)
subgoal
by (*auto simp*: *cdcl-flush-unit.simps distinct-mset-remdups-mset-id*)
done
supply[[*goals-limit=1*]]
subgoal for *C L D C' D' M E N U NE UE NS US Q*
apply (*rule subres*)
using *cdcl-subresolution.intros*(*2*)[*of M D D'* ‹*clauses N*› ‹*clauses U*› *L None NE UE NS US*]
by *auto*
subgoal for *C L D C' D' M E N U NE UE NS US N0 U0 Q*
apply (*rule unit*[*of* ‹(*M*, *clauses N*, *clauses U* + {#*clause C*, *D'*#}, *None*, *NE*, *UE*,
NS, *add-mset* (*clause C'*) *US*, *N0*, *U0*)›
‹(*Propagated E D'* # *M*, *clauses N*, *clauses U* + {#*clause C*, *D'*#}, *None*, *NE*, *UE*, *NS*,
add-mset (*clause C'*) *US*, *N0*, *U0*)›])
subgoal
apply (*auto simp*: *cdcl-subresolution.simps distinct-mset-remdups-mset-id*)
apply (*rule-tac x*=*D* **in** *exI*)
apply (*rule-tac x*=*D'* **in** *exI*)
apply *auto*
done
subgoal
by (*auto simp*: *cdcl-propagate.simps distinct-mset-remdups-mset-id*)
subgoal
by (*auto simp*: *cdcl-flush-unit.simps distinct-mset-remdups-mset-id*)
done
subgoal for *C L D C' D' M E N U NE UE NS US Q*
apply (*rule subres*)
using *cdcl-subresolution.intros*(*3*)[*of M D D'* ‹*clauses N*› *L* ‹*clauses U*› *None NE UE NS US*]
by *auto*
subgoal for *C L D C' D' M E N U NE UE NS US N0 U0 Q*
apply (*rule unit*[*of* ‹(*M*, *clauses N* + {#*clause C*#}, *clauses U* + {#*D'*#}, *None*, *NE*, *UE*,
NS, *add-mset* (*clause C'*) *US*, *N0*, *U0*)›
‹(*Propagated E D'* # *M*, *clauses N* + {#*clause C*#}, *clauses U* + {#*D'*#}, *None*, *NE*, *UE*, *NS*,
add-mset (*clause C'*) *US*, *N0*, *U0*)›])
subgoal

apply (*auto simp*: *cdcl-subresolution.simps distinct-mset-remdups-mset-id*)
apply (*drule-tac x*=*D* **in** *spec*)
apply (*drule-tac x*=*D′* **in** *spec*)
apply *auto*
done
subgoal
by (*auto simp*: *cdcl-propagate.simps distinct-mset-remdups-mset-id*)
subgoal
by (*auto simp*: *cdcl-flush-unit.simps distinct-mset-remdups-mset-id*)
done
subgoal for *C L D C′ D′ M E N U NE UE NS US N0 U0 Q*
apply (*rule subres*)
using *cdcl-subresolution.intros*(*4*)[*of M D′ D* ‹*clauses N*› ‹−*L*› ‹*clauses U*› *None NE UE NS US*]
by (*auto simp*: *ac-simps*)
subgoal for *C L D C′ D′ M K N U NE UE NS US N0 U0 Q*
apply (*rule unit*[*of* ‹(*M*, *clauses N* + {#*D′*#}, *clauses U* + {#*clause C*#}, *None*, *NE*, *UE*,
add-mset (*clause C′*) *NS*, *US*, *N0*, *U0*)›
‹(*Propagated K D′* # *M*, *clauses N* + {#*D′*#}, *clauses U* + {#*clause C*#}, *None*, *NE*, *UE*,
add-mset (*clause C′*) *NS*, *US*, *N0*, *U0*)›])
subgoal
apply (*auto simp*: *cdcl-subresolution.simps distinct-mset-remdups-mset-id*)
apply (*auto 5 5 intro*: *dest*: *spec*[*of* - ‹−*L*›] *dest*!: *spec*[*of* - ‹*clauses N*›])
done
subgoal
by (*auto simp*: *cdcl-propagate.simps distinct-mset-remdups-mset-id*)
subgoal
by (*auto simp*: *cdcl-flush-unit.simps distinct-mset-remdups-mset-id*)
done
done

lemma *cdcl-twl-subresolution-pcdcl-all-struct-invs*:
‹*cdcl-twl-subresolution S T* ⟹
pcdcl-all-struct-invs (*pstate*$_W$ *-of S*) ⟹ *pcdcl-all-struct-invs* (*pstate*$_W$ *-of T*)›
apply (*elim cdcl-twl-subresolution-decompE*)
subgoal
by (*cases S*)
(*auto simp*: *pcdcl-all-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def*
dest!: *multi-member-split*)
subgoal
by (*drule cdcl-subresolution*)
(*simp-all add*: *rtranclp-pcdcl-all-struct-invs*)
subgoal
apply (*drule pcdcl.intros pcdcl-core.intros*)+
apply (*drule cdcl-subresolution*)
using *rtranclp-pcdcl-all-struct-invs* **by** *blast*
done

lemma *cdcl-twl-subresolution-smaller-propa*:
‹*cdcl-twl-subresolution S T* ⟹ *cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of T*)›
apply (*induction rule*: *cdcl-twl-subresolution.induct*)
apply (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def*)
apply (*case-tac M′*; *auto*; *fail*)+
done
lemma *cdcl-twl-subresolution-twl-st-exception-inv*:
‹*cdcl-twl-subresolution S T* ⟹ *no-dup* (*get-trail S*) ⟹

twl-st-exception-inv S ⟹ twl-st-exception-inv T›
apply (*induction rule*: *cdcl-twl-subresolution.induct*)
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis mset-subset-eqD mset-subset-eq-add-left set-mset-remdups-mset uminus-lits-of-l-definedD*)
apply (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
by (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis Un-iff clause.simps no-has-blit-propagate twl-clause.sel*(*1*) *twl-clause.sel*(*2*))+
done
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis mset-subset-eqD mset-subset-eq-add-left set-mset-remdups-mset uminus-lits-of-l-definedD*)
apply (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
by (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis Un-iff clause.simps no-has-blit-propagate twl-clause.sel*(*1*) *twl-clause.sel*(*2*))+
done
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis mset-subset-eqD mset-subset-eq-add-left set-mset-remdups-mset uminus-lits-of-l-definedD*)
apply (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
by (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis Un-iff clause.simps no-has-blit-propagate twl-clause.sel*(*1*) *twl-clause.sel*(*2*))+
done
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis mset-subset-eqD mset-subset-eq-add-left set-mset-remdups-mset*

uminus-lits-of-l-definedD)
apply (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
by (*metis Un-iff clause.simps twl-clause.sel*(*1*) *twl-clause.sel*(*2*))
subgoal
unfolding *twl-st-exception-inv.simps*
apply (*intro ballI*)
apply (*rename-tac x*; *case-tac x*)
apply (*auto simp*: *twl-exception-inv.simps*)
apply (*metis Un-iff clause.simps no-has-blit-propagate twl-clause.sel*(*1*) *twl-clause.sel*(*2*))+
done
done

lemma *cdcl-twl-subresolution-dup-enqueued*:
‹*cdcl-twl-subresolution S T* ⟹ *no-duplicate-queued S* ⟹ *no-duplicate-queued T*›
by (*induction rule*: *cdcl-twl-subresolution.induct*) *auto*

lemma *cdcl-twl-subresolution-distinct-enqueued*:
‹*cdcl-twl-subresolution S T* ⟹ *distinct-queued S* ⟹ *no-duplicate-queued S* ⟹ *distinct-queued T*›
apply (*induction rule*: *cdcl-twl-subresolution.induct*)
unfolding *distinct-queued.simps*
by (*auto dest*: *mset-le-add-mset simp*: *undefined-notin*)

lemma *Cons-entails-CNotE*:
assumes ‹*K* # *M* ⊨*as CNot Ca*› ‹*distinct-mset Ca*› **and**
1: ‹*M* ⊨*as CNot Ca* ⟹ −*lit-of K* ∉# *Ca* ⟹ *thesis*› **and**
2: ‹*M* ⊨*as CNot* (*remove1-mset* (−*lit-of K*) *Ca*) ⟹ −*lit-of K* ∈# *Ca* ⟹ *thesis*›
shows *thesis*
using *assms*(*1*,*2*)
apply (*cases* ‹−*lit-of K* ∈# *Ca*›)
subgoal
by (*rule 2*)
(*auto dest*!: *multi-member-split*
simp: *true-annots-true-cls-def-iff-negation-in-model uminus-lit-swap*
add-mset-eq-add-mset)
subgoal
by (*rule 1*)
(*auto dest*!: *multi-member-split*
simp: *true-annots-true-cls-def-iff-negation-in-model uminus-lit-swap*
add-mset-eq-add-mset)
done

lemma *propa-confl-cands-enqueued-simps*[*simp*]:
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *add-mset E NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟷
propa-confl-cands-enqueued (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#},*Q*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *add-mset E UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟷
propa-confl-cands-enqueued (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#},*Q*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*C′*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)
⟷
propa-confl-cands-enqueued (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *add-mset* (*C′*) *US*, *N0*, *U0*, {#}, *Q*)
⟷
propa-confl-cands-enqueued (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
apply (*auto*)
done

lemma *propa-confl-cands-enqueuedD*:
  ‹*propa-confl-cands-enqueued* (*M*, *add-mset C N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟹
    propa-confl-cands-enqueued (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
  ‹*propa-confl-cands-enqueued* (*M*, *N*, *add-mset C U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟹
    propa-confl-cands-enqueued (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
  by *auto*

lemma *add-mset-diff-add-mset-If*:
  (*add-mset L' C*) − *add-mset L C'*= (*if L = L' then C − C' else remove1-mset L C* + {#*L'*#} − *C'*)
  by (*auto simp*: *multiset-eq-iff*)

lemma *propa-confl-cands-enqueued-propagate*:
  assumes
    ‹*Multiset.Ball* (*N*+*U*) (*struct-wf-twl-cls*)› **and**
    prev: ‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)› **and**
    excep: ‹*twl-st-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)› **and**
    ‹*count-decided M* = *0*› **and**
    nd: ‹*no-dup* (*Propagated L C* # *M*)›
  shows ‹*propa-confl-cands-enqueued* (*Propagated L C* # *M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*,
    {#}, *add-mset* (−*L*) *Q*)›
  unfolding *propa-confl-cands-enqueued.simps*
proof (*intro ballI impI*)
  fix *Ca La*
  assume *NU*: ‹*Ca*∈#*N* + *U*› **and** *La-Ca*: ‹*La* ∈# *clause Ca*› **and**
    ent: ‹*Propagated L C* # *M* ⊨*as CNot* (*remove1-mset La* (*clause Ca*))› **and**
    not-true: ‹*La* ∉ *lits-of-l* (*Propagated L C* # *M*)›
  have [*simp*]: ‹*get-level M L* = *0*› **for** *L*
    using *count-decided-ge-get-level*[*of M*] *assms*
      by *auto*
  have *dist2*: ‹*distinct-mset* (*clause Ca*)› **and** *watched*: ‹*size* (*watched Ca*) = *2*›
    using *assms*(*1*) *NU* **by** (*cases Ca*; *auto dest*!: *multi-member-split*)+
  then have *dist*: ‹*distinct-mset* (*remove1-mset La* (*clause Ca*))›
    by *auto*
  show ‹(∃ *L'*. *L'* ∈# *watched Ca* ∧ *L'* ∈# *add-mset* (− *L*) *Q*) ∨ (∃ *L*. (*L*, *Ca*) ∈# {#})›
  proof (*rule Cons-entails-CNotE*[*OF ent dist*])
    assume ‹*M* ⊨*as CNot* (*remove1-mset La* (*clause Ca*))› **and**
      ‹− *lit-of* (*Propagated L C*) ∉# *remove1-mset La* (*clause Ca*)›
    then have ‹∃ *L'*. *L'* ∈# *watched Ca* ∧ *L'* ∈# *Q*›
      using *prev NU not-true La-Ca*
      by (*auto simp*: *dest*!: *multi-member-split*)
    then show *?thesis*
      by *auto*
  next
    assume *neg*: ‹*M* ⊨*as CNot* (*remove1-mset* (− *lit-of* (*Propagated L C*)) (*remove1-mset La* (*clause*
Ca)))› **and**
      inL: ‹− *lit-of* (*Propagated L C*) ∈# *remove1-mset La* (*clause Ca*)›
    with *in-diffD*[*OF this*(*2*)] **have** [*simp*]: ‹*L* ≠ −*La*› ‹*L* ≠ *La*›
      using *dist2 not-true* **by** (*auto dest*!: *multi-member-split*)

    have ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) *Ca*›
      using *excep NU* **by** *auto*
    then show ‹(∃ *L'*. *L'* ∈# *watched Ca* ∧ *L'* ∈# *add-mset* (− *L*) *Q*) ∨ (∃ *L*. (*L*, *Ca*) ∈# {#})›
      apply −
      apply (*rule ccontr*)
      using *neg watched not-true nd inL*
      apply (*cases Ca*)

apply (*auto elim*!: *Cons-entails-CNotE dest*!: *multi-member-split*[*of* - *N*] *multi-member-split*[*of* ‹−*L*›]
simp: *distinct-mset-remove1-All uminus-lit-swap removeAll-notin has-blit-def add-mset-diff-add-mset-If twl-exception-inv.simps size-2-iff all-conj-distrib remove1-mset-add-mset-If*)
apply (*auto split*: *if-splits simp*: *remove1-mset-add-mset-If Decided-Propagated-in-iff-in-lits-of-l dest*: *no-dup-consistentD uminus-lits-of-l-definedD dest*!: *multi-member-split*; *fail*)+
done
qed
qed

lemma *propa-confl-cands-enqueued-learn*:
assumes
prev: ‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)› **and**
‹∀ *L* ∈# *clause C*. *undefined-lit M L*› ‹*struct-wf-twl-cls C*› ‹*no-dup M*›
shows ‹*propa-confl-cands-enqueued* (*M*, *add-mset C N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *add-mset C U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
using *assms*
by (*cases C*; *force simp*: *size-2-iff Decided-Propagated-in-iff-in-lits-of-l*)+

lemma *propa-confl-cands-enqueued-learn-empty*:
assumes
prev: ‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
shows ‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *Some C*, *NE*, *UE*, *NS*, *US*, *add-mset* {#} *N0*, *U0*, {#}, *Q′*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *Some C*, *NE*, *UE*, *NS*, *US*, *N0*, *add-mset* {#} *U0*, {#}, *Q′*)›
using *assms*
by (*cases C*; *force simp*: *Decided-Propagated-in-iff-in-lits-of-l*)+

lemma *twl-exception-inv-skip-clause*:
‹*twl-exception-inv* (*M*, *add-mset C′* (*N*), *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) *C* ⟹
twl-exception-inv (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) *C*›
‹*twl-exception-inv* (*M*, *N*, *add-mset C′ U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) *C* ⟹
twl-exception-inv (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) *C*›
by (*cases C*) (*auto simp*: *twl-exception-inv.simps*)

lemma *cdcl-twl-subresolution-propa-confl-cands-enqueued*:
assumes
‹*cdcl-twl-subresolution S T*›
‹*Multiset.Ball* (*get-clauses S*) (*struct-wf-twl-cls*)› **and**
prev: ‹*propa-confl-cands-enqueued S*› **and**
excep: ‹*twl-st-exception-inv S*› **and**
nd: ‹*no-dup* (*get-trail S*)›
shows ‹*propa-confl-cands-enqueued T*›
using *assms*
apply (*induction rule*: *cdcl-twl-subresolution.induct*)
subgoal for *C L D C′ D′ M E N U NE UE NS US N0 U0 Q*
by (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -]
intro!: *propa-confl-cands-enqueued-learn*(*1*)[**where** *C*=*E*]
dest: *propa-confl-cands-enqueuedD*)
subgoal for *C L D C′ D′ M K N U NE UE NS US Q*
apply (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -] *twl-exception-inv-skip-clause*[**where** *C′*=*C′* **and** *N*=‹*add-mset C*

N›]
intro: *propa-confl-cands-enqueued-learn*(*1*)[**where** *C*=*C'* **and** *N*=‹*add-mset C N*›]
intro!: *propa-confl-cands-enqueued-propagate*
dest: *propa-confl-cands-enqueuedD*
dest!: *multi-member-split*[*of* - *N*] *multi-member-split*[*of* - *U*])
done
subgoal for *C L D C' D' M E N U NE UE NS US Q*
by (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -]
intro!: *propa-confl-cands-enqueued-learn*(*2*)[**where** *C*=*E*] *struct-wf-twl-cls-remdupsI*
dest: *propa-confl-cands-enqueuedD*)
subgoal for *C L D C' D' M K N U NE UE NS US Q*
apply (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -] *twl-exception-inv-skip-clause*[**where** *C'*=*C'* **and** *N*=‹*add-mset C*
N›]
intro: *propa-confl-cands-enqueued-learn*(*1*)[**where** *C*=*C'* **and** *N*=‹*add-mset C N*›]
intro!: *propa-confl-cands-enqueued-propagate*
dest: *propa-confl-cands-enqueuedD*
dest!: *multi-member-split*[*of* - *N*] *multi-member-split*[*of* - *U*])
apply (*simp-all add*: *twl-exception-inv.simps*(*1*))
done
subgoal for *C L D C' D' M E N U NE UE NS US Q*
by (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -]
intro!: *propa-confl-cands-enqueued-learn*(*2*)[**where** *C*=*E*] *struct-wf-twl-cls-remdupsI*
dest: *propa-confl-cands-enqueuedD*)
subgoal for *C L D C' D' M K N U NE UE NS US Q*
apply (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -] *twl-exception-inv-skip-clause*[**where** *C'*=*C'* **and** *N*=‹*add-mset C*
N›]
intro: *propa-confl-cands-enqueued-learn*(*1*)[**where** *C*=*C'* **and** *N*=‹*add-mset C N*›]
intro!: *propa-confl-cands-enqueued-propagate*
dest: *propa-confl-cands-enqueuedD*
dest!: *multi-member-split*[*of* - *N*] *multi-member-split*[*of* - *U*])
done
subgoal for *C L D C' D' M E N U NE UE NS US Q*
by (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -]
intro!: *propa-confl-cands-enqueued-learn*(*1*)[**where** *C*=*E*] *struct-wf-twl-cls-remdupsI*
dest: *propa-confl-cands-enqueuedD*)
subgoal for *C L D C' D' M K N U NE UE NS US Q*
apply (*auto simp del*: *propa-confl-cands-enqueued.simps*
simp: *add-mset-commute*[*of C* -] *twl-exception-inv-skip-clause*[**where** *C'*=*C'* **and** *N*=‹*add-mset C*
N›]
intro: *propa-confl-cands-enqueued-learn*(*1*)[**where** *C*=*C'* **and** *N*=‹*add-mset C N*›]
intro!: *propa-confl-cands-enqueued-propagate*
dest: *propa-confl-cands-enqueuedD*
dest!: *multi-member-split*[*of* - *N*] *multi-member-split*[*of* - *U*])
apply (*simp-all add*: *twl-exception-inv.simps*(*1*))
done
done

lemma *cdcl-twl-subresolution-conflict*:
‹*cdcl-twl-subresolution S T* ⟹ *get-conflict T* = *None*›
by (*induction rule*: *cdcl-twl-subresolution.induct*) *auto*

lemma *clause-alt-def*:
  ‹*clause C* = *watched C* + *unwatched C*›
  by (*cases C*) *auto*

lemma *cdcl-twl-subresolution-clauses-to-update-inv*:
  ‹*cdcl-twl-subresolution S T* ⟹ *no-dup* (*get-trail S*) ⟹
  clauses-to-update-inv S ⟹ *clauses-to-update-inv T*›
  apply (*induction rule*: *cdcl-twl-subresolution.induct*)
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      add-mset-eq-add-mset dest: *no-has-blit-propagate*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      add-mset-eq-add-mset dest: *no-has-blit-propagate*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      add-mset-eq-add-mset dest: *no-has-blit-propagate*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  subgoal
    by (*auto simp*: *all-conj-distrib clauses-to-update-prop.simps filter-mset-empty-conv*
      eq-commute[*of* - ‹*remdups-mset* -›] *clause-alt-def Decided-Propagated-in-iff-in-lits-of-l*
      add-mset-eq-add-mset dest: *no-has-blit-propagate*
      dest!: *multi-member-split*[*of* ‹- :: - *literal*›])
  done

lemma *cdcl-twl-subresolution-twl-struct-invs*:
  assumes ‹*cdcl-twl-subresolution S T*›
    ‹*twl-struct-invs S*›
  shows ‹*twl-struct-invs T*›
proof −
  have ‹*Multiset.Ball* (*get-clauses S*) *struct-wf-twl-cls*› ‹*no-dup* (*get-trail S*)›
    using *assms*(*2*) **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def*
      $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
    by (*cases S*; *auto simp*: *twl-st-simps*)+
  moreover have ‹*pcdcl-all-struct-invs* (*pstate*$_W$*-of T*)› ‹*twl-st-inv T*›

using *assms cdcl-twl-subresolution-pcdcl-all-struct-invs*[*of S T*]
cdcl-twl-subresolution-twl-st-inv[*of S T*]
unfolding *twl-struct-invs-def*
by *auto*
then have ‹*Multiset.Ball* (*get-clauses T*) *struct-wf-twl-cls*› ‹*no-dup* (*get-trail T*)›
unfolding *twl-struct-invs-def pcdcl-all-struct-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by (*cases T*; *auto simp*: *twl-st-simps*; *fail*)+
ultimately show *?thesis*
using *cdcl-twl-subresolution-twl-st-inv*[*of S T*]
cdcl-twl-subresolution-valid-enqueued[*of S T*]
cdcl-twl-subresolution-pcdcl-all-struct-invs[*of S T*]
cdcl-twl-subresolution-smaller-propa[*of S T*]
cdcl-twl-subresolution-twl-st-exception-inv[*of S T*]
cdcl-twl-subresolution-dup-enqueued[*of S T*]
cdcl-twl-subresolution-distinct-enqueued[*of S T*]
cdcl-twl-subresolution-propa-confl-cands-enqueued[*of S T*]
cdcl-twl-subresolution-propa-confl-cands-enqueued[*of S T*]
propa-confl-cands-enqueued-propa-confl-enqueued[*of S*]
propa-confl-cands-enqueued-propa-confl-enqueued[*of T*]
cdcl-twl-subresolution-conflict[*of S T*]
cdcl-twl-subresolution-twl-st-exception-inv[*of S T*]
cdcl-twl-subresolution-clauses-to-update-inv[*of S T*]
cdcl-twl-subresolution-past-invs[*of S T*] *assms*
unfolding *twl-struct-invs-def*
by *clarsimp*
qed

lemma *Propagated-eq-DecidedD*:
‹*Propagated L C # M1 = M @ Decided K # M'* ⟷
M ≠ [] ∧ *hd M = Propagated L C* ∧ *M1 = tl M @ Decided K # M'*›
by (*cases M*) *auto*

lemma *cdcl-twl-subresolution-twl-stgy-invs*:
assumes ‹*cdcl-twl-subresolution S T*›
‹*twl-struct-invs S*›
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
by (*induction rule*: *cdcl-twl-subresolution.induct*)
(*auto simp*: *twl-stgy-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def*
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
cdcl$_W$*-restart-mset.no-smaller-confl-def*
Propagated-eq-DecidedD)

inductive *cdcl-twl-unitres-true* :: ‹*'v twl-st* ⇒ *'v twl-st* ⇒ *bool*› **where**
‹*cdcl-twl-unitres-true* (*M*, *N* + {#*C*#}, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N*, *U*, *None*, *add-mset* (*clause C*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*L* ∈# *clause C*› ‹*get-level M L = 0*› ‹*L* ∈ *lits-of-l M*› |
‹*cdcl-twl-unitres-true* (*M*, *N* , *U*+ {#*C*#}, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N*, *U*, *None*, *NE*, *add-mset* (*clause C*) *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*L* ∈# *clause C*› ‹*get-level M L = 0*› ‹*L* ∈ *lits-of-l M*›

lemma *cdcl-twl-unitres-true-cdcl-unitres-true*:
‹*cdcl-twl-unitres-true S T* ⟹ *cdcl-unitres-true* (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)›

by (*force simp*: *cdcl-twl-unitres-true.simps cdcl-unitres-true.simps*
dest!: *multi-member-split*)

lemma *cdcl-twl-unitres-true-invs*:
‹*cdcl-twl-unitres-true S T* ⟹ *twl-st-inv S* ⟹ *twl-st-inv T*›
‹*cdcl-twl-unitres-true S T* ⟹ *valid-enqueued S* ⟹ *valid-enqueued T*›
‹*cdcl-twl-unitres-true S T* ⟹ *twl-st-exception-inv S* ⟹ *twl-st-exception-inv T*›
‹*cdcl-twl-unitres-true S T* ⟹ *no-duplicate-queued S* ⟹ *no-duplicate-queued T*›
‹*cdcl-twl-unitres-true S T* ⟹ *distinct-queued S* ⟹ *distinct-queued T*›
‹*cdcl-twl-unitres-true S T* ⟹ *confl-cands-enqueued S* ⟹ *confl-cands-enqueued T*›
‹*cdcl-twl-unitres-true S T* ⟹ *propa-cands-enqueued S* ⟹ *propa-cands-enqueued T*›
‹*cdcl-twl-unitres-true S T* ⟹ *clauses-to-update-inv S* ⟹ *clauses-to-update-inv T*›
‹*cdcl-twl-unitres-true S T* ⟹ *past-invs S* ⟹ *past-invs T*›
‹*cdcl-twl-unitres-true S T* ⟹ *get-conflict T* = *None*›
‹*cdcl-twl-unitres-true S T* ⟹ *cdcl*$_W$*-restart-mset.no-smaller-propa* (*state-of* (*pstate*$_W$*-of S*)) ⟹
cdcl$_W$*-restart-mset.no-smaller-propa* (*state-of* (*pstate*$_W$*-of T*))›
‹*cdcl-twl-unitres-true S T* ⟹ *pcdcl-all-struct-invs* (*pstate*$_W$*-of S*) ⟹ *pcdcl-all-struct-invs* (*pstate*$_W$*-of T*)›
apply (*auto simp*: *cdcl-twl-unitres-true.simps twl-st-inv.simps*
twl-exception-inv.simps filter-mset-empty-conv clauses-to-update-prop.simps
past-invs.simps dest: *cdcl-twl-unitres-true-cdcl-unitres-true*)[*11*]
apply (*auto dest*!: *cdcl-twl-unitres-true-cdcl-unitres-true*
pcdcl.intros(*8*) *pcdcl-all-struct-invs*)
done

lemma *cdcl-twl-unitres-true-twl-struct-invs*:
‹*cdcl-twl-unitres-true S T* ⟹ *twl-struct-invs S* ⟹ *twl-struct-invs T*›
using *cdcl-twl-unitres-true-invs*[*of S T*]
by (*auto simp*: *twl-struct-invs-def*)

lemma *cdcl-twl-unitres-true-twl-stgy-invs*:
assumes ‹*cdcl-twl-unitres-true S T*›
‹*twl-struct-invs S*›
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
by (*induction rule*: *cdcl-twl-unitres-true.induct*)
(*auto simp*: *twl-stgy-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def*
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
cdcl$_W$*-restart-mset.no-smaller-confl-def*
Propagated-eq-DecidedD)

inductive *cdcl-twl-unitres* :: ‹*'v twl-st* ⇒ *'v twl-st* ⇒ *bool*› **where**
‹*cdcl-twl-unitres* (*M*, *N* + {#*D*#}, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *add-mset E N*, *U*, *None*, *NE*, *UE*, *add-mset* (*clause D*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*count-decided M* = *0*› **and**
‹*clause D* = *C*+*C'*›
‹*add-mset* (*C*+*C'*) (*clauses N* + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C'*)›
‹¬*tautology C*› ‹*distinct-mset C*›
‹*struct-wf-twl-cls E*›
‹*Multiset.Ball* (*clause E*) (*undefined-lit M*)›
‹*clause E* = *C*› |
‹*cdcl-twl-unitres* (*M*, *N* + {#*D*#}, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K C* # *M*, *N*, *U*, *None*, *add-mset C NE*, *UE*, *add-mset* (*clause D*) *NS*, *US*, *N0*, *U0*,

{#}, *add-mset* (−*K*) *Q*)›
  if ‹*count-decided M* = *0*› **and**
    ‹*clause D* = *C*+*C*′›
    ‹*add-mset* (*C*+*C*′) (*clauses N* + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C*′)›
    ‹¬*tautology C*› ‹*distinct-mset C*›
    ‹*C* = {#*K*#}›
    ‹*undefined-lit M K*› |
‹*cdcl-twl-unitres* (*M*, *N*, *U* + {#*D*#}, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
    (*M*, *N*, *add-mset E U*, *None*, *NE*, *UE*, *NS*, *add-mset* (*clause D*) *US*, *N0*, *U0*, {#}, *Q*)›
  if ‹*count-decided M* = *0*› **and**
    ‹*clause D* = *C*+*C*′›
    ‹(*clauses N* + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C*′)›
    ‹¬*tautology C*› ‹*distinct-mset C*›
    ‹*struct-wf-twl-cls E*›
    ‹*clause E* = *C*›
    ‹*Multiset.Ball* (*clause E*) (*undefined-lit M*)›
    ‹*atms-of C* ⊆ *atms-of-ms* (*clause* ‘ *set-mset N*) ∪ *atms-of-mm NE* ∪ *atms-of-mm NS* ∪ *atms-of-mm N0*› |
‹*cdcl-twl-unitres* (*M*, *N*, *U* + {#*D*#}, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
    (*Propagated K C* # *M*, *N*, *U*, *None*, *NE*, *add-mset C UE*, *NS*, *add-mset* (*clause D*) *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
  if ‹*count-decided M* = *0*› **and**
    ‹*clause D* = *C*+*C*′›
    ‹*clauses N* + *NE* + *NS* + *N0* ⊨*psm mset-set* (*CNot C*′)›
    ‹¬*tautology C*› ‹*distinct-mset C*›
    ‹*C* = {#*K*#}›
    ‹*undefined-lit M K*›
    ‹*atms-of C* ⊆ *atms-of-ms* (*clause* ‘ *set-mset N*) ∪ *atms-of-mm NE* ∪ *atms-of-mm NS* ∪ *atms-of-mm N0*› |
‹*cdcl-twl-unitres* (*M*, *N*, *U* + {#*D*#}, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
    (*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, *NS*, *add-mset* (*clause D*) *US*, *N0*, *add-mset* {#} *U0*, {#}, {#})›
  if ‹*count-decided M* = *0*› **and**
    ‹(*clauses N* + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot* (*clause D*))› |
‹*cdcl-twl-unitres* (*M*, *N* + {#*D*#}, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
    (*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, *add-mset* (*clause D*) *NS*, *US*, *add-mset* {#} *N0*, *U0*, {#}, {#})›
  if ‹*count-decided M* = *0*› **and**
    ‹(*clauses* (*add-mset D N*) + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot* (*clause D*))›

lemma *cdcl-twl-unitres-cdcl-unitres*:
  ‹*cdcl-twl-unitres S T* ⟹ *cdcl-unitres* (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*)›
  by (*induction rule*: *cdcl-twl-unitres.cases*)
   (*fastforce simp*: *cdcl-unitres.simps*)+

lemma *cdcl-twl-unitres-invs*:
  ‹*cdcl-twl-unitres S T* ⟹ *twl-st-inv S* ⟹ *twl-st-inv T*›
  by (*induction rule*: *cdcl-twl-unitres.induct*)
   (*auto simp*: *twl-st-inv.simps twl-lazy-update-subresII*
    twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0
    twl-is-an-exception-add-mset-to-queue)

lemma *struct-wf-twl-cls-alt-def*:
  ‹*struct-wf-twl-cls C* ⟷ *distinct-mset* (*clause C*) ∧ *size* (*watched C*) = *2*›
  by (*cases C*) *auto*

lemma *cdcl-twl-unitres-struct-invs*:
  assumes ‹*cdcl-twl-unitres S T*› **and** ‹*twl-struct-invs S*› **and**

```
    ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init ((state_W-of S))›
  shows ‹twl-struct-invs T›
  unfolding twl-struct-invs-def
proof (intro conjI)
  have st-inv: ‹twl-st-inv S› and
    valid: ‹valid-enqueued S› and
    except: ‹twl-st-exception-inv S› and
    dup: ‹no-duplicate-queued S› and
    dist: ‹distinct-queued S› and
    clss: ‹clauses-to-update-inv S› and
    past: ‹past-invs S› and
    confl-cands: ‹confl-cands-enqueued S› and
    propa-cands: ‹propa-cands-enqueued S› and
    dupq: ‹no-duplicate-queued S› and
    distq: ‹distinct-queued S› and
    invs: ‹pcdcl-all-struct-invs (pstate_W-of S)› and
    propa: ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of S)›
    using assms(2) unfolding twl-struct-invs-def by fast+

  have n-d: ‹no-dup (get-trail S)›
    using invs unfolding pcdcl-all-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def
    by auto
  show st-inv-T: ‹twl-st-inv T› and ‹valid-enqueued T›
    using valid count-decided-ge-get-level[of ‹get-trail S›] assms(1) st-inv n-d except
    by (auto simp: twl-st-inv.simps twl-lazy-update-subresII
    twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0 get-level-cons-if
    twl-is-an-exception-add-mset-to-queue cdcl-twl-unitres.simps
    twl-exception-inv.simps filter-mset-empty-conv clauses-to-update-prop.simps
    past-invs.simps Decided-Propagated-in-iff-in-lits-of-l clause-alt-def
      uminus-lit-swap
    dest!: multi-member-split
      dest: mset-le-add-mset simp: undefined-notin; fail)+

  show ‹twl-st-exception-inv T›
    using count-decided-ge-get-level[of ‹get-trail S›] assms(1) except n-d
    by (auto simp: twl-st-inv.simps twl-lazy-update-subresII
    twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0 get-level-cons-if
    twl-is-an-exception-add-mset-to-queue cdcl-twl-unitres.simps
    twl-exception-inv.simps filter-mset-empty-conv clauses-to-update-prop.simps
    past-invs.simps Decided-Propagated-in-iff-in-lits-of-l clause-alt-def
      uminus-lit-swap
    dest!: multi-member-split
      dest: mset-le-add-mset simp: undefined-notin)
  show ‹clauses-to-update-inv T›
    using clss assms(1) n-d
    by (auto simp: cdcl-twl-unitres.simps filter-mset-empty-conv all-conj-distrib
      Decided-Propagated-in-iff-in-lits-of-l clauses-to-update-prop.simps
      split: if-splits dest: has-blit-Cons)
     (auto simp:  clause-alt-def Decided-Propagated-in-iff-in-lits-of-l split: if-splits; fail)+
  show invs-T: ‹pcdcl-all-struct-invs (pstate_W-of T)›
    using cdcl-twl-unitres-cdcl-unitres[OF assms(1)] invs assms(3)
    by (auto dest!: cdcl-unitres-learn-subsume rtranclp-pcdcl-all-struct-invs)
  show ‹past-invs T›
    using assms(1)
    by (auto simp: cdcl-twl-unitres.simps past-invs.simps Propagated-eq-DecidedD)
```

have *struct*: ‹∀ *C* ∈# *get-clauses S*. *struct-wf-twl-cls C*›
by (*use st-inv n-d propa-cands confl-cands* **in** ‹*cases S*; *auto simp*: *twl-st-inv.simps*; *fail*›)+
then have ‹*propa-confl-cands-enqueued S*›
by (*subst propa-confl-cands-enqueued-propa-confl-enqueued*[*of S*])
(*use st-inv n-d propa-cands confl-cands* **in** ‹*auto simp*: *twl-st-inv.simps*; *fail*›)+
with *assms*(*1*) **have** ‹*propa-confl-cands-enqueued T*›
using *n-d struct except*
by (*induction rule*: *cdcl-twl-unitres.cases*)
(*auto 5 4 intro*!: *propa-confl-cands-enqueued-propagate*
simp: *propa-confl-cands-enqueued-learn propa-confl-cands-enqueued-learn-empty*
dest: *propa-confl-cands-enqueuedD twl-exception-inv-skip-clause*
dest: *multi-member-split*[*of* ‹- :: - *twl-clause*›]
simp del: *propa-confl-cands-enqueued.simps*)
moreover have *struct*: ‹∀ *C* ∈# *get-clauses T*. *struct-wf-twl-cls C*›
by (*use st-inv-T* **in** ‹*cases T*; *auto simp*: *twl-st-inv.simps*; *fail*›)+
moreover have ‹*no-dup* (*get-trail T*)›
using *invs-T* **unfolding** *pcdcl-all-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by *auto*
ultimately show ‹*confl-cands-enqueued T*› **and** ‹*propa-cands-enqueued T*›
by (*subst* (*asm*) *propa-confl-cands-enqueued-propa-confl-enqueued*[*of T*]; *auto*; *fail*)+
show ‹*no-duplicate-queued T*› **and** ‹*distinct-queued T*›
using *dupq distq n-d assms*(*1*)
by (*force simp*: *cdcl-twl-unitres.simps past-invs.simps Propagated-eq-DecidedD undefined-notin*
dest!: *multi-member-split split-list dest*: *mset-le-add-mset*)+

show ‹*get-conflict T* ≠ *None* ⟶ *clauses-to-update T* = {#} ∧ *literals-to-update T* = {#}›
using *assms*(*1*)
by (*auto simp*: *cdcl-twl-unitres.simps past-invs.simps Propagated-eq-DecidedD*)

show ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of T*)›
using *assms*(*1*)
by (*auto simp*: *cdcl-twl-unitres.simps past-invs.simps Propagated-eq-DecidedD*
cdcl$_W$*-restart-mset.no-smaller-propa-def*)
qed

lemma *cdcl-twl-unitres-twl-stgy-invs*:
assumes ‹*cdcl-twl-unitres S T*›
‹*twl-struct-invs S*›
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
by (*induction rule*: *cdcl-twl-unitres.induct*)
(*auto simp*: *twl-stgy-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def*
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
cdcl$_W$*-restart-mset.no-smaller-confl-def*
Propagated-eq-DecidedD)

lemma *twl-exception-inv-add-subsumed*:
‹*twl-exception-inv* (*M1*, *N*, *U*, *D*, *NE*, *UE*, *add-mset* (*C'*) *NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
twl-exception-inv (*M1*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*twl-exception-inv* (*M1*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset* (*C'*) *US*, *N0*, *U0*, *WS*, *Q*) =
twl-exception-inv (*M1*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*intro ext*; *cases D*; *auto simp*: *twl-exception-inv.simps*)+

lemma *propa-confl-cands-enqueued-add-subsumed*:
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset* (*C'*) *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
propa-confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset* (*C'*) *US*, *N0*, *U0*, *WS*, *Q*) ⟷
propa-confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto*)+

lemma *propa-confl-cands-enqueued-remove-learnD*:
‹*propa-confl-cands-enqueued* (*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
propa-confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto*)

lemma *propa-confl-cands-enqueued-remove-learnD2*:
‹*propa-confl-cands-enqueued* (*M*, *add-mset C* (*add-mset C' N*), *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
propa-confl-cands-enqueued (*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*propa-confl-cands-enqueued* (*M*, *N*, *add-mset C* (*add-mset C' U*), *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
propa-confl-cands-enqueued (*M*, *N*, *add-mset C U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*propa-confl-cands-enqueued* (*M*, *add-mset C N*, (*add-mset C' U*), *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
propa-confl-cands-enqueued (*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto*)+

lemma *cdcl-subsumed-RI-all-struct-invs*:
‹*cdcl-subsumed-RI S T* ⟹ *pcdcl-all-struct-invs S* ⟹ *pcdcl-all-struct-invs T*›
by (*elim cdcl-subsumed-RID*)
(*auto simp add*: *cdcl-learn-clause-all-struct-inv cdcl-learn-clause-entailed-clss-inv*
cdcl-learn-clause-psubsumed-invs cdcl-subsumed-all-struct-inv cdcl-subsumed-entailed-clss-inv
cdcl-subsumed-psubsumed-invs pcdcl-all-struct-invs-def
intro: *pcdcl.intros*(*2*) *pcdcl.intros*(*4*) *pcdcl-clauses0-inv*)

lemma *cdcl-unitres-pcdcl-all-struct-invs*:
‹*cdcl-unitres S T* ⟹ *cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state-of S*) ⟹
pcdcl-all-struct-invs (*S*) ⟹
pcdcl-all-struct-invs (*T*)›
using *cdcl-unitres-learn-subsume rtranclp-pcdcl-all-struct-invs* **by** *blast*

lemma *cdcl-twl-unitres-pcdcl-all-struct-invs*:
‹*cdcl-twl-unitres S T* ⟹ *cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of S*) ⟹
pcdcl-all-struct-invs (*pstate$_W$-of S*) ⟹
pcdcl-all-struct-invs (*pstate$_W$-of T*)›
by (*drule cdcl-twl-unitres-cdcl-unitres*)
(*simp add*: *cdcl-unitres-pcdcl-all-struct-invs*)

lemma *cdcl-twl-subsumed-struct-invs*:
assumes ‹*cdcl-twl-subsumed S T*› **and** ‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
unfolding *twl-struct-invs-def*
proof (*intro conjI*)
have *st-inv*: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
except: ‹*twl-st-exception-inv S*› **and**
dup: ‹*no-duplicate-queued S*› **and**
dist: ‹*distinct-queued S*› **and**

clss: ‹*clauses-to-update-inv S*› **and**
past: ‹*past-invs S*› **and**
confl-cands: ‹*confl-cands-enqueued S*› **and**
propa-cands: ‹*propa-cands-enqueued S*› **and**
dupq: ‹*no-duplicate-queued S*› **and**
distq: ‹*distinct-queued S*› **and**
invs: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of S*)› **and**
propa: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of S*)› **and**
confl: ‹*get-conflict S* ≠ *None* ⟶ *clauses-to-update S* = {#} ∧ *literals-to-update S* = {#}› **and**
smaller: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of S*)›
using *assms*(*2*) **unfolding** *twl-struct-invs-def* **by** *fast+*

have *n-d*: ‹*no-dup* (*get-trail S*)›
using *invs* **unfolding** *pcdcl-all-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by *auto*
show *st-inv-T*: ‹*twl-st-inv T*› **and** ‹*valid-enqueued T*›
using *valid count-decided-ge-get-level*[*of* ‹*get-trail S*›] *assms*(*1*) *st-inv n-d except*
by (*auto simp*: *twl-st-inv.simps twl-lazy-update-subresII*
twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0 get-level-cons-if
twl-is-an-exception-add-mset-to-queue cdcl-twl-subsumed.simps
twl-exception-inv.simps filter-mset-empty-conv clauses-to-update-prop.simps
past-invs.simps Decided-Propagated-in-iff-in-lits-of-l clause-alt-def
uminus-lit-swap
dest!: *multi-member-split*
dest: *mset-le-add-mset simp*: *undefined-notin*; *fail*)+

show ‹*twl-st-exception-inv T*›
using *count-decided-ge-get-level*[*of* ‹*get-trail S*›] *assms*(*1*) *except n-d*
by (*cases* ‹*get-conflict S*›)
(*auto simp*: *twl-st-inv.simps twl-lazy-update-subresII*
twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0 get-level-cons-if
twl-is-an-exception-add-mset-to-queue cdcl-twl-subsumed.simps
twl-exception-inv.simps filter-mset-empty-conv clauses-to-update-prop.simps
past-invs.simps Decided-Propagated-in-iff-in-lits-of-l clause-alt-def
uminus-lit-swap
dest!: *multi-member-split*
dest: *mset-le-add-mset simp*: *undefined-notin*)

show ‹*clauses-to-update-inv T*›
using *clss assms*(*1*) *n-d*
by (*cases* ‹*get-conflict S*›)
(*auto simp*: *cdcl-twl-subsumed.simps filter-mset-empty-conv all-conj-distrib*
Decided-Propagated-in-iff-in-lits-of-l clauses-to-update-prop.simps
split: *if-splits dest*: *has-blit-Cons*)
show *invs-T*: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of T*)›
using *cdcl-twl-subsumed-cdcl-subsumed*[*OF assms*(*1*)] *assms*(*2*) *invs*
by (*auto dest*!: *cdcl-unitres-learn-subsume rtranclp-pcdcl-all-struct-invs cdcl-subsumed-RI-all-struct-invs*
simp: *cdcl-subsumed-all-struct-inv cdcl-subsumed-entailed-clss-inv cdcl-subsumed-psubsumed-invs*
pcdcl-all-struct-invs-def
intro: *pcdcl.intros*(*4*) *pcdcl-clauses0-inv*)

show ‹*past-invs T*›
using *assms*(*1*) *past*
by (*auto simp*: *cdcl-twl-subsumed.simps past-invs.simps Propagated-eq-DecidedD all-conj-distrib*
twl-lazy-update-subresII

twl-lazy-update-undefined watched-literals-false-of-max-level-lvl0 get-level-cons-if
twl-is-an-exception-add-mset-to-queue
twl-exception-inv.simps filter-mset-empty-conv clauses-to-update-prop.simps
Decided-Propagated-in-iff-in-lits-of-l clause-alt-def
dest!: *multi-member-split*
dest: *mset-le-add-mset simp*: *undefined-notin*)

have *propa-confl-cands-enqueued-IR*: ‹*propa-confl-cands-enqueued* (*M*, *add-mset C′ N*, *add-mset C U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟹
propa-confl-cands-enqueued (*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)› **for** *M C′ N C U D NE UE US Q NS N0 U0*
by (*cases D*) (*auto simp*:)
have *struct*: ‹∀ *C* ∈# *get-clauses S*. *struct-wf-twl-cls C*›
by (*use st-inv n-d propa-cands confl-cands* **in** ‹*cases S*; *auto simp*: *twl-st-inv.simps*; *fail*›)+
then have ‹*propa-confl-cands-enqueued S*›
by (*subst propa-confl-cands-enqueued-propa-confl-enqueued*[*of S*])
(*use st-inv n-d propa-cands confl-cands* **in** ‹*auto simp*: *twl-st-inv.simps*; *fail*›)+
with *assms*(*1*) **have** ‹*propa-confl-cands-enqueued T*›
using *n-d struct except*
by (*induction rule*: *cdcl-twl-subsumed.cases*)
(*auto 5 4 intro*!: *propa-confl-cands-enqueued-propagate*
simp: *propa-confl-cands-enqueued-learn twl-exception-inv-add-subsumed*
propa-confl-cands-enqueued-add-subsumed add-mset-commute
dest: *propa-confl-cands-enqueuedD twl-exception-inv-skip-clause*
propa-confl-cands-enqueued-remove-learnD2 propa-confl-cands-enqueued-IR
dest: *multi-member-split*[*of* ‹- :: - *twl-clause*›]
simp del: *propa-confl-cands-enqueued.simps*)
moreover have *struct*: ‹∀ *C* ∈# *get-clauses T*. *struct-wf-twl-cls C*›
by (*use st-inv-T* **in** ‹*cases T*; *auto simp*: *twl-st-inv.simps*; *fail*›)+
moreover have ‹*no-dup* (*get-trail T*)›
using *invs-T* **unfolding** *pcdcl-all-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by *auto*
ultimately show ‹*confl-cands-enqueued T*› **and** ‹*propa-cands-enqueued T*›
by (*subst* (*asm*) *propa-confl-cands-enqueued-propa-confl-enqueued*[*of T*]; *auto*; *fail*)+
show ‹*no-duplicate-queued T*› **and** ‹*distinct-queued T*›
using *dupq distq n-d assms*(*1*)
by (*force simp*: *cdcl-twl-subsumed.simps past-invs.simps Propagated-eq-DecidedD undefined-notin*
dest!: *multi-member-split split-list dest*: *mset-le-add-mset*)+

show ‹*get-conflict T* ≠ *None* ⟶ *clauses-to-update T* = {#} ∧ *literals-to-update T* = {#}›
using *assms*(*1*) *confl*
by (*auto simp*: *cdcl-twl-subsumed.simps past-invs.simps Propagated-eq-DecidedD*)

show ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of T*)›
using *assms*(*1*) *smaller*
apply (*auto simp*: *cdcl-twl-subsumed.simps past-invs.simps Propagated-eq-DecidedD*
cdcl$_W$*-restart-mset.no-smaller-propa-def*)
apply *blast*+
done
qed

lemma *cdcl-subresolutions-entailed-by-init*:
‹*cdcl-subresolution S T* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of T*)›

apply (*induction rule*: *cdcl-subresolution.induct*)
subgoal by (*auto simp*: $cdcl_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init-def*
insert-commute)
subgoal for *M C C′ N U L D NE UE NS US N0 U0*
using *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*[*of* ‹*set-mset N* ∪ *set-mset NE* ∪ *set-mset NS* ∪ *set-mset N0*› *L C′ C*]
true-clss-cls-cong-set-mset[*of* ‹*N* + *NE* + *NS* + *N0*› ‹*C*+*C′*› *C′*]
by (*auto simp*: $cdcl_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init-def*
insert-commute subset-mset.le-iff-add ac-simps)
subgoal for *M C C′ N L U D NE UE NS US N0 U0*
using *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*[*of* ‹*set-mset* (*N* + {#*add-mset L C*#}) ∪ *set-mset NE* ∪ *set-mset NS* ∪ *set-mset N0*› *L C′ C*]
true-clss-cls-cong-set-mset[*of* ‹(*N* + {#*add-mset L C*#}) + *NE* + *NS* + *N0*› ‹*C*+*C′*› *C′*]
by (*auto simp*: $cdcl_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init-def*
insert-commute subset-mset.le-iff-add ac-simps)
subgoal for *M C C′ N L U D NE UE NS US N0 U0*
using *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*[*of*
‹*set-mset* (*N* + {#*add-mset L C*#}) ∪ *set-mset NE* ∪ *set-mset NS* ∪ *set-mset N0*› *L C′ C*]
true-clss-cls-cong-set-mset[*of* ‹(*N* + {#*add-mset L C*#}) + *NE* + *NS* + *N0*› ‹*C*+*C′*› *C*]
by (*force simp*: $cdcl_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init-def*
insert-commute ac-simps)
done

inductive *cdcl-twl-promote-false* :: ‹′*v twl-st* ⇒ ′*v twl-st* ⇒ *bool*› **where**
‹*cdcl-twl-promote-false* (*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
(*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, *NS*, *US*, *add-mset* {#} *N0*, *U0*, *WS*, *Q*)›
if ‹*clause C* = {#}› ‹*count-decided M* = *0*›|
‹*cdcl-twl-promote-false* (*M*, *N*, *add-mset C U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
(*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, *NS*, *US*, *N0*, *add-mset* {#} *U0*, *WS*, *Q*)›
if ‹*clause C* = {#}› ‹*count-decided M* = *0*›

lemma *cdcl-twl-promote-false-cdcl-promote-false*:
‹*cdcl-twl-promote-false S T* ⟹ *cdcl-promote-false* ($pstate_W$*-of S*) ($pstate_W$*-of T*)›
by (*auto simp*: *cdcl-twl-promote-false.simps cdcl-promote-false.simps*)

lemma *cdcl-twl-promote-false-twl-stgy-invs*:
assumes ‹*cdcl-twl-promote-false S T*›
‹*twl-struct-invs S*›
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
by (*induction rule*: *cdcl-twl-promote-false.induct*)
(*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.no-smaller-confl-def clauses-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)

lemma *cdcl-subsumed-RI-stgy-invs*:
‹*cdcl-subsumed-RI* ($pstate_W$*-of S*) ($pstate_W$*-of T*) ⟹ *twl-stgy-invs S* ⟹
twl-stgy-invs T›
apply (*cases rule*: *cdcl-subsumed-RI.cases*, *assumption*)
apply (*cases S*; *cases T*; *auto simp*: *twl-stgy-invs-def*
$cdcl_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def* $cdcl_W$*-restart-mset.no-smaller-confl-def*
clauses-def $cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
all-conj-distrib)
apply (*metis member-add-mset set-image-mset*)
apply (*metis member-add-mset set-image-mset*)

apply (*metis member-add-mset set-image-mset*)
apply (*metis image-mset-add-mset member-add-mset multi-member-split set-image-mset*)
done

lemma *cdcl-twl-subsumed-stgy-invs*:
‹*cdcl-twl-subsumed S T* ⟹
twl-struct-invs S ⟹
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (*state$_W$-of S*) ⟹
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (*state$_W$-of T*) ⟹
twl-stgy-invs S ⟹ *twl-stgy-invs T*›
apply (*drule cdcl-twl-subsumed-cdcl-subsumed*)
apply (*elim disjE*)
apply (*simp add*: *state-of-cdcl-subsumed twl-stgy-invs-def*)
apply (*simp add*: *cdcl-subsumed-RI-stgy-invs*)
done

lemma *cdcl-twl-subsumed-twl-stgy-invs*:
assumes ‹*cdcl-twl-subsumed S T*›
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
by (*metis* (*no-types*, *lifting*) *assms cdcl-subsumed-RI-stgy-invs*
cdcl-twl-subsumed-cdcl-subsumed state$_W$-of-def state-of-cdcl-subsumed twl-stgy-invs-def)

inductive *cdcl-twl-pure-remove* :: ‹'*v twl-st* ⇒ '*v twl-st* ⇒ *bool*› **where**
‹*cdcl-twl-pure-remove* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated L* {#*L*#} # *M*, *N*, *U*, *None*, *add-mset* {#*L*#} *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *add-mset* (−*L*) *Q*)›
if ‹*count-decided M* = *0*›
‹−*L* ∉ ⋃ ((*set-mset o clause*) ‘ *set-mset N*)›
‹*atm-of L* ∈ *atms-of-mm* (*clauses N* + *NE* + *NS* + *N0*)›
‹*undefined-lit M L*›

lemma *cdcl-twl-pure-remove-cdcl-pure-remove*:
‹*cdcl-twl-pure-remove S T* ⟹ *cdcl-pure-literal-remove* (*pstate$_W$-of S*) (*pstate$_W$-of T*)›
by (*auto simp*: *cdcl-twl-pure-remove.simps cdcl-pure-literal-remove.simps*
dest!: *multi-member-split*)

lemma *cdcl-twl-pure-remove-twl-struct-invs*:
assumes *pure*: ‹*cdcl-twl-pure-remove S T*› **and**
‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
proof −
have *st-inv*: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
smaller: ‹*cdcl$_W$-restart-mset.no-smaller-propa* (*state$_W$-of S*)› **and**
exception: ‹*twl-st-exception-inv S*› **and**
‹*pcdcl-all-struct-invs* (*pstate$_W$-of S*)› **and**
dup: ‹*no-duplicate-queued S*›
‹*distinct-queued S*›
‹*confl-cands-enqueued S*›
‹*propa-cands-enqueued S*› **and**
invs: ‹*pcdcl-all-struct-invs* (*pstate$_W$-of S*)› **and**
upd-inv: ‹*clauses-to-update-inv S*›

```
  using assms unfolding twl-struct-invs-def by fast+
have n-d: ‹no-dup (get-trail S)›
  using invs unfolding pcdcl-all-struct-invs-def
    cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
    cdcl_W-restart-mset.cdcl_W-M-level-inv-def
  by auto
have wf: ‹∀ C ∈# get-clauses S. struct-wf-twl-cls C›
  using st-inv by (cases S) (auto simp: twl-st-inv.simps)
have propa-confl: ‹propa-confl-cands-enqueued S›
  by (subst propa-confl-cands-enqueued-propa-confl-enqueued)
   (use wf dup n-d in auto)
have ‹twl-st-inv T›
  using pure st-inv
  by (auto simp: cdcl-twl-pure-remove.simps twl-st-inv.simps
    twl-is-an-exception-add-mset-to-queue watched-literals-false-of-max-level-prop-lvl0
    intro!: twl-lazy-update-subresII
    dest: multi-member-split)
moreover have ‹valid-enqueued T›
  using valid pure
  by (auto simp: cdcl-twl-pure-remove.simps twl-st-inv.simps
    get-level-cons-if)
moreover have ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of T)›
  using pure smaller
  by (auto simp: cdcl_W-restart-mset.no-smaller-propa-def
    cdcl-twl-pure-remove.simps Propagated-eq-DecidedD)
moreover have ‹twl-st-exception-inv T›
  using pure n-d exception apply −
  by (cases rule: cdcl-twl-pure-remove.cases, assumption)
   (auto simp: cdcl-twl-pure-remove.simps Propagated-eq-DecidedD
      twl-exception-inv.simps uminus-lit-swap
    intro!: propa-confl-cands-enqueued-propagate
    dest!: no-has-blit-propagate'
    dest!: multi-member-split[of ‹- :: - clause›])
 moreover have ‹no-duplicate-queued T›
   ‹distinct-queued T›
   using pure dup apply (auto simp: no-duplicate-queued.simps
     cdcl-twl-pure-remove.simps
     dest!: multi-member-split[of - ‹- :: - clause›]
     dest: mset-subset-eq-insertD msed-map-invR)[]
   using pure dup apply (auto simp: no-duplicate-queued.simps
     cdcl-twl-pure-remove.simps undefined-notin
     dest!: multi-member-split[of - ‹- :: - clause›]
     dest: mset-subset-eq-insertD dest: msed-map-invR)[]
   apply (solves ‹frule mset-subset-eq-insertD; clarsimp simp: undefined-notin›)+
   done
 moreover {
   have ‹propa-confl-cands-enqueued T›
     using pure dup propa-confl wf exception n-d
     by (auto simp: cdcl-twl-pure-remove.simps
       twl-exception-inv.simps
       simp del: propa-confl-cands-enqueued.simps
       intro!: propa-confl-cands-enqueued-propagate)[]
   then have
    ‹confl-cands-enqueued T ∧ propa-cands-enqueued T›
  by (subst (asm) propa-confl-cands-enqueued-propa-confl-enqueued)
   (use pure wf dup n-d in ‹auto simp: cdcl-twl-pure-remove.simps›)[3]
```

```
  }
  moreover have ‹get-conflict T ≠ None ⟶ clauses-to-update T = {#} ∧ literals-to-update T = {#}›
    using pure by (auto simp: cdcl-twl-pure-remove.simps)
  moreover have ‹clauses-to-update-inv T›
    using n-d pure upd-inv
    by (auto simp: cdcl-twl-pure-remove.simps filter-mset-empty-conv all-conj-distrib
      Decided-Propagated-in-iff-in-lits-of-l clauses-to-update-prop.simps
      split: if-splits dest: has-blit-Cons)
     (auto simp:  clause-alt-def Decided-Propagated-in-iff-in-lits-of-l split: if-splits; fail)+
  moreover have ‹past-invs T›
    using pure by (auto simp: cdcl-twl-pure-remove.simps past-invs.simps
      Propagated-eq-DecidedD)
  moreover have ‹pcdcl-all-struct-invs (pstate_W-of T)›
    using cdcl-twl-pure-remove-cdcl-pure-remove invs pcdcl.intros(10) pcdcl-all-struct-invs pure
    by blast
  ultimately show ?thesis
    unfolding twl-struct-invs-def
    by simp
qed

lemma cdcl-twl-pure-remove-twl-stgy-invs:
  assumes pure: ‹cdcl-twl-pure-remove S T› and
    ‹twl-stgy-invs S›
  shows ‹twl-stgy-invs T›
  using assms by (auto simp: cdcl-twl-pure-remove.simps
    twl-stgy-invs-def cdcl_W-restart-mset.conflict-non-zero-unless-level-0-def
    cdcl_W-restart-mset.cdcl_W-stgy-invariant-def Propagated-eq-DecidedD
    cdcl_W-restart-mset.no-smaller-confl-def)

end
theory Watched-Literals-Transition-System-Restart
imports CDCL.Pragmatic-CDCL-Restart Watched-Literals-Transition-System
  Watched-Literals-Transition-System-Inprocessing
begin
```

Unlike the basic CDCL, it does not make any sense to fully restart the trail: the part propagated at level 0 (only the part due to unit clauses) has to be kept. Therefore, we allow fast restarts (i.e. a restart where part of the trail is reused).

There are two cases:

- either the trail is strictly decreasing;
- or it is kept and the number of clauses is strictly decreasing.

This ensures that *something* changes to prove termination.

In practice, there are two types of restarts that are done:

- First, a restart can be done to enforce that the SAT solver goes more into the direction expected by the decision heuristics.
- Second, a full restart can be done to simplify inprocessing and garbage collection of the memory: instead of properly updating the trail, we restart the search. This is not necessary (i.e., glucose and minisat do not do it), but it simplifies the proofs by allowing to move clauses without taking care of updating references in the trail. Moreover, as this happens "rarely" (around once every few thousand conflicts), it should not matter too much.

Restarts are the "local search" part of all modern SAT solvers.

inductive *cdcl-twl-restart* :: ‹'v twl-st ⇒ 'v twl-st ⇒ bool› **where**
restart-trail:
‹*cdcl-twl-restart* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M'*, *N'*, *U'*, *None*, *NE* + *clauses NE'*, *UE''*, *NS*, {#},*N0*, {#},
{#}, {#})›
if
‹(*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
‹*U'* + *UE'* ⊆# *U*› **and**
‹*N* = *N'* + *NE'*› **and**
‹∀ *E*∈#*NE'*+*UE'*. ∃ *L*∈#*clause E*. *L* ∈ *lits-of-l M'* ∧ *get-level M' L* = 0›
‹∀ *L E*. *Propagated L E* ∈ *set M'* ⟶ *E* ∈# *clause* '# (*N* + *U'*) + (*NE*+ *clauses NE'*) + *UE''*›
‹*UE''* ⊆# *UE* + *clauses UE'*› |
restart-clauses:
‹*cdcl-twl-restart* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N'*, *U'*, *None*, *NE* + *clauses NE'*, *UE''*, *NS*, *US'*, *N0*, {#}, {#}, *Q*)›
if
‹*U'* + *UE'* ⊆# *U*› **and**
‹*N* = *N'* + *NE'*› **and**
‹∀ *E*∈#*NE'*+*UE'*. ∃ *L*∈#*clause E*. *L* ∈ *lits-of-l M* ∧ *get-level M L* = 0›
‹∀ *L E*. *Propagated L E* ∈ *set M* ⟶ *E* ∈# *clause* '# (*N* + *U'*) + (*NE*+ *clauses NE'*) + *UE''*›
‹*US'* = {#}›
‹*UE''* ⊆# *UE* + *clauses UE'*›

inductive-cases *cdcl-twl-restartE*: ‹*cdcl-twl-restart S T*›

inductive *cdcl-twl-restart-only* :: ‹'v twl-st ⇒ 'v twl-st ⇒ bool› **where**
restart-trail:
‹*cdcl-twl-restart-only* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
if
‹(*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› |
norestart-trail:
‹*cdcl-twl-restart-only* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›

lemma *past-invs-simps-emptyU0*:
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀ *M1 M2 K*. *M* = *M2* @ *Decided K* # *M1* ⟶ (
(∀ *C* ∈# *N* + *U*. *twl-lazy-update M1 C* ∧
watched-literals-false-of-max-level M1 C ∧
twl-exception-inv (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, {#}, {#}, {#}) *C*) ∧
confl-cands-enqueued (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, {#}, {#}, {#}) ∧
propa-cands-enqueued (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, {#}, {#}, {#}) ∧
clauses-to-update-inv (*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, {#}, {#}, {#})))›
unfolding *past-invs.simps* **by** (*auto simp*: *twl-exception-inv.simps*)

lemma *cdcl-twl-restart-pcdcl*: ‹*cdcl-twl-restart S T* ⟹ *pcdcl-restart* (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*)›
by (*induction rule*: *cdcl-twl-restart.induct*)
(*auto simp add*: *pcdcl-restart.simps dest*: *image-mset-subseteq-mono*
intro!: *exI*[*of* - ‹*clauses* -›])

lemma *cdcl-twl-restart-twl-struct-invs*:

assumes
‹*cdcl-twl-restart S T*› **and**
‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
using *assms cdcl-twl-restart-pcdcl*[*OF assms*(*1*)]
proof (*induction rule*: *cdcl-twl-restart.induct*)
case (*restart-trail K M' M2 M U' UE' U N N' NE' NE UE'' UE NS US N0 U0 Q*)
note *decomp* = *this*(*1*) **and** *learned'* = *this*(*2*) **and** *N* = *this*(*3*) **and**
has-true = *this*(*4*) **and** *kept* = *this*(*5*) **and** *incl* = *this*(*6*) **and** *inv* = *this*(*7*) **and** *st'* = *this*(*8*)
let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
let *?S'* = ‹(*M'*, *N'*, *U'*, *None*, *NE*+ *clauses NE'*, *UE''*, *NS*, {#}, *N0*, {#}, {#}, {#})›
have *learned*: ‹*U'* ⊆# *U*›
using *learned'* **by** (*rule mset-le-decr-left1*)
have
twl-st-inv: ‹*twl-st-inv ?S*› **and**
‹*valid-enqueued ?S*› **and**
struct-inv: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of ?S*)› **and**
smaller: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of ?S*)› **and**
‹*twl-st-exception-inv ?S*› **and**
no-dup-q: ‹*no-duplicate-queued ?S*› **and**
dist: ‹*distinct-queued ?S*› **and**
‹*confl-cands-enqueued ?S*› **and**
‹*propa-cands-enqueued ?S*› **and**
‹*get-conflict ?S* ≠ *None* ⟶
clauses-to-update ?S = {#} ∧
literals-to-update ?S = {#}› **and**
to-upd: ‹*clauses-to-update-inv ?S*› **and**
past: ‹*past-invs ?S*›
using *inv* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *clarsimp+*
have
ex: ‹(∀ *C*∈#*N* + *U*. *twl-lazy-update M' C* ∧
watched-literals-false-of-max-level M' C ∧
twl-exception-inv (*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*)› **and**
conf-cands: ‹*confl-cands-enqueued* (*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
propa-cands: ‹*propa-cands-enqueued* (*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
clss-to-upd: ‹*clauses-to-update-inv* (*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *past get-all-ann-decomposition-exists-prepend*[*OF decomp*] **unfolding** *past-invs.simps*
by *force+*

have *excp-inv*: ‹*twl-st-exception-inv* (*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *ex* **unfolding** *twl-st-exception-inv.simps* **by** *blast+*
have *twl-st-inv'*: ‹*twl-st-inv* (*M'*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *ex learned twl-st-inv*
unfolding *twl-st-exception-inv.simps twl-st-inv.simps*
by *auto*
have *n-d*: ‹*no-dup M*›
using *struct-inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*auto simp*: *trail.simps*)
obtain *M3* **where**
M: ‹*M* = *M3* @ *M2* @ *Decided K* # *M'*›
using *decomp* **by** *blast*
define *M3'* **where** ‹*M3'* = *M3* @ *M2*›
then have *M3'*: ‹*M* = *M3'* @ *Decided K* # *M'*›
unfolding *M* **by** *auto*
have *a*: ‹*N* ⊆# *N*› **and** *NN'*: ‹*N'* ⊆# *N*› **using** *N* **by** *auto*

```
  have past-invs: ‹past-invs ?S'›
    unfolding past-invs.simps
  proof (intro conjI impI allI)
    fix M1 M2 K'
    assume H:‹M' = M2 @ Decided K' # M1›
    let ?U = ‹(M1, N, U, None, NE, UE, NS, US, N0, {#}, {#}, {#})›
    let ?U' = ‹(M1, N', U', None, NE+clauses NE', UE'', NS, {#}, N0, {#}, {#}, {#})›
    have ‹M = (M3' @ Decided K # M2) @ Decided K' # M1›
      using H M3' by simp
    then have
      1: ‹∀ C∈#N + U.
          twl-lazy-update M1 C ∧
          watched-literals-false-of-max-level M1 C ∧
          twl-exception-inv ?U C› and
      2: ‹confl-cands-enqueued ?U› and
      3: ‹propa-cands-enqueued ?U› and
      4: ‹clauses-to-update-inv ?U›
      using past unfolding past-invs-simps-emptyU0 by blast+
    show ‹∀ C∈#N' + U'.
        twl-lazy-update M1 C ∧
        watched-literals-false-of-max-level M1 C ∧
        twl-exception-inv ?U' C›
      using 1 learned twl-st-exception-inv-mono[OF learned NN', of M1 None NE ‹UE›
        NS US N0 ‹{#}› ‹{#}› ‹{#}› ‹NE+clauses NE'› ‹UE''›] N
       twl-st-exception-inv-U0-mono[OF - twl-st-exception-inv-subsumed-mono[of ‹{#}› US M1 N' U'
None ‹NE+clauses NE'›
        ‹UE''› NS N0 ‹{#}› ‹{#}› ‹{#}›]]
       by auto

    show ‹confl-cands-enqueued ?U'›
      by (rule confl-cands-enqueued-subsumed-mono[OF - confl-cands-enqueued-mono[OF learned NN'
2]])
        auto
    show ‹propa-cands-enqueued ?U'›
     by (rule propa-cands-enqueued-subsumed-mono[OF - propa-cands-enqueued-mono[OF learned NN'
3]])
        auto
    show ‹clauses-to-update-inv ?U'›
      using 4 learned by (auto simp add: filter-mset-empty-conv N)
  qed
  have clss-to-upd: ‹clauses-to-update-inv ?S'›
    using clss-to-upd learned by (auto simp add: filter-mset-empty-conv N)

  have struct-inv': ‹pcdcl-all-struct-invs (pstate_W-of ?S')›
    using pcdcl-restart-pcdcl-all-struct-invs[OF st' struct-inv] by simp
  have smaller': ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of ?S')›
    using pcdcl-restart-no-smaller-propa'[OF st'] smaller by simp
  show ?case
    unfolding twl-struct-invs-def
    apply (intro conjI)
    subgoal
      using twl-st-inv-subsumed-mono[OF - twl-st-inv-mono[OF learned NN' twl-st-inv']]
      by (auto simp: ac-simps N)
    subgoal by simp
    subgoal by (rule struct-inv')
    subgoal by (rule smaller')
```

subgoal by (*rule twl-st-exception-inv-U0-mono*[*OF - twl-st-exception-inv-subsumed-mono*[*OF - twl-st-exception-inv-mono*[*OF learned NN' excp-inv*]]])
auto
subgoal using *no-dup-q* **by** *auto*
subgoal using *dist* **by** *auto*
subgoal
by (*rule confl-cands-enqueued-U0-mono*[*OF - confl-cands-enqueued-subsumed-mono*[*OF - confl-cands-enqueued-mono learned NN' conf-cands*]]])
auto
subgoal by (*rule propa-cands-enqueued-U0-mono*[*OF - propa-cands-enqueued-subsumed-mono*[*OF - propa-cands-enqueued-mono*[*OF learned NN' propa-cands*]]])
auto
subgoal by *simp*
subgoal by (*rule clss-to-upd*)
subgoal by (*rule past-invs*)
done
next
case (*restart-clauses U' UE' U N N' NE' M NE UE'' US' UE NS US N0 U0 Q*)
note *learned'* = *this*(*1*) **and** *N* = *this*(*2*) **and** *has-true* = *this*(*3*) **and** *kept* = *this*(*4*) **and**
US = *this*(*5*) **and** *incl* = *this*(*6*) **and** *invs* = *this*(*7*) **and** *st'* = *this*(*8*)
let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
let *?T* = ‹(*M*, *N'*, *U'*, *None*, *NE*+*clauses NE'*, *UE''*, *NS*, *US'*, *N0*, {#}, {#}, *Q*)›
have *learned*: ‹*U'* ⊆# *U*›
using *learned'* **by** (*rule mset-le-decr-left1*)
have
twl-st-inv: ‹*twl-st-inv ?S*› **and**
valid: ‹*valid-enqueued ?S*› **and**
struct-inv: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of ?S*)› **and**
smaller: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of ?S*)› **and**
excp-inv: ‹*twl-st-exception-inv ?S*› **and**
no-dup-q: ‹*no-duplicate-queued ?S*› **and**
dist: ‹*distinct-queued ?S*› **and**
confl-cands: ‹*confl-cands-enqueued ?S*› **and**
propa-cands: ‹*propa-cands-enqueued ?S*› **and**
‹*get-conflict ?S* ≠ *None* ⟶
clauses-to-update ?S = {#} ∧
literals-to-update ?S = {#}› **and**
to-upd: ‹*clauses-to-update-inv ?S*› **and**
past: ‹*past-invs ?S*›
using *invs* **unfolding** *twl-struct-invs-def* **by** *clarify*+
have *n-d*: ‹*no-dup M*›
using *struct-inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def pcdcl-all-struct-invs-def* **by** (*auto simp*: *trail.simps*)
have *valid'*: ‹*valid-enqueued ?T*›
using *valid* **by** *auto*

have *NN'*: ‹*N'* ⊆# *N*› **unfolding** *N* **by** *auto*
have *past-invs*: ‹*past-invs ?T*›
using *past* **unfolding** *past-invs.simps*
proof (*intro conjI impI allI*)
fix *M1 M2 K'*
assume *H*:‹*M* = *M2* @ *Decided K'* # *M1*›
let *?U* = ‹(*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, {#}, {#}, {#})›
let *?U'* = ‹(*M1*, *N'*, *U'*, *None*, *NE*+*clauses NE'*, *UE''*, *NS*, *US'*, *N0*, {#}, {#}, {#})›
have
1: ‹∀ *C*∈#*N* + *U*.

```
        twl-lazy-update M1 C ∧
        watched-literals-false-of-max-level M1 C ∧
        twl-exception-inv ?U C› and
    2: ‹confl-cands-enqueued ?U› and
    3: ‹propa-cands-enqueued ?U› and
    4: ‹clauses-to-update-inv ?U›
    using H past unfolding past-invs-simps-emptyU0 by blast+
  show ‹∀ C∈#N' + U'.
      twl-lazy-update M1 C ∧
      watched-literals-false-of-max-level M1 C ∧
      twl-exception-inv ?U' C›
    using 1 learned twl-st-exception-inv-subsumed-mono[OF -
        twl-st-exception-inv-mono[OF learned NN', of M1 None NE UE NS US N0 ‹{#}› ‹{#}›
‹{#}›
        ‹NE + clauses NE'› ‹UE''›], of ‹US'›] N US
    by (auto split: if-splits)
  show ‹confl-cands-enqueued ?U'›
   by (rule confl-cands-enqueued-subsumed-mono[OF - confl-cands-enqueued-mono[OF learned NN'
2]])
     (use US in ‹auto split: if-splits›)
  show ‹propa-cands-enqueued ?U'›
   by (rule propa-cands-enqueued-subsumed-mono[OF - propa-cands-enqueued-mono[OF learned NN'
3]])
     (use US in ‹auto split: if-splits›)
  show ‹clauses-to-update-inv ?U'›
   using 4 learned by (auto simp add: filter-mset-empty-conv N)
qed

let ?S' = ‹(M, N', U', None, NE + clauses NE', UE'', NS, US', N0, {#}, {#}, Q)›
have struct-inv': ‹pcdcl-all-struct-invs (pstate_W-of ?S')›
  using pcdcl-restart-pcdcl-all-struct-invs[OF st' struct-inv] by simp
have smaller': ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of ?S')›
  using pcdcl-restart-no-smaller-propa'[OF st'] smaller by simp

have clss-to-upd: ‹clauses-to-update-inv ?T›
  using to-upd learned by (auto simp add: filter-mset-empty-conv N ac-simps)


show ?case
 unfolding twl-struct-invs-def
 apply (intro conjI)
 subgoal using twl-st-inv-U0-subsumed-mono[OF -
   twl-st-inv-subsumed-mono[OF - twl-st-inv-mono[OF learned NN' twl-st-inv]]] US
   by (auto simp: ac-simps N split: if-splits)
 subgoal by (rule valid')
 subgoal by (rule struct-inv')
 subgoal by (rule smaller')
 subgoal by (rule twl-st-exception-inv-U0-mono[OF -
   twl-st-exception-inv-subsumed-mono[OF - twl-st-exception-inv-mono[OF learned NN' excp-inv]]])
   (use US in ‹auto split: if-splits›)
 subgoal using no-dup-q by auto
 subgoal using dist by auto
  subgoal by (rule confl-cands-enqueued-U0-mono[OF - confl-cands-enqueued-subsumed-mono[OF -
confl-cands-enqueued-mono[OF learned NN' confl-cands]]])
    (use US in ‹auto split: if-splits›)
 subgoal by (rule propa-cands-enqueued-U0-mono [
```

OF - propa-cands-enqueued-subsumed-mono[*OF - propa-cands-enqueued-mono*[*OF learned NN' propa-cands*]]])
(*use US* **in** ‹*auto split*: *if-splits*›)
subgoal by *simp*
subgoal by (*rule clss-to-upd*)
subgoal by (*rule past-invs*)
done
qed

lemma *in-multiset-subsetD*:
‹$A \in\# B \Longrightarrow B \subseteq\# C + D \Longrightarrow A \in\# C \vee A \in\# D$›
by *auto*

lemma *rtranclp-cdcl-twl-restart-twl-struct-invs*:
assumes
‹*cdcl-twl-restart*** *S T*› **and**
‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto simp*: *cdcl-twl-restart-twl-struct-invs*)

lemma *cdcl-twl-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-restart S T*› **and** ‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms*
by (*induction rule*: *cdcl-twl-restart.induct*)
(*auto 4 3 simp*: *twl-stgy-invs-def cdcl$_W$-restart-mset.cdcl$_W$-stgy-invariant-def*
cdcl$_W$-restart-mset.conflict-non-zero-unless-level-0-def
conflicting.simps cdcl$_W$-restart-mset.no-smaller-confl-def clauses-def trail.simps
dest: *in-multiset-subsetD*
dest!: *get-all-ann-decomposition-exists-prepend*
split: *if-splits*)

lemma *rtranclp-cdcl-twl-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-restart*** *S T*› **and**
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto simp*: *cdcl-twl-restart-twl-stgy-invs*)

lemma *cdcl-twl-restart-only-cdcl*: ‹*cdcl-twl-restart-only T U* $\Longrightarrow$
pcdcl-restart-only (*pstate$_W$-of T*) (*pstate$_W$-of U*)›
by (*auto 5 3 simp*: *cdcl-twl-restart-only.simps pcdcl-restart-only.simps*)

lemma *cdcl-twl-restart-only-twl-struct-invs*:
assumes
‹*cdcl-twl-restart-only S T*› **and**
‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
using *assms cdcl-twl-restart-only-cdcl*[*OF assms*(*1*)]
proof (*induction rule*: *cdcl-twl-restart-only.induct*)
case (*norestart-trail M N U NE UE NS US Q*)
note *invs* = *this*(*1*) **and** *st'* = *this*(*2*)
then show *?case*
by *simp*
next

case (*restart-trail K M′ M2 M N U NE UE NS US N0 U0 Q*)
note *decomp* = *this*(*1*) **and** *inv* = *this*(*2*) **and** *st′* = *this*(*3*)
let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
let *?T* = ‹(*M′*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
have
  twl-st-inv: ‹*twl-st-inv ?S*› **and**
  ‹*valid-enqueued ?S*› **and**
  struct-inv: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of ?S*)› **and**
  smaller: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of ?S*)› **and**
  excep: ‹*twl-st-exception-inv ?S*› **and**
  no-dup-q: ‹*no-duplicate-queued ?S*› **and**
  dist: ‹*distinct-queued ?S*› **and**
  confl: ‹*confl-cands-enqueued ?S*› **and**
  propa: ‹*propa-cands-enqueued ?S*› **and**
  ‹*get-conflict ?S* ≠ *None* ⟶
   clauses-to-update ?S = {#} ∧
   literals-to-update ?S = {#}› **and**
  to-upd: ‹*clauses-to-update-inv ?S*› **and**
  past: ‹*past-invs ?S*›
  using *inv* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def* **by** *clarsimp+*
have
  ex: ‹(∀ *C*∈#*N* + *U*. *twl-lazy-update M′ C* ∧
        watched-literals-false-of-max-level M′ C ∧
        twl-exception-inv (*M′*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}) *C*)› **and**
   conf-cands: ‹*confl-cands-enqueued* (*M′*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
   propa-cands: ‹*propa-cands-enqueued* (*M′*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
   clss-to-upd: ‹*clauses-to-update-inv* (*M′*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
  using *past get-all-ann-decomposition-exists-prepend*[*OF decomp*]
    confl propa to-upd excep
  unfolding *past-invs.simps*
  by *force+*

 have *excp-inv*: ‹*twl-st-exception-inv ?T*›
   using *ex* **unfolding** *twl-st-exception-inv.simps* **by** *blast+*
 have *twl-st-inv′*: ‹*twl-st-inv ?T*›
   using *ex twl-st-inv*
   unfolding *twl-st-exception-inv.simps twl-st-inv.simps*
   by *auto*
 have *n-d*: ‹*no-dup M*›
   using *struct-inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
     cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
   by (*auto simp*: *trail.simps*)
 obtain *M3* **where**
   M: ‹*M* = *M3* @ *M2* @ *Decided K* # *M′*›
   using *decomp* **by** *blast*
 define *M3′* **where** ‹*M3′* = *M3* @ *M2*›
 then have *M3′*: ‹*M* = *M3′* @ *Decided K* # *M′*›
   unfolding *M* **by** *auto*
 have *past-invs*: ‹*past-invs ?T*›
   unfolding *past-invs.simps*
 proof (*intro conjI impI allI*)
   fix *M1 M2 K′*
   assume *H*:‹*M′* = *M2* @ *Decided K′* # *M1*›
   let *?U* = ‹(*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
   let *?U′* = ‹(*M1*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
   have ‹*M* = (*M3′* @ *Decided K* # *M2*) @ *Decided K′* # *M1*›

```
      using H M3' by simp
    then show
      1: ‹∀ C∈#N + U.
          twl-lazy-update M1 C ∧
          watched-literals-false-of-max-level M1 C ∧
          twl-exception-inv ?U C› and
      2: ‹confl-cands-enqueued ?U› and
      3: ‹propa-cands-enqueued ?U› and
      4: ‹clauses-to-update-inv ?U›
      using past unfolding past-invs.simps by blast+
  qed
  have clss-to-upd: ‹clauses-to-update-inv ?T›
    using clss-to-upd by (auto simp add: filter-mset-empty-conv)

  have struct-inv': ‹pcdcl-all-struct-invs (pstate_W-of ?T)›
    using pcdcl-restart-only-pcdcl-all-struct-invs st' struct-inv by fastforce
  have smaller': ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of ?T)›
    using pcdcl-restart-only-no-smaller-propa smaller st' state_W-of-def struct-inv by force
  show ?case
    unfolding twl-struct-invs-def
    apply (intro conjI)
    subgoal
      using twl-st-inv' by (auto simp: ac-simps)
    subgoal by simp
    subgoal by (rule struct-inv')
    subgoal by (rule smaller')
    subgoal using excp-inv by auto
    subgoal using no-dup-q by auto
    subgoal using dist by auto
    subgoal using conf-cands by auto

    subgoal using propa-cands by auto
    subgoal by simp
    subgoal by (rule clss-to-upd)
    subgoal by (rule past-invs)
    done
qed

definition pcdcl-twl-final-state :: ‹'v twl-st ⇒ bool› where
  ‹pcdcl-twl-final-state S ⟷ no-step cdcl-twl-stgy S ∨
  (count-decided (get-trail S) = 0 ∧ get-conflict S ≠ None)›

end
theory Watched-Literals-Transition-System-Reduce
imports Watched-Literals-Transition-System-Restart
begin

inductive cdcl-twl-inp :: ‹'v twl-st ⇒ 'v twl-st ⇒ bool› where
  ‹cdcl-twl-subsumed S T ⟹ cdcl-twl-inp S T› |
  ‹cdcl-twl-subresolution S T ⟹ cdcl-twl-inp S T› |
  ‹cdcl-twl-unitres S T ⟹ cdcl-twl-inp S T› |
  ‹cdcl-twl-unitres-true S T ⟹ cdcl-twl-inp S T› |
  ‹cdcl-twl-restart S T ⟹ cdcl-twl-inp S T› |
  ‹cdcl-twl-pure-remove S T ⟹ cdcl-twl-inp S T›

lemma true-clss-clss-subset-entailedI:
```

‹$UE + UE' = UE'' + ca \Longrightarrow A \models ps\ set\text{-}mset\ UE \Longrightarrow A \models ps\ set\text{-}mset\ UE' \Longrightarrow A \models ps\ set\text{-}mset\ UE''$›
‹$UE + UE' = ca + UE'' \Longrightarrow A \models ps\ set\text{-}mset\ UE \Longrightarrow A \models ps\ set\text{-}mset\ UE' \Longrightarrow A \models ps\ set\text{-}mset\ UE''$›
for *UE* :: ‹*'v clauses*›
by (*metis set-mset-union true-clss-clss-union-and*)+

lemma *cdcl-twl-restart-entailed-init*:
‹*cdcl-twl-restart S T* ⟹
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (*state-of* (*pstate$_W$-of S*)) ⟹
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (*state-of* (*pstate$_W$-of T*))›
by (*induction rule*: *cdcl-twl-restart.induct*)
(*auto simp*: *cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init-def*
subset-mset.le-iff-add Un-left-commute image-Un sup-commute
intro: *true-clss-clss-subset-entailedI*)

lemma *cdcl-twl-inp-invs*:
assumes ‹*cdcl-twl-inp S T*›
‹*twl-struct-invs S*›
‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of S*)›
shows
cdcl-twl-inp-twl-struct-invs: ‹*twl-struct-invs T*› (**is** *?A*) **and**
cdcl-twl-inp-twl-stgy-invs: ‹*twl-stgy-invs S* ⟹ *twl-stgy-invs T*› (**is** ‹- ⟹ *?B*›) **and**
‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of T*)› (**is** *?C*)
proof −
show *?A*
using *assms*(*1*,*2*,*3*)
by (*induction rule*: *cdcl-twl-inp.induct*)
(*blast dest*: *cdcl-twl-subsumed-struct-invs cdcl-twl-subresolution-twl-struct-invs*
cdcl-twl-unitres-struct-invs cdcl-twl-unitres-true-twl-struct-invs
cdcl-twl-pure-remove-twl-struct-invs
cdcl-twl-restart-twl-struct-invs)+
show *?C*
using *assms*(*1*,*2*,*3*)
apply (*induction rule*: *cdcl-twl-inp.induct*)
apply (*metis cdcl-subsumed-RI-pcdcl cdcl-subsumed-entailed-by-init cdcl-twl-subsumed-cdcl-subsumed*
rtranclp-pcdcl-entailed-by-init state$_W$-of-def twl-struct-invs-def)
unfolding *state$_W$-of-def*
apply (*elim cdcl-twl-subresolution-decompE*)
apply (*auto elim*!: *cdcl-twl-subresolution-decompE*
simp: *twl-struct-invs-def struct-wf-twl-cls-alt-def twl-st-inv-alt-def*; *fail*)[]
apply (*rule cdcl-subresolutions-entailed-by-init*; *assumption*)
apply (*metis cdcl-flush-unit-unchanged cdcl-subresolution cdcl-subresolutions-entailed-by-init*
pcdcl.intros(*1*) *pcdcl-core.intros*(*2*) *pcdcl-entailed-by-init rtranclp-pcdcl-all-struct-invs*
twl-struct-invs-def)
apply (*drule cdcl-twl-unitres-cdcl-unitres*, *drule cdcl-unitres-learn-subsume*)
apply *assumption*+
using *rtranclp-pcdcl-entailed-by-init twl-struct-invs-def* **apply** *blast*
apply (*drule cdcl-twl-unitres-true-cdcl-unitres-true*, *drule pcdcl.intros*)
using *pcdcl-entailed-by-init twl-struct-invs-def* **apply** *blast*
apply (*solves* ‹*simp add*: *cdcl-twl-restart-entailed-init*›)
using *cdcl-pure-literal-remove-entailed-by-init cdcl-twl-pure-remove-cdcl-pure-remove*
twl-struct-invs-def **by** *blast*
with *assms* **show** *?B* **if** ‹*twl-stgy-invs S*›
using *that*
apply (*induction rule*: *cdcl-twl-inp.induct*)
apply (*metis* (*no-types*, *lifting*) *cdcl-twl-subsumed-stgy-invs*)
using *cdcl-twl-subresolution-twl-stgy-invs* **apply** *blast*

using *cdcl-twl-unitres-twl-stgy-invs* **apply** *blast*
apply (*metis* (*no-types*, *lifting*) *cdcl-twl-unitres-true-cdcl-unitres-true cdcl-unitres-true-same* $state_W$*-of-def twl-stgy-invs-def*)
using *cdcl-twl-restart-twl-stgy-invs* **apply** *blast*
using *cdcl-twl-pure-remove-twl-stgy-invs* **by** *blast*
qed

lemma *rtranclp-cdcl-twl-inp-invs*:
assumes ‹*cdcl-twl-inp*** *S T*›
‹*twl-struct-invs S*›
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)›
shows
rtranclp-cdcl-twl-inp-twl-struct-invs: ‹*twl-struct-invs T*› **and**
rtranclp-cdcl-twl-inp-twl-stgy-invs: ‹*twl-stgy-invs S* ⟹ *twl-stgy-invs T*› **and**
rtranclp-cdcl-twl-inp-entailed-init:
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*)›
using *assms*
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *T U* **using** *cdcl-twl-inp-invs*[*of T U*] **by** *auto*
subgoal for *T U* **using** *cdcl-twl-inp-invs*[*of T U*] **by** *auto*
subgoal for *T U* **using** *cdcl-twl-inp-invs*[*of T U*] **by** *auto*
done

lemma *cdcl-twl-inp-no-new-conflict*:
‹*cdcl-twl-inp S T* ⟹ *get-conflict T* = *get-conflict S* ∨ *get-conflict T* ≠ *None* ∧ *count-decided*(*get-trail T*) = *0*›
by (*induction rule*: *cdcl-twl-inp.induct*)
(*auto simp*: *cdcl-twl-subsumed.simps cdcl-twl-subresolution.simps cdcl-twl-unitres.simps cdcl-twl-unitres-true.simps cdcl-twl-restart.simps cdcl-twl-pure-remove.simps*)

lemma *rtranclp-pcdcl-restart-inprocessing*: ‹*pcdcl*** *S T* ⟹ *pcdcl-inprocessing*** *S T*›
by (*induction rule*: *rtranclp-induct*) (*auto dest*: *pcdcl-inprocessing.intros*)

lemma *cdcl-twl-inp-pcdcl*:
‹*cdcl-twl-inp S T* ⟹ *twl-struct-invs S* ⟹
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*) ⟹
*pcdcl-inprocessing*** (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)›
apply (*induction rule*: *cdcl-twl-inp.induct*)
subgoal
by (*meson cdcl-subsumed-RI-pcdcl cdcl-twl-subsumed-cdcl-subsumed pcdcl.intros*(*4*) *r-into-rtranclp rtranclp-pcdcl-restart-inprocessing*)
subgoal
apply (*rule rtranclp-pcdcl-restart-inprocessing*, *elim cdcl-twl-subresolution-decompE*)
apply (*auto elim*!: *cdcl-twl-subresolution-decompE*
simp: *twl-struct-invs-def struct-wf-twl-cls-alt-def twl-st-inv-alt-def*; *fail*)[]
apply (*drule cdcl-subresolution*)
apply (*auto elim*!: *cdcl-twl-subresolution-decompE*
simp: *twl-struct-invs-def struct-wf-twl-cls-alt-def twl-st-inv-alt-def*; *fail*)[]
by (*meson cdcl-subresolution pcdcl.intros*(*1*) *pcdcl.intros*(*5*) *pcdcl-core.intros*(*2*) *rtranclp.rtrancl-into-rtrancl*)
subgoal
by (*simp add*: *cdcl-twl-unitres-cdcl-unitres cdcl-unitres-learn-subsume rtranclp-pcdcl-restart-inprocessing*)
subgoal

by (*simp add*: *cdcl-twl-unitres-true-cdcl-unitres-true pcdcl.intros*(*8*) *r-into-rtranclp*
rtranclp-pcdcl-restart-inprocessing)
subgoal
using *cdcl-twl-restart-pcdcl pcdcl-inprocessing.intros*(*2*) **by** *blast*
subgoal
by (*simp add*: *cdcl-twl-pure-remove-cdcl-pure-remove pcdcl.intros*(*10*)
pcdcl-inprocessing.intros(*1*) *r-into-rtranclp*)
done

lemma *rtranclp-cdcl-twl-inp-pcdcl*:
‹*cdcl-twl-inp*** *S T* ⟹ *twl-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$ *-of S*) ⟹
*pcdcl-inprocessing*** (*pstate*$_W$ *-of S*) (*pstate*$_W$ *-of T*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-inp-pcdcl*[*of T U*] *rtranclp-cdcl-twl-inp-invs*[*of S T*]
by *simp*
done

context *twl-restart-ops*
begin

This is en essence the calculus with restarts we are now using. Compared to the version in my thesis, the major difference is that we don't restrict restarts anymore, by requiring only that at least one clause has been learned since.

However, this has a major drawback: The transition do not depend only on the current state, but also on the path that was taken. This is annoying for refinement, because the main loop does not do one transition anymore, but only a part of transitions. The difference is very small on the practical side, but that makes the termination more involved.

We allow inprocessing, but restrict it a lot. We could allow anything such that the invariants are still fulfilled afterwards, but we currently restrict to be some CDCL steps (TODO: generalise to also include restarts) and add requirements on the output.

There is a second termination condition that is not included here, namely UNSAT by inprocessing. We decided against including because it breaks the termination argument and makes expressing the relation between the elements of the state much more complicated without helping at all. At the end, we talk about conclusive states anyway.

type-synonym *'v twl-st-restart* = ‹*'v twl-st* × *'v twl-st* × *'v twl-st* × *nat* × *nat* × *bool*›
inductive *cdcl-twl-stgy-restart* :: ‹*'v twl-st-restart* ⇒ *'v twl-st-restart* ⇒ *bool*› **where**
step:
‹*cdcl-twl-stgy-restart* (*R*, *S*, *T*, *m*, *n*, *True*) (*R*, *S*, *U*, *m*, *n*, *True*)›
if
‹*cdcl-twl-stgy T U*› |
restart-step:
‹*cdcl-twl-stgy-restart* (*R*, *S*, *T*, *m*, *n*, *True*) (*W*, *W*, *W*, *m*, *Suc n*, *True*)›
if
‹*size* (*get-all-learned-clss T*) − *size* (*get-all-learned-clss R*) > *f n*› **and**
‹*cdcl-twl-inp*** *T U*› **and**
‹*cdcl-twl-stgy*** *U W*›
‹*clauses-to-update W* = {#}›
‹*get-conflict W* ≠ *None* ⟶ *count-decided* (*get-trail W*) = *0*› |

restart-noGC:
‹*cdcl-twl-stgy-restart* (*R*, *S*, *T*, *m*, *n*, *True*) (*R*, *U*, *U*, *Suc m*, *n*, *True*)›
if
‹*size* (*get-all-learned-clss T*) > *size* (*get-all-learned-clss S*)› **and**
‹*cdcl-twl-restart-only T U*› |
restart-full:
‹*cdcl-twl-stgy-restart* (*R*, *S*, *T*, *m*, *n*, *True*) (*R*, *S*, *T*, *m*, *n*, *False*)›
if
‹*pcdcl-twl-final-state T*›

lemma *cdcl-twl-stgy-restart-induct*[*consumes 1*, *case-names restart-step restart-noGC full*]:
assumes
‹*cdcl-twl-stgy-restart* (*R*, *S*, *T*, *m*, *n*, *b*) (*R'*, *S'*, *T'*, *m'*, *n'*, *b'*)› **and**
‹⋀*R S T U*. *cdcl-twl-stgy T U* ⟹ *m'* = *m* ⟹ *n'* = *n* ⟹ *b* ⟹ *b'* ⟹ *P R S T m n True R S U m n True*› **and**
‹⋀*R S T U V W*.
f n < *size* (*get-all-learned-clss T*) − *size* (*get-all-learned-clss R*) ⟹ *cdcl-twl-inp*** *T U* ⟹ *cdcl-twl-stgy*** *U W* ⟹
clauses-to-update W = {#} ⟹ (*get-conflict W* ≠ *None* ⟹ *count-decided* (*get-trail W*) = *0*) ⟹
m' = *m* ⟹ *n'* = *Suc n* ⟹
P R S T m n True W W W m (*Suc n*) *True*›**and**
‹⋀*R S T U*.
size (*get-all-learned-clss T*) > *size* (*get-all-learned-clss S*) ⟹
cdcl-twl-restart-only T U ⟹ *m'* = *Suc m* ⟹ *n'* = *n* ⟹
P R S T m n True R U U (*Suc m*) *n True*›
‹*pcdcl-twl-final-state T* ⟹ *R'* = *R* ⟹ *S'* = *S* ⟹ *T'* = *T* ⟹ *P R S T m n True R S T m n False*›
shows ‹*P R S T m n b R' S' T' m' n' b'*›
using *assms*(*1*)
apply (*cases rule*: *cdcl-twl-stgy-restart.cases*)
subgoal
using *assms*(*2*)[*of T T' R S*]
by *simp*
subgoal for *U*
using *assms*(*3*)[*of*]
by *simp*
subgoal
using *assms*(*4*)[*of*]
by *simp*
subgoal
using *assms*(*5*)[*of*]
by *simp*
done

lemma *tranclp-cdcl-twl-stgy-cdcl$_W$-stgy2*:
‹*cdcl-twl-stgy*$^{++}$ *S T* ⟹
twl-struct-invs S ⟹ (*pstate$_W$-of S*) ≠ (*pstate$_W$-of T*) ⟹
pcdcl-tcore-stgy^{++} (*pstate$_W$-of S*) (*pstate$_W$-of T*)›
using *rtranclp-cdcl-twl-stgy-cdcl$_W$-stgy*[*of S T*] **unfolding** *rtranclp-unfold*
by *auto*

lemma *tranclp-cdcl-twl-stgy-cdcl$_W$-stgy3*:
‹*cdcl-twl-stgy*$^{++}$ *S T* ⟹
size (*get-all-learned-clss T*) > *size* (*get-all-learned-clss S*) ⟹

twl-struct-invs S ⟹
pcdcl-tcore-stgy^{++} (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)›
using *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*[*of S T*] **unfolding** *rtranclp-unfold*
apply *auto*
apply (*cases T*; *cases S*)
apply (*auto simp*: *clauses-def dest*!: *arg-cong*[*of* ‹*clauses* -› - *size*])
done

lemma [*twl-st*, *simp*]:
‹*pget-all-learned-clss* (*pstate*$_W$*-of T*) = *get-all-learned-clss T*›
‹*pget-conflict* (*pstate*$_W$*-of T*) = *get-conflict T*›
by (*cases T*; *auto*; *fail*)+

lemma *pcdcl-twl-final-state-pcdcl*:
‹*pcdcl-twl-final-state S* ⟹
twl-struct-invs S ⟹ *pcdcl-final-state* (*pstate*$_W$*-of S*)›
using *no-step-cdcl-twl-stgy-no-step-cdcl*$_W$*-stgy*[*of S*]
unfolding *pcdcl-twl-final-state-def pcdcl-final-state-def*
by (*auto simp add*: *no-step-pcdcl-core-stgy-pcdcl-coreD*)

lemma *pcdcl-stgy-restart-stepI*:
‹*pcdcl-tcore-stgy*** *T T'* ⟹ *pcdcl-stgy-restart*** (*R*, *S*, *T*, *m*, *n*, *True*) (*R*, *S*, *T'*, *m*, *n*, *True*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *V W*
by (*auto dest*!: *pcdcl-stgy-restart.intros*(*1*)[*of* - - *R S m n*])
done

lemma *rtranclp-cdcl-twl-inp-pcdcl-inprocessing*:
‹*cdcl-twl-inp*** *S T* ⟹ *twl-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*) ⟹
pcdcl-inprocessing** (*pstate*$_W$*-of S*) (*pstate*$_W$*-of T*)›
by (*drule rtranclp-cdcl-twl-inp-pcdcl*; *assumption?*)

lemma *cdcl-twl-stgy-restart-pcdcl*:
‹*cdcl-twl-stgy-restart* (*R*, *S* :: 'v *twl-st*, *T*, *m*, *n*, *g*) (*R'*, *S'*, *T'*, *m'*, *n'*, *h*) ⟹
twl-struct-invs R ⟹ *twl-struct-invs S* ⟹ *twl-struct-invs T* ⟹ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-i
(*state*$_W$*-of T*) ⟹
pcdcl-stgy-restart** (*pstate*$_W$*-of R*, *pstate*$_W$*-of S*, *pstate*$_W$*-of T*, *m*, *n*, *g*)
(*pstate*$_W$*-of R'*, *pstate*$_W$*-of S'*, *pstate*$_W$*-of T'*, *m'*, *n'*, *h*)›
apply (*induction rule*: *cdcl-twl-stgy-restart-induct*)
subgoal for *R S T U*
by (*drule cdcl-twl-stgy-cdcl*$_W$*-stgy*)
(*simp add*: *pcdcl-stgy-restart-stepI*)+
subgoal
apply (*rule r-into-rtranclp*)
apply (*rule pcdcl-stgy-restart.intros*(*2*))
apply *simp*
apply (*rule rtranclp-cdcl-twl-inp-pcdcl-inprocessing*; *assumption*)
using *cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy rtranclp-pcdcl-tcore-stgy-pcdcl-stgy'*
apply (*simp-all add*: *cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*
rtranclp-pcdcl-tcore-stgy-pcdcl-stgy' rtranclp-cdcl-twl-inp-twl-struct-invs)
apply (*smt cdcl-twl-restart-pcdcl pcdcl-restart-no-smaller-propa' rtranclp-cdcl-twl-inp-twl-struct-invs*
state$_W$*-of-def twl-struct-invs-def*)
done

subgoal
apply (*rule r-into-rtranclp*)
apply (*rule pcdcl-stgy-restart.intros*(*3*))
apply *simp*
apply (*rule cdcl-twl-restart-only-cdcl*, *assumption*)
done
subgoal
apply (*rule r-into-rtranclp*)
apply (*rule pcdcl-stgy-restart.intros*(*4*))
by (*simp add*: *twl-restart-ops.pcdcl-twl-final-state-pcdcl*)
done

lemma *cdcl-twl-stgy-restart-twl-struct-invs*:
assumes
‹*cdcl-twl-stgy-restart S T*› **and**
‹*twl-struct-invs* (*fst S*)› **and**
‹*twl-struct-invs* (*fst* (*snd S*))› **and**
‹*twl-struct-invs* (*fst* (*snd* (*snd S*)))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* (*fst* (*snd* (*snd S*))))›
shows ‹*twl-struct-invs* (*fst T*) ∧ *twl-struct-invs* (*fst* (*snd T*)) ∧ *twl-struct-invs* (*fst* (*snd* (*snd T*)))›
using *assms*
by (*induction rule*: *cdcl-twl-stgy-restart.induct*)
(*auto simp add*: *full1-def intro*: *rtranclp-cdcl-twl-stgy-twl-struct-invs*
dest: *cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-twl-stgy-invs*
rtranclp-cdcl-twl-stgy-twl-struct-invs
cdcl-twl-restart-only-twl-struct-invs
simp:
cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-inp-twl-struct-invs
rtranclp-cdcl-twl-stgy-twl-struct-invs rtranclp-cdcl-twl-inp-twl-struct-invs
dest!: *tranclp-into-rtranclp intro*: *rtranclp-cdcl-twl-inp-twl-struct-invs*)

lemma *pcdcl-restart-entailed-by-init*:
assumes ‹*pcdcl-restart S T*› **and**
‹*pcdcl-all-struct-invs S*› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of S*)›
shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of T*)›
using *assms*
apply (*induction rule*: *pcdcl-restart.induct*)
subgoal
by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def*
subset-mset.le-iff-add ac-simps intro: *true-clss-clss-subset-entailedI*)
subgoal
by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def*
subset-mset.le-iff-add ac-simps intro: *true-clss-clss-subset-entailedI*)
done

lemma *pcdcl-restart-only-entailed-by-init*:
assumes ‹*pcdcl-restart-only S T*› **and**
‹*pcdcl-all-struct-invs S*› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of S*)›
shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of T*)›
using *assms*
apply (*induction rule*: *pcdcl-restart-only.induct*)
subgoal
by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def*

subset-mset.le-iff-add ac-simps)
done

lemma *cdcl-twl-stgy-restart-entailed-by-init*:
assumes
‹*cdcl-twl-stgy-restart S T*› **and**
‹*twl-struct-invs* (*fst S*)› **and**
‹*twl-struct-invs* (*fst* (*snd S*))› **and**
‹*twl-struct-invs* (*fst* (*snd* (*snd S*)))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst S*))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd S*)))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd* (*snd S*))))›
shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst T*)) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd T*))) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd* (*snd T*))))›
using *assms*
apply (*induction rule*: *cdcl-twl-stgy-restart.induct*)
subgoal apply *simp*
using *cdcl-twl-stgy-*$cdcl_W$*-stgy rtranclp-pcdcl-entailed-by-init rtranclp-pcdcl-stgy-pcdcl*
rtranclp-pcdcl-tcore-stgy-pcdcl-stgy′ twl-struct-invs-def **by** *blast*
subgoal apply *simp*
by (*metis* (*mono-tags*, *lifting*) *cdcl-twl-inp.intros*(*5*) *rtranclp.rtrancl-into-rtrancl*
rtranclp-cdcl-twl-stgy-$cdcl_W$*-stgy rtranclp-cdcl-twl-inp-entailed-init*
rtranclp-cdcl-twl-inp-twl-struct-invs rtranclp-pcdcl-entailed-by-init
rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-tcore-stgy-pcdcl-stgy′ $state_W$*-of-def*
twl-struct-invs-def)
subgoal apply *simp*
using *cdcl-twl-restart-only-cdcl twl-restart-ops.pcdcl-restart-only-entailed-by-init*
twl-struct-invs-def **by** *blast*
subgoal
by *simp*
done

lemma *rtranclp-cdcl-twl-stgy-restart-twl-struct-invs*:
assumes
‹*cdcl-twl-stgy-restart*** *S T*› **and**
‹*twl-struct-invs* (*fst S*)› **and**
‹*twl-struct-invs* (*fst* (*snd S*))› **and**
‹*twl-struct-invs* (*fst* (*snd* (*snd S*)))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst S*))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd S*)))› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd* (*snd S*))))›
shows ‹*twl-struct-invs* (*fst T*) ∧ *twl-struct-invs* (*fst* (*snd T*)) ∧ *twl-struct-invs* (*fst* (*snd* (*snd T*))) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst T*)) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd T*))) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* (*fst* (*snd* (*snd T*))))›
using *assms*
apply (*induction*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-stgy-restart-entailed-by-init*[*of T U*] *cdcl-twl-stgy-restart-twl-struct-invs*[*of T U*]
by *simp*
done

lemma *rtranclp-cdcl-twl-stgy-restart-pcdcl*:

‹*cdcl-twl-stgy-restart*** (*R*, *S* :: *'v twl-st*, *T*, *m*, *n*, *g*) (*R'*, *S'*, *T'*, *m'*, *n'*, *h*) ⟹
twl-struct-invs R ⟹ *twl-struct-invs S* ⟹ *twl-struct-invs T* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of R*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*) ⟹
*pcdcl-stgy-restart*** (*pstate*$_W$*-of R*, *pstate*$_W$*-of S*, *pstate*$_W$*-of T*, *m*, *n*, *g*)
(*pstate*$_W$*-of R'*, *pstate*$_W$*-of S'*, *pstate*$_W$*-of T'*, *m'*, *n'*, *h*)›
apply (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -, -, -, -)› ‹(-, -, -, -, -, -)›, *split-format*(*complete*), *of* **for** *r*])
subgoal by *auto*
subgoal for *R' S' T' m' n' g' R'' S'' T'' m'' n'' g''*
using *rtranclp-cdcl-twl-stgy-restart-twl-struct-invs*[*of* ‹(*R*, *S*, *T*, *m*, *n*, *g*)› ‹(*R'*, *S'*, *T'*, *m'*, *n'*, *g'*)›]
by (*auto dest*: *cdcl-twl-stgy-restart-pcdcl*)
done

lemma *cdcl-twl-stgy-cdcl*$_W$*-stgy-restart2*:
assumes ‹*cdcl-twl-stgy-restart* (*S*, *T*, *U*, *m*, *n*, *g*) (*S'*, *T'*, *U'*, *m'*, *n'*, *g'*)›
and *twl*: ‹*twl-struct-invs U*› **and** ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of U*)›
shows ‹*pcdcl-stgy-restart* (*pstate*$_W$*-of S*, *pstate*$_W$*-of T*, *pstate*$_W$*-of U*, *m*, *n*, *g*)
(*pstate*$_W$*-of S'*, *pstate*$_W$*-of T'*, *pstate*$_W$*-of U'*, *m'*, *n'*, *g'*) ∨
(*S* = *S'* ∧ *T* = *T'* ∧ *m* = *m'* ∧ *n* = *n'* ∧ *g* = *g'* ∧
pstate$_W$*-of U* = *pstate*$_W$*-of U'* ∧ (*literals-to-update-measure U'*, *literals-to-update-measure U*)
∈ *lexn less-than 2*)›
using *assms*(*1*,*2*,*3*)
apply (*induction rule*: *cdcl-twl-stgy-restart-induct*)
subgoal for *R S T U*
by (*drule cdcl-twl-stgy-cdcl*$_W$*-stgy2*)
(*auto simp add*: *pcdcl-stgy-restart.step*)
subgoal
apply (*rule disjI1*)
apply (*rule pcdcl-stgy-restart.intros*(*2*))
apply *simp*
apply (*rule rtranclp-cdcl-twl-inp-pcdcl-inprocessing*)
apply *assumption+*
using *cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy rtranclp-pcdcl-tcore-stgy-pcdcl-stgy'*
apply (*simp-all add*: *cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy rtranclp-pcdcl-tcore-stgy-pcdcl-stgy' rtranclp-cdcl-twl-inp-twl-struct-invs*)
by (*smt cdcl-twl-restart-pcdcl pcdcl-restart-no-smaller-propa' rtranclp-cdcl-twl-inp-twl-struct-invs state*$_W$*-of-def twl-struct-invs-def*)
subgoal
apply (*rule disjI1*)
apply (*rule pcdcl-stgy-restart.intros*(*3*))
apply *simp*
apply (*rule cdcl-twl-restart-only-cdcl*, *assumption*)
done
subgoal
apply (*rule disjI1*)
apply (*rule pcdcl-stgy-restart.intros*(*4*))
by (*simp add*: *twl-restart-ops.pcdcl-twl-final-state-pcdcl*)
done

abbreviation *state*$_W$*-of-restart* **where**
‹*state*$_W$*-of-restart* ≡ (λ(*S*, *T*, *U*, *n*, *b*). (*state*$_W$*-of S*, *state*$_W$*-of T*, *state*$_W$*-of U*, *n*))›

definition *twl-restart-inv* :: ‹*'v twl-st-restart* ⇒ *bool*› **where**

‹*twl-restart-inv* = (λ(*Q*, *R*, *S*, *m*, *n*). *twl-struct-invs Q* ∧ *twl-struct-invs R* ∧ *twl-struct-invs S* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of Q*) ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of R*) ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*) ∧
pcdcl-stgy-restart-inv (*pstate*$_W$*-of Q*, *pstate*$_W$*-of R*, *pstate*$_W$*-of S*, *m*, *n*))›

lemma *cdcl-twl-stgy-entailed-by-init*:
‹*cdcl-twl-stgy S T* ⟹ *twl-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*)›
apply (*induction rule*: *cdcl-twl-stgy.induct*)
apply (*metis cdcl-twl-stgy-cdcl*$_W$*-stgy cp rtranclp-pcdcl-entailed-by-init rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-tcore-stgy-pcdcl-stgy*′ *state*$_W$*-of-def twl-struct-invs-def*)
by (*metis cdcl-twl-o-cdcl*$_W$*-o-stgy pcdcl-tcore-pcdcl pcdcl-tcore-stgy-pcdcl-tcoreD rtranclp-pcdcl-entailed-by-init state*$_W$*-of-def twl-struct-invs-def*)

lemma *rtranclp-cdcl-twl-stgy-entailed-by-init*:
‹*cdcl-twl-stgy*** *S T* ⟹ *twl-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*)›
apply (*induction rule*: *rtranclp-induct*)
apply *auto*[]
using *rtranclp-cdcl-twl-stgy-twl-struct-invs twl-restart-ops.cdcl-twl-stgy-entailed-by-init* **by** *blast*

lemma *cdcl-twl-stgy-restart-twl-restart-inv*:
‹*cdcl-twl-stgy-restart S T* ⟹ *twl-restart-inv S* ⟹ *twl-restart-inv T*›
apply (*induction rule*: *cdcl-twl-stgy-restart.induct*)
subgoal for *T U R S m n*
using *cdcl-twl-stgy-cdcl*$_W$*-stgy-restart2*[*of R S T m n True R S U m n True*]
unfolding *twl-restart-inv-def*
apply (*auto intro*: *cdcl-twl-stgy-twl-struct-invs*
simp: *pcdcl-stgy-restart-pcdcl-stgy-restart-inv cdcl-twl-stgy-restart.intros*)
using *cdcl-twl-stgy-cdcl*$_W$*-stgy rtranclp-pcdcl-entailed-by-init rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-tcore-stgy-pcdcl-stgy*′ **unfolding** *twl-struct-invs-def* **by** *blast*
subgoal for *T R n U W S m*
using *cdcl-twl-stgy-cdcl*$_W$*-stgy-restart2*[*of R S T m n True W W W m* ‹*Suc n*› *True*]
unfolding *twl-restart-inv-def*
apply (*auto intro*: *cdcl-twl-stgy-twl-struct-invs*
simp: *pcdcl-stgy-restart-pcdcl-stgy-restart-inv cdcl-twl-stgy-restart.intros*
cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-twl-struct-invs
rtranclp-cdcl-twl-inp-twl-struct-invs rtranclp-pcdcl-all-struct-invs
rtranclp-pcdcl-entailed-by-init pcdcl-restart-entailed-by-init
intro: *pcdcl-restart-entailed-by-init rtranclp-pcdcl-entailed-by-init*
dest: *rtranclp-cdcl-twl-inp-pcdcl cdcl-twl-restart-pcdcl*)
apply (*metis cdcl-twl-restart-twl-struct-invs cdcl-twl-inp.intros*(*5*)
cdcl-twl-inp-invs(*3*) *rtranclp-cdcl-twl-inp-entailed-init*
rtranclp-cdcl-twl-inp-twl-struct-invs state$_W$*-of-def*
rtranclp-cdcl-twl-stgy-entailed-by-init)
by (*metis cdcl-twl-restart-twl-struct-invs cdcl-twl-inp.intros*(*5*)
cdcl-twl-inp-invs(*3*) *rtranclp-cdcl-twl-inp-entailed-init*
rtranclp-cdcl-twl-inp-twl-struct-invs state$_W$*-of-def*
rtranclp-cdcl-twl-stgy-entailed-by-init)
subgoal for *T S U R m n*
using *cdcl-twl-stgy-cdcl*$_W$*-stgy-restart2*[*of R S T m n True R U U* ‹*Suc m*› *n True*]
unfolding *twl-restart-inv-def*
apply (*auto intro*: *cdcl-twl-stgy-twl-struct-invs*

simp: *pcdcl-stgy-restart-pcdcl-stgy-restart-inv cdcl-twl-stgy-restart.intros*
cdcl-twl-restart-only-twl-struct-invs)
using *cdcl-twl-restart-only-cdcl twl-restart-ops.pcdcl-restart-only-entailed-by-init twl-struct-invs-def*
by *blast*
subgoal
unfolding *twl-restart-inv-def pcdcl-stgy-restart-inv-def prod.simps* **by** *blast*
done

lemma *rtranclp-cdcl-twl-stgy-restart-twl-restart-inv*:
‹*cdcl-twl-stgy-restart*** *S T* ⟹ *twl-restart-inv S* ⟹ *twl-restart-inv T*›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *cdcl-twl-stgy-restart-twl-restart-inv*)

definition *twl-stgy-restart-inv* :: ‹'*v twl-st-restart* ⇒ *bool*› **where**
‹*twl-stgy-restart-inv* = (λ(*Q*, *R*, *S*, *m*, *n*). *twl-stgy-invs Q* ∧ *twl-stgy-invs R* ∧ *twl-stgy-invs S*)›
lemma *cdcl-twl-restart-only-twl-stgy-invs*:
‹*cdcl-twl-restart-only S T* ⟹ *twl-stgy-invs S* ⟹ *twl-stgy-invs T*›
by (*auto simp*: *cdcl-twl-restart-only.simps twl-stgy-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def cdcl*$_W$*-restart-mset.no-smaller-confl-def*
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
dest!: *get-all-ann-decomposition-exists-prepend*)

lemma *cdcl-twl-stgy-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-stgy-restart S T*› **and**
‹*twl-restart-inv S*› **and**
‹*twl-stgy-invs* (*fst S*)› **and**
‹*twl-stgy-invs* (*fst* (*snd S*))› **and**
‹*twl-stgy-invs* (*fst* (*snd* (*snd S*)))›**and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* (*fst S*))› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* (*fst* (*snd S*)))› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* (*fst* (*snd* (*snd S*))))›
shows ‹*twl-stgy-invs* (*fst T*) ∧ *twl-stgy-invs* (*fst* (*snd T*)) ∧ *twl-stgy-invs* (*fst* (*snd* (*snd T*)))›
using *assms*
apply (*induction rule*: *cdcl-twl-stgy-restart.induct*)
subgoal for *T U R S m n*
using *rtranclp-cdcl-twl-stgy-twl-stgy-invs*[*of T U*]
by (*auto simp*: *twl-restart-inv-def*)
subgoal for *T R n U W S m*
using *cdcl-twl-restart-twl-stgy-invs*[*of U V*]
by (*auto simp*: *twl-restart-inv-def rtranclp-cdcl-twl-inp-twl-stgy-invs*
cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-twl-stgy-invs
rtranclp-cdcl-twl-inp-twl-struct-invs
intro: *cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-twl-stgy-invs*)
subgoal for *T S U R m n*
using *cdcl-twl-restart-only-twl-stgy-invs*[*of T U*]
by (*auto simp*: *twl-restart-inv-def*)
subgoal
by *auto*
done

lemma *rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-stgy-restart*** *S T*› **and**
‹*twl-restart-inv S*› **and**
‹*twl-stgy-invs* (*fst S*)› **and**

⟨*twl-stgy-invs* (*fst* (*snd S*))⟩ **and**
⟨*twl-stgy-invs* (*fst* (*snd* (*snd S*)))⟩
shows ⟨*twl-stgy-invs* (*fst T*) ∧ *twl-stgy-invs* (*fst* (*snd T*)) ∧ *twl-stgy-invs* (*fst* (*snd* (*snd T*)))⟩
using *assms*
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-stgy-restart-twl-stgy-invs*[*of T U*]
rtranclp-cdcl-twl-stgy-restart-twl-restart-inv[*of S T*]
by (*auto dest*!: *simp*: *twl-restart-inv-def*)
done

lemma *cdcl-twl-stgy-restart-twl-stgy-restart-inv*:
⟨*cdcl-twl-stgy-restart S T* ⟹ *twl-restart-inv S* ⟹ *twl-stgy-restart-inv S* ⟹
twl-stgy-restart-inv T⟩
using *cdcl-twl-stgy-restart-twl-stgy-invs*[*of S T*] **unfolding** *twl-stgy-restart-inv-def twl-restart-inv-def*
by *auto*

lemma *rtranclp-cdcl-twl-stgy-restart-twl-stgy-restart-inv*:
⟨*cdcl-twl-stgy-restart*** *S T* ⟹ *twl-restart-inv S* ⟹ *twl-stgy-restart-inv S* ⟹
twl-stgy-restart-inv T⟩
using *rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs*[*of S T*] **unfolding** *twl-stgy-restart-inv-def*
by *auto*

lemma *cdcl$_W$-ex-cdcl$_W$-stgy*:
⟨*cdcl$_W$-restart-mset.cdcl$_W$ S T* ⟹ ∃ *U*. *cdcl$_W$-restart-mset.cdcl$_W$-stgy S U*⟩
by (*meson cdcl$_W$-restart-mset.cdcl$_W$.cases cdcl$_W$-restart-mset.cdcl$_W$-stgy.simps*)

lemma *rtranclp-cdcl$_W$-cdcl$_W$-init-state*:
⟨*cdcl$_W$-restart-mset.cdcl$_W$*** (*init-state* {#}) *S* ⟷ *S* = *init-state* {#}⟩
unfolding *rtranclp-unfold*
by (*subst tranclp-unfold-begin*)
(*auto simp*: *cdcl$_W$-stgy-cdcl$_W$-init-state-empty-no-step*
cdcl$_W$-stgy-cdcl$_W$-init-state
simp del: *init-state.simps*
dest: *cdcl$_W$-restart-mset.cdcl$_W$-stgy-cdcl$_W$ cdcl$_W$-ex-cdcl$_W$-stgy*)

lemma *rtranclp-pcdcl-core-is-cdcl*:
⟨*pcdcl-core*** *S T* ⟹ *cdcl$_W$-restart-mset.cdcl$_W$*** (*state-of S*) (*state-of T*)⟩
by (*induction rule*: *rtranclp.induct*)
(*auto dest*: *pcdcl-core-is-cdcl*)

lemma *pcdcl-tcore-is-cdclD*:
⟨*pcdcl-tcore T T′* ⟹
*cdcl$_W$-restart-mset.cdcl$_W$-restart*** (*state-of T*) (*state-of T′*)⟩
by (*induction rule*: *pcdcl-tcore.induct*)
(*auto intro*: *pcdcl-restart.intros dest*!: *pcdcl-core-is-cdcl*
cdcl$_W$-restart-mset.cdcl$_W$-cdcl$_W$-restart pcdcl-tcore-stgy-pcdcl-tcoreD
state-of-cdcl-subsumed cdcl-flush-unit-unchanged
cdcl-backtrack-unit-is-CDCL-backtrack)

lemma *rtranclp-pcdcl-entailed-by-init*:
assumes ⟨*pcdcl*** *S T*⟩ **and**
⟨*pcdcl-all-struct-invs S*⟩ **and**
⟨*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state-of S*)⟩
shows ⟨*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state-of T*)⟩

using *assms*
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *pcdcl-entailed-by-init*
intro: *rtranclp-pcdcl-all-struct-invs*)

lemma *pcdcl-inprocessing-entailed-by-init*:
‹*pcdcl-inprocessing S T* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of T*)›
apply (*induction rule*: *pcdcl-inprocessing.induct*)
subgoal for *S T*
using *pcdcl-entailed-by-init*[*of S T*] **by** *auto*
subgoal
using *pcdcl-restart-entailed-by-init* **by** *blast*
done

lemma *rtranclp-pcdcl-inprocessing-entailed-by-init*:
‹*pcdcl-inprocessing*** *S T* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of T*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal
by *auto*
subgoal for *T U*
using *pcdcl-inprocessing-entailed-by-init*[*of T U*] *rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs*[*of S T*]
by *blast*
done

lemma *pcdcl-stgy-restart-entailed-by-init*:
fixes *g g*′
assumes ‹*pcdcl-stgy-restart* (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1*′, *R2*′, *T*, *m*′, *n*′, *g*′)› **and**
‹*pcdcl-all-struct-invs S*› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*)›
shows ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of T*)›
using *assms*
apply (*induction* ‹(*R1*, *R2*, *S*, *m*, *n*, *g*)› ‹(*R1*′, *R2*′, *T*, *m*′, *n*′, *g*′)› *rule*: *pcdcl-stgy-restart.induct*)
subgoal
using *pcdcl-tcore-stgy-pcdcl-stgy*′ *rtranclp-pcdcl-entailed-by-init rtranclp-pcdcl-stgy-pcdcl*
by *blast*
subgoal for *U*
using *pcdcl-restart-entailed-by-init*[*of U R1*′] *pcdcl-restart-pcdcl-all-struct-invs*[*of S U*]
rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs[*of S U*]
rtranclp-pcdcl-inprocessing-entailed-by-init[*of S U*]
by (*auto dest*!: *pcdcl-tcore-stgy-pcdcl-stgy*′ *rtranclp-pcdcl-entailed-by-init*
rtranclp-pcdcl-stgy-pcdcl
dest: *pcdcl-restart-pcdcl-all-struct-invs*)
subgoal
using *pcdcl-restart-only-entailed-by-init*[*of S U*]
by (*auto dest*!: *pcdcl-tcore-stgy-pcdcl-stgy*′ *rtranclp-pcdcl-entailed-by-init*
rtranclp-pcdcl-stgy-pcdcl
dest: *pcdcl-restart-only-entailed-by-init*)
subgoal
by *auto*
done

lemma *pcdcl-stgy-restart-pcdcl-all-struct-invs*:
assumes ‹*pcdcl-stgy-restart* (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*)› **and**
‹*pcdcl-all-struct-invs S*›
shows ‹*pcdcl-all-struct-invs T*›
using *assms*
apply (*induction* ‹(*R1*, *R2*, *S*, *m*, *n*, *g*)› ‹(*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*)› *rule*: *pcdcl-stgy-restart.induct*)
apply (*simp-all add*: *pcdcl-tcore-stgy-all-struct-invs pcdcl-restart-pcdcl-all-struct-invs*
rtranclp-pcdcl-all-struct-invs rtranclp-pcdcl-stgy-pcdcl)
using *pcdcl-restart-pcdcl-all-struct-invs rtranclp-pcdcl-all-struct-invs rtranclp-pcdcl-stgy-pcdcl*
rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs **apply** *blast*
using *pcdcl-restart-only-pcdcl-all-struct-invs* **by** *blast*

lemma *rtranclp-pcdcl-stgy-restart-pcdcl-all-struct-invs*:
assumes ‹*pcdcl-stgy-restart*** (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*)› **and**
‹*pcdcl-all-struct-invs S*›
shows ‹*pcdcl-all-struct-invs T*›
using *assms*
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -, -, -, -)› ‹(-, -, -, -, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*!: *pcdcl-stgy-restart-pcdcl-all-struct-invs*)

lemma *rtranclp-pcdcl-stgy-restart-entailed-by-init*:
assumes ‹*pcdcl-stgy-restart*** (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*)› **and**
‹*pcdcl-all-struct-invs S*› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of S*)›
shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of T*)›
using *assms*
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -, -, -, -)› ‹(-, -, -, -, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*!: *pcdcl-stgy-restart-entailed-by-init rtranclp-pcdcl-stgy-restart-pcdcl-all-struct-invs*)

lemma *pcdcl-core-entailed-iff*:
‹*pcdcl-core S T* $\Longrightarrow$ *M* $\models m$ *pget-all-init-clss T* $\longleftrightarrow$ *M* $\models m$ *pget-all-init-clss S*›
by (*induction rule*: *pcdcl-core.induct*)
(*auto simp*: *cdcl-conflict.simps cdcl-propagate.simps cdcl-skip.simps*
cdcl-decide.simps cdcl-resolve.simps cdcl-backtrack.simps)

lemma *pcdcl-entailed-iff*:
‹*pcdcl S T* $\Longrightarrow$ *consistent-interp M* $\Longrightarrow$
total-over-set M (*atms-of-mm* (*pget-all-init-clss T*)) $\Longrightarrow$
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of S*) $\Longrightarrow$
M $\models m$ *pget-all-init-clss T* $\Longrightarrow$ *M* $\models m$ *pget-all-init-clss S*›
apply (*induction rule*: *pcdcl.induct*)
subgoal
by (*auto simp*: *pcdcl-core-entailed-iff*)
subgoal
by (*auto simp*: *cdcl-learn-clause.simps true-clss-cls-def total-over-m-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def*
dest: *spec*[*of* - *M*])
subgoal
by (*auto simp*: *cdcl-resolution.simps total-over-m-def consistent-interp-def*)
subgoal
by (*auto simp*: *cdcl-subsumed.simps*)
subgoal
by (*auto simp*: *cdcl-flush-unit.simps*)
subgoal

```
    by (auto simp: cdcl-inp-propagate.simps)
  subgoal
    by (auto simp: cdcl-inp-conflict.simps)
  subgoal
    by (auto simp: cdcl-unitres-true.simps)
  subgoal
    by (auto simp: cdcl-promote-false.simps)
  subgoal
    by (auto simp: cdcl-pure-literal-remove.simps)
  done

lemma cdcl-pure-literal-remove-satisfiable-iff:
  assumes
    ‹cdcl-pure-literal-remove S T› and
    ‹pcdcl-all-struct-invs S› and
    ent-init: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state-of S)›
  shows
    ‹satisfiable (set-mset (pget-all-init-clss S)) ⟷ satisfiable (set-mset (pget-all-init-clss T))›
  using assms(1)
proof (cases)
  case (cdcl-pure-literal-remove L N NE NS N0 M U UE US U0) note S = this(1) and T = this(2)
and
    L = this(3,4) and undef = this(5) and lev0 = this(6)
  have
    ent: ‹entailed-clss-inv S› and
    sub: ‹psubsumed-invs S› and
    clss0: ‹clauses0-inv S› and
    struct-invs: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state-of S)›
    using assms(2) unfolding pcdcl-all-struct-invs-def by fast+
  show ?thesis (is ‹?A ⟷ ?B›)
  proof
    assume ?B
    then show ?A
      by (auto simp: S T satisfiable-carac[symmetric])
  next
    assume ?A
    then obtain I where
      cons: ‹consistent-interp I› and
      IS: ‹I ⊨sm pget-all-init-clss S› and
      tot: ‹total-over-m I (set-mset (pget-all-init-clss S))›
      unfolding satisfiable-def by blast
    let ?J = ‹insert L (I - {#-L#})›
    have cons2: ‹consistent-interp ?J›
      using cons
      by (metis Diff-empty Diff-iff Diff-insert0 consistent-interp-insert-iff insert-Diff singletonI)

    have ‹all-decomposition-implies-m (cdcl_W-restart-mset.clauses (state-of S))
      (get-all-ann-decomposition (trail (state-of S)))› and
      alien: ‹cdcl_W-restart-mset.no-strange-atm (state-of S)›
      using struct-invs unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def by fast+
    moreover have ‹set-mset N ∪ set-mset NE ∪ set-mset NS ∪ set-mset N0 ⊨ps
      (set-mset U ∪ set-mset UE ∪ set-mset US ∪ set-mset U0)›
      using ent-init by (auto simp: S cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init-def)
    ultimately have ‹set-mset N ∪ set-mset NE ∪ set-mset NS ∪ set-mset N0 ⊨ps unmark-l M›
      using tot lev0 by (auto simp: S clauses-def count-decided-0-iff
        no-decision-get-all-ann-decomposition)
```

```
      (metis true-clss-clss-left-right true-clss-clss-union-and)
    moreover have ‹total-over-m I (unmark-l M)›
      using alien tot by (auto simp: cdcl_W-restart-mset.no-strange-atm-def S
        total-over-m-def total-over-set-def)
    ultimately have ‹I ⊨s unmark-l M›
      using IS tot cons by (auto simp: S true-clss-clss-def)
    then have ‹lits-of-l M ⊆ I›
      by (auto simp: true-clss-def lits-of-def)

    have IN: ‹I ⊨m N› and ‹I ⊨m NS› ‹I ⊨m N0› ‹I ⊨m NE›
      using IS by (auto simp: S)
    have totJ: ‹total-over-m ?J (set-mset (pget-all-init-clss T))›
      using tot apply (auto simp: total-over-m-def S T total-over-set-alt-def
        uminus-lit-swap)
      apply (metis atm-of-uminus literal.sel)+
      done
    have ‹?J ⊨m N›
      using IN L by (clarsimp simp: true-cls-mset-def add-mset-eq-add-mset true-cls-def
        dest!: multi-member-split)
    moreover have ‹?J ⊨m NE›
        using ‹I ⊨m NS› ent ‹I ⊨m N0› ‹I ⊨m NE› totJ L(1) undef ‹lits-of-l M ⊆ I›
        apply (auto simp: entailed-clss-inv-def S true-cls-mset-def T Decided-Propagated-in-iff-in-lits-of-l
          dest!: multi-member-split[of ‹- :: - clause›])
       unfolding true-cls-def[of ]
       apply clarsimp
       apply (rule-tac x=La in bexI)
       by auto
    moreover have ‹?J ⊨m N0›
      using ‹I ⊨m NS› ent ‹I ⊨m N0› ‹I ⊨m NE› totJ L undef clss0
      by (auto simp: entailed-clss-inv-def S true-cls-mset-def T Decided-Propagated-in-iff-in-lits-of-l
        clauses0-inv-def
        dest!: multi-member-split)
    moreover have ‹?J ⊨m NS›
      using ‹I ⊨m NS› sub ‹I ⊨m N0› ‹I ⊨m N› totJ calculation
      apply (auto simp: psubsumed-invs-def S true-cls-mset-def T
        dest!: multi-member-split dest: true-cls-mono-leD)
      apply (simp add: tautology-def)
      apply (metis calculation true-cls-mono-leD true-cls-mset-add-mset)+
      done
    ultimately have ‹?J ⊨sm pget-all-init-clss T›
      using ent by (auto simp: S T)
    then show ?B
      using cons2 by auto
  qed
qed

lemma pcdcl-core-same-init-vars:
  ‹pcdcl-core S T ⟹ atms-of-mm (pget-all-init-clss S) = atms-of-mm (pget-all-init-clss T)›
  by (induction rule: pcdcl-core.induct)
   (auto simp: cdcl-conflict.simps cdcl-propagate.simps cdcl-skip.simps
    cdcl-decide.simps cdcl-resolve.simps cdcl-backtrack.simps)

lemma pcdcl-restart-same-init-vars:
  ‹pcdcl-restart S T ⟹ atms-of-mm (pget-all-init-clss S) = atms-of-mm (pget-all-init-clss T)›
  by (induction rule: pcdcl-restart.induct) auto
```

lemma *pcdcl-restart-only-same-init-vars*:
‹*pcdcl-restart-only S T* ⟹ *atms-of-mm* (*pget-all-init-clss S*) = *atms-of-mm* (*pget-all-init-clss T*)›
by (*induction rule*: *pcdcl-restart-only.induct*) *auto*

lemma *pcdcl-satisfiable-iff*:
assumes
‹*pcdcl S T*› **and**
‹*pcdcl-all-struct-invs S*› **and**
ent-init: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state-of S*)›
shows
‹*satisfiable* (*set-mset* (*pget-all-init-clss S*)) ⟷ *satisfiable* (*set-mset* (*pget-all-init-clss T*))›
(**is** ‹*?A* ⟷ *?B*›)
using *assms pcdcl-pget-all-init-clss*[*OF assms*(*1*)]
proof −
have *atms-eq*: ‹*atms-of-mm* (*pget-all-init-clss S*) = *atms-of-mm* (*pget-all-init-clss T*)›
by (*rule pcdcl-pget-all-init-clss*[*OF assms*(*1*)])
(*use assms*(*2*) **in** ‹*auto simp*: *pcdcl-all-struct-invs-def*›)
show *?thesis*
proof
assume *?B*
with *assms* **show** *?A*
using *atms-eq*
by (*induction rule*: *pcdcl.induct*)
(*auto simp*: *pcdcl-core-entailed-iff satisfiable-carac*[*symmetric*]
cdcl-learn-clause.simps cdcl-resolution.simps
cdcl-subsumed.simps cdcl-flush-unit.simps cdcl-inp-propagate.simps
cdcl-inp-conflict.simps cdcl-unitres-true.simps cdcl-promote-false.simps
cdcl-pure-literal-remove.simps)
next
assume *?A*
then obtain *I* **where**
‹*I* ⊨*sm pget-all-init-clss S*› **and**
I: ‹*consistent-interp I*›
‹*total-over-m I* (*set-mset* (*pget-all-init-clss S*))›
unfolding *satisfiable-def* **by** *blast*
with *assms* **have** ‹¬*cdcl-pure-literal-remove S T* ⟹ *I* ⊨*sm pget-all-init-clss T*›
using *atms-eq*
apply (*induction rule*: *pcdcl.induct*)
subgoal by (*auto simp*: *pcdcl-core-entailed-iff satisfiable-carac*[*symmetric*])
subgoal by (*auto simp*: *cdcl-learn-clause.simps true-clss-cls-def total-over-m-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init-def satisfiable-def*)
subgoal by (*auto simp*: *cdcl-resolution.simps total-over-m-def consistent-interp-def*
satisfiable-carac[*symmetric*])
subgoal
by (*auto simp*: *cdcl-subsumed.simps satisfiable-carac*[*symmetric*])
subgoal
by (*auto simp*: *cdcl-flush-unit.simps satisfiable-carac*[*symmetric*])
subgoal
by (*auto simp*: *cdcl-inp-propagate.simps satisfiable-carac*[*symmetric*])
subgoal
by (*auto simp*: *cdcl-inp-conflict.simps satisfiable-carac*[*symmetric*])
subgoal
by (*auto simp*: *cdcl-unitres-true.simps satisfiable-carac*[*symmetric*])
subgoal
by (*auto simp*: *cdcl-promote-false.simps satisfiable-carac*[*symmetric*])
subgoal

by *fast*
done
then show *?B*
using *cdcl-pure-literal-remove-satisfiable-iff*[*of S T*] *I atms-eq assms* ‹*?A*›
by (*auto simp*: *satisfiable-carac*[*symmetric*] *intro*: *exI*[*of - I*])
qed
qed

lemma *rtranclp-pcdcl-entailed-iff*:
‹*pcdcl*** *S T* ⟹ *consistent-interp M* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
total-over-set M (*atms-of-mm* (*pget-all-init-clss T*)) ⟹ *M* ⊨*m pget-all-init-clss T* ⟹ *M* ⊨*m pget-all-init-clss S*›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *rtranclp-pcdcl-pget-all-init-clss*[*of S T*] *pcdcl-pget-all-init-clss*[*of T U*]
Pragmatic-CDCL.rtranclp-pcdcl-entailed-by-init[*of S T*]
rtranclp-pcdcl-all-struct-invs[*of S T*]
by (*auto dest*!: *pcdcl-entailed-iff*[*of - - M*] *simp del*: *atms-of-ms-union*)
done

lemma *pcdcl-restart-entailed-iff*:
‹*pcdcl-restart S T* ⟹ *M* ⊨*m pget-all-init-clss T* ⟷ *M* ⊨*m pget-all-init-clss S*›
by (*induction rule*: *pcdcl-restart.induct*) (*auto*)

lemma *pcdcl-inprocessing-entailed-iff*:
‹*pcdcl-inprocessing S T* ⟹ *consistent-interp M* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
total-over-set M (*atms-of-mm* (*pget-all-init-clss T*)) ⟹ *M* ⊨*m pget-all-init-clss T* ⟹ *M* ⊨*m pget-all-init-clss S*›
apply (*induction rule*: *pcdcl-inprocessing.induct*)
using *pcdcl-entailed-iff* **apply** *blast*
using *pcdcl-restart-entailed-iff* **by** *blast*

lemma *pcdcl-restart-satisfiable-iff*:
‹*pcdcl-restart S T* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
satisfiable (*set-mset* (*pget-all-init-clss T*)) ⟷ *satisfiable* (*set-mset* (*pget-all-init-clss S*))›
by (*induction rule*: *pcdcl-restart.induct*)
(*auto simp*: *satisfiable-carac*[*symmetric*])

lemma *pcdcl-inprocessing-satisfiable-iff*:
‹*pcdcl-inprocessing S T* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
satisfiable (*set-mset* (*pget-all-init-clss T*)) ⟷ *satisfiable* (*set-mset* (*pget-all-init-clss S*))›
apply (*induction rule*: *pcdcl-inprocessing.induct*)
subgoal for *S T*
using *pcdcl-satisfiable-iff*[*of S T*] **by** *blast*
using *pcdcl-restart-satisfiable-iff* **by** *blast*

lemma *rtranclp-pcdcl-inprocessing-entailed-iff*:
‹*pcdcl-inprocessing*** *S T* ⟹ *consistent-interp M* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
total-over-set M (*atms-of-mm* (*pget-all-init-clss T*)) ⟹ *M* ⊨*m pget-all-init-clss T* ⟹ *M* ⊨*m*

pget-all-init-clss S›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using
pcdcl-inprocessing-entailed-iff[*of T U M*] *rtranclp-pcdcl-inprocessing-entailed-by-init*[*of T U*]
rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs[*of S T*]
rtranclp-pcdcl-inprocessing-pget-all-init-clss[*of S T*]
pcdcl-inprocessing-pget-all-init-clss[*of T U*]
apply *auto*
using *rtranclp-pcdcl-inprocessing-entailed-by-init* **apply** *blast+*
done
done

lemma *rtranclp-pcdcl-inprocessing-satisfiable-iff*:
‹*pcdcl-inprocessing*** *S T* ⟹ *pcdcl-all-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
satisfiable (*set-mset* (*pget-all-init-clss T*)) ⟷ *satisfiable* (*set-mset* (*pget-all-init-clss S*))›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using
pcdcl-inprocessing-satisfiable-iff[*of T U*] *rtranclp-pcdcl-inprocessing-entailed-by-init*[*of T U*]
rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs[*of S T*]
rtranclp-pcdcl-inprocessing-pget-all-init-clss[*of S T*]
pcdcl-inprocessing-pget-all-init-clss[*of T U*]
apply *auto*
using *rtranclp-pcdcl-inprocessing-entailed-by-init* **apply** *blast+*
done
done

lemma *pcdcl-restart-only-entailed-iff*:
‹*pcdcl-restart-only S T* ⟹ *M* ⊨*m pget-all-init-clss T* ⟷ *M* ⊨*m pget-all-init-clss S*›
by (*induction rule*: *pcdcl-restart-only.induct*) (*auto*)

lemma *pcdcl-stgy-restart-same-init-vars*:
‹*pcdcl-stgy-restart* (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1′*, *R2′*, *T*, *m′*, *n′*, *g′*) ⟹
pcdcl-all-struct-invs S ⟹
atms-of-mm (*pget-all-init-clss S*) = *atms-of-mm* (*pget-all-init-clss T*)›
apply (*induction* ‹(*R1*, *R2*, *S*, *m*, *n*, *g*)› ‹(*R1′*, *R2′*, *T*, *m′*, *n′*, *g′*)› *rule*: *pcdcl-stgy-restart.induct*)
subgoal
by (*auto dest*!: *pcdcl-restart-only-entailed-iff pcdcl-restart-entailed-iff*
dest!: *rtranclp-pcdcl-stgy-pcdcl pcdcl-tcore-stgy-pcdcl-stgy′*
simp: *rtranclp-pcdcl-pget-all-init-clss*)
subgoal for *U*
apply (*auto simp*: *pcdcl-restart-same-init-vars rtranclp-pcdcl-pget-all-init-clss*
dest!: *rtranclp-pcdcl-stgy-pcdcl pcdcl-tcore-stgy-pcdcl-stgy′*)
using *rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs rtranclp-pcdcl-inprocessing-pget-all-init-clss rtranclp-pcdcl-pcdcl-pget-all-init-clss* **apply** *blast*
by (*smt pcdcl-restart-pcdcl-all-struct-invs pcdcl-restart-same-init-vars rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs rtranclp-pcdcl-inprocessing-pget-all-init-clss rtranclp-pcdcl-pget-all-init-clss*)
subgoal
by (*auto simp*: *pcdcl-restart-only-same-init-vars*)
subgoal
by *auto*

done

lemma *rtranclp-pcdcl-stgy-restart-same-init-vars*:
‹*pcdcl-stgy-restart*** (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*) ⟹
pcdcl-all-struct-invs S ⟹
atms-of-mm (*pget-all-init-clss S*) = *atms-of-mm* (*pget-all-init-clss T*)›
apply (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -, -, -, -)› ‹(-, -, -, -, -, -)›, *split-format*(*complete*), *of* **for** *r*])
subgoal
by (*auto dest*!: *pcdcl-restart-only-entailed-iff pcdcl-restart-entailed-iff*
dest!: *rtranclp-pcdcl-stgy-pcdcl pcdcl-tcore-stgy-pcdcl-stgy'*
rtranclp-pcdcl-stgy-pcdcl simp: *rtranclp-pcdcl-pget-all-init-clss*)
subgoal
by (*auto dest*: *rtranclp-pcdcl-stgy-restart-pcdcl-all-struct-invs*
pcdcl-stgy-restart-same-init-vars)
done

lemma *pcdcl-stgy-restart-entailed-iff*:
‹*pcdcl-stgy-restart* (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*) ⟹
pcdcl-all-struct-invs S ⟹ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
consistent-interp M ⟹ *total-over-set M* (*atms-of-mm* (*pget-all-init-clss T*)) ⟹
M ⊨*m pget-all-init-clss T* ⟹ *M* ⊨*m pget-all-init-clss S*›
apply (*induction* ‹(*R1*, *R2*, *S*, *m*, *n*, *g*)› ‹(*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*)› *rule*: *pcdcl-stgy-restart.induct*)
apply (*auto dest*: *pcdcl-restart-only-entailed-iff pcdcl-restart-entailed-iff*
dest: *rtranclp-pcdcl-stgy-pcdcl pcdcl-tcore-stgy-pcdcl-stgy'*
rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-entailed-iff)[]
apply (*smt pcdcl-restart-entailed-iff pcdcl-restart-pcdcl-all-struct-invs pcdcl-restart-same-init-vars*
rtranclp-pcdcl-entailed-iff rtranclp-pcdcl-inprocessing-entailed-by-init
rtranclp-pcdcl-inprocessing-entailed-iff rtranclp-pcdcl-inprocessing-pcdcl-all-struct-invs
rtranclp-pcdcl-stgy-pcdcl rtranclp-pcdcl-stgy-pget-all-init-clss twl-restart-ops.pcdcl-restart-entailed-by-init)
using *pcdcl-restart-only-entailed-iff* **apply** *blast*
by *simp*

lemma *rtranclp-pcdcl-restart-entailed-iff*:
‹*pcdcl-stgy-restart*** (*R1*, *R2*, *S*, *m*, *n*, *g*) (*R1'*, *R2'*, *T*, *m'*, *n'*, *g'*) ⟹
pcdcl-all-struct-invs S ⟹ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state-of S*) ⟹
consistent-interp M ⟹ *total-over-set M* (*atms-of-mm* (*pget-all-init-clss T*)) ⟹
M ⊨*m pget-all-init-clss T* ⟹ *M* ⊨*m pget-all-init-clss S*›
apply (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -, -, -, -)› ‹(-, -, -, -, -, -)›, *split-format*(*complete*), *of* **for** *r*])
subgoal by *auto*
subgoal for *T U*
by (*simp add*: *pcdcl-stgy-restart-entailed-iff pcdcl-stgy-restart-same-init-vars*
rtranclp-pcdcl-stgy-restart-entailed-by-init rtranclp-pcdcl-stgy-restart-pcdcl-all-struct-invs)
done

lemma [*simp*]: ‹*pget-all-init-clss* (*pstate*$_W$*-of S*) = (*get-all-init-clss S*)›
by (*cases S*) *auto*

lemma *empty-entails-novars-iff*: ‹*atms-of-mm S* = {} ⟹ {} ⊨*ps set-mset S* ⟷ *S* = {#}›
unfolding *true-clss-clss-def*
by (*cases S*) (*auto simp*: *satisfiable-def total-over-m-def intro*: *Ex-consistent-interp*)
end

end
theory *Watched-Literals-Algorithm-Restart*

imports *Watched-Literals-Algorithm Watched-Literals-Transition-System-Restart*
Watched-Literals-Transition-System-Reduce
Weidenbach-Book-Base.Explorer
begin

end
theory *Watched-Literals-List*
imports *More-Sepref.WB-More-Refinement-List Watched-Literals-Algorithm CDCL.DPLL-CDCL-W-Implementation*
Watched-Literals-Clauses
begin

Chapter 5

Second Refinement: Lists as Clause

declare *RETURN-as-SPEC-refine*[*refine2*]

lemma *mset-take-mset-drop-mset*: ‹(λ*x*. *mset* (*take* 2 *x*) + *mset* (*drop* 2 *x*)) = *mset*›
  unfolding *mset-append*[*symmetric*] *append-take-drop-id* **..**

lemma *mset-take-mset-drop-mset'*: ‹*mset* (*take* 2 *x*) + *mset* (*drop* 2 *x*) = *mset x*›
  unfolding *mset-append*[*symmetric*] *append-take-drop-id* **..**

lemma *uminus-lit-of-image-mset*:
  ‹{#− *lit-of x* . *x* ∈# *A*#} = {#− *lit-of x*. *x* ∈# *B*#} ⟷
    {#*lit-of x* . *x* ∈# *A*#} = {#*lit-of x*. *x* ∈# *B*#}›
  for *A* :: ‹('*a literal*, '*a literal*, '*b*) *annotated-lit multiset*›
proof −
  have *1*: ‹(λ*x*. −*lit-of x*) `# *A* = *uminus* `# *lit-of* `# *A*›
    for *A* :: ‹('*d*::*uminus*, '*d*, '*e*) *annotated-lit multiset*›
    by *auto*
  show *?thesis*
    unfolding *1*
    by (*rule inj-image-mset-eq-iff*) (*auto simp*: *inj-on-def*)
qed

lemma *twl-struct-invs-no-alien-in-trail*:
  assumes ‹*twl-struct-invs S*›
  shows ‹*L* ∈ *lits-of-l* (*get-trail S*) ⟹ *L* ∈# *all-lits-of-st S*›
  using *assms* **by** (*cases S*)
   (*auto simp*: *twl-struct-invs-def pcdcl-all-struct-invs-def*
    cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def all-lits-of-st-def*
    cdcl$_W$*-restart-mset.no-strange-atm-def trail.simps*
    init-clss.simps learned-clss.simps
    in-all-lits-of-mm-ain-atms-of-iff)

5.1 Types

type-synonym '*v clauses-to-update-l* = ‹*nat multiset*›

type-synonym '*v cconflict* = ‹'*v clause option*›
type-synonym '*v cconflict-l* = ‹'*v literal list option*›

type-synonym '*v twl-st-l* =
  ‹('*v*, *nat*) *ann-lits* × '*v clauses-l* × '*v cconflict* × '*v clauses* × '*v clauses* × '*v clauses* ×

'v clauses × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses-to-update-l* × *'v lit-queue*›

fun *clauses-to-update-l* :: ‹*'v twl-st-l* ⇒ *'v clauses-to-update-l*› **where**
‹*clauses-to-update-l* (-, -, -, -, -, -, -, -, -,-, -, *WS*, -) = *WS*›

fun *get-trail-l* :: ‹*'v twl-st-l* ⇒ (*'v*, *nat*) *ann-lit list*› **where**
‹*get-trail-l* (*M*, -, -, -, -, -, -) = *M*›

fun *set-clauses-to-update-l* :: ‹*'v clauses-to-update-l* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*set-clauses-to-update-l WS* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, -, *Q*) =
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›

fun *literals-to-update-l* :: ‹*'v twl-st-l* ⇒ *'v clause*› **where**
‹*literals-to-update-l* (-, -, -, -, -, -, -, -, -, -, -,*Q*) = *Q*›

fun *set-literals-to-update-l* :: ‹*'v clause* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*set-literals-to-update-l Q* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, -) =
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›

fun *get-conflict-l* :: ‹*'v twl-st-l* ⇒ *'v cconflict*› **where**
‹*get-conflict-l* (-, -, *D*, -, -, -, -) = *D*›

fun *get-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses-l*› **where**
‹*get-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *N*›

fun *get-unit-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-unit-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NE* + *NEk* + *UE* + *UEk*›

fun *get-kept-unit-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-kept-unit-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NEk*+ *UEk*›

fun *get-unkept-unit-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-unkept-unit-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NE*+ *UE*›

fun *get-unit-init-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-unit-init-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NE* + *NEk*›

fun *get-unit-learned-clss-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-unit-learned-clss-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *UE* + *UEk*›

fun *get-kept-init-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-kept-init-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NEk*›

fun *get-kept-learned-clss-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-kept-learned-clss-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *UEk*›

fun *get-unkept-init-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-unkept-init-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NE*›

fun *get-unkept-learned-clss-l* :: ‹*'v twl-st-l* ⇒ *'v clauses*› **where**
‹*get-unkept-learned-clss-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *UE*›

fun *get-init-clauses* :: ‹*'v twl-st* ⇒ *'v twl-clss*› **where**
‹*get-init-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *N*›

definition *get-learned-clss-l* :: ‹*'v twl-st-l* ⇒ *'v clause-l multiset*› **where**

‹*get-learned-clss-l S = learned-clss-lf* (*get-clauses-l S*)›

definition *get-init-clss-l* :: ‹'*v twl-st-l* ⇒ '*v clause-l multiset*› **where**
‹*get-init-clss-l S = init-clss-lf* (*get-clauses-l S*)›

fun *get-subsumed-init-clauses-l* :: ‹'*v twl-st-l* ⇒ '*v clauses*› **where**
‹*get-subsumed-init-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *NS*›

fun *get-subsumed-learned-clauses-l* :: ‹'*v twl-st-l* ⇒ '*v clauses*› **where**
‹*get-subsumed-learned-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) = *US*›

fun *get-subsumed-clauses-l* :: ‹'*v twl-st-l* ⇒ '*v clauses*› **where**
‹*get-subsumed-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *NS* + *US*›

fun *get-init-clauses0-l* :: ‹'*v twl-st-l* ⇒ '*v clauses*› **where**
‹*get-init-clauses0-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *N0*›

fun *get-learned-clauses0-l* :: ‹'*v twl-st-l* ⇒ '*v clauses*› **where**
‹*get-learned-clauses0-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *U0*›

fun *get-clauses0-l* :: ‹'*v twl-st-l* ⇒ '*v clauses*› **where**
‹*get-clauses0-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *N0* + *U0*›

abbreviation *get-all-clss-l* :: ‹'*v twl-st-l* ⇒ '*v clause multiset*› **where**
‹*get-all-clss-l S* ≡
mset ‘# *ran-mf* (*get-clauses-l S*) + *get-unit-clauses-l S* + *get-subsumed-clauses-l S* + *get-clauses0-l S*›

abbreviation *get-all-init-clss-l* :: ‹'*v twl-st-l* ⇒ '*v clause multiset*› **where**
‹*get-all-init-clss-l S* ≡ *mset* ‘# *get-init-clss-l S* + *get-unit-init-clauses-l S* +
get-subsumed-init-clauses-l S + *get-init-clauses0-l S*›

abbreviation *get-all-learned-clss-l* :: ‹'*v twl-st-l* ⇒ '*v clause multiset*› **where**
‹*get-all-learned-clss-l S* ≡ *mset* ‘# *get-learned-clss-l S* + *get-unit-learned-clss-l S* +
get-subsumed-learned-clauses-l S + *get-learned-clauses0-l S*›

lemma *state-decomp-to-state*:
‹(*case S of* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⇒ *P M N U D NE UE NS US N0 U0 WS Q*) =
P (*get-trail S*) (*get-init-clauses S*) (*get-learned-clss S*) (*get-conflict S*)
(*unit-init-clauses S*) (*get-init-learned-clss S*)
(*subsumed-init-clauses S*) (*subsumed-learned-clauses S*)
(*get-init-clauses0 S*) (*get-learned-clauses0 S*)
(*clauses-to-update S*)
(*literals-to-update S*)›
by (*cases S*) *auto*

lemma *state-decomp-to-state-l*:
‹(*case S of* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⇒ *P M N D NE UE NEk UEk NS US N0 U0 WS Q*) =
P (*get-trail-l S*) (*get-clauses-l S*) (*get-conflict-l S*)
(*get-unkept-init-clauses-l S*) (*get-unkept-learned-clss-l S*)
(*get-kept-init-clauses-l S*) (*get-kept-learned-clss-l S*)
(*get-subsumed-init-clauses-l S*) (*get-subsumed-learned-clauses-l S*)
(*get-init-clauses0-l S*) (*get-learned-clauses0-l S*)

(*clauses-to-update-l S*)
(*literals-to-update-l S*)›
by (*cases S*) *auto*

definition *set-conflict'* :: ‹'*v clause option* ⇒ '*v twl-st* ⇒ '*v twl-st*› **where**
‹*set-conflict'* = (λ*C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*). (*M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*))›

definition *all-lits-of-st-l* :: ‹'*v twl-st-l* ⇒ '*v literal multiset*› **where**
‹*all-lits-of-st-l S* = *all-lits-of-mm* (*get-all-clss-l S*)›

lemma *get-unit-clauses-l-alt-def*:
‹*get-unit-clauses-l S* = *get-unkept-unit-clauses-l S* + *get-kept-unit-clauses-l S*›
by (*cases S*) (*auto*)

inductive *convert-lit*
:: ‹'*v clauses-l* ⇒ '*v clauses* ⇒ ('*v*, *nat*) *ann-lit* ⇒ ('*v*, '*v clause*) *ann-lit* ⇒ *bool*›
where
‹*convert-lit N E* (*Decided K*) (*Decided K*)› |
‹*convert-lit N E* (*Propagated K C*) (*Propagated K C'*)›
if ‹*C'* = *mset* (*N* ∝ *C*)› ‹*C* ∈# *dom-m N*› **and** ‹*C* ≠ *0*› |
‹*convert-lit N E* (*Propagated K C*) (*Propagated K C'*)›
if ‹*C* = *0*› **and** ‹*C'* ∈# *E*›

definition *convert-lits-l* **where**
‹*convert-lits-l N E* = ⟨*p2rel* (*convert-lit N E*)⟩ *list-rel*›

lemma *convert-lits-l-nil*[*simp*]:
‹([], *a*) ∈ *convert-lits-l N E* ⟷ *a* = []›
‹(*b*, []) ∈ *convert-lits-l N E* ⟷ *b* = []›
by (*auto simp*: *convert-lits-l-def*)

lemma *convert-lits-l-cons*[*simp*]:
‹(*L* # *M*, *L'* # *M'*) ∈ *convert-lits-l N E* ⟷
convert-lit N E L L' ∧ (*M*, *M'*) ∈ *convert-lits-l N E*›
by (*auto simp*: *convert-lits-l-def p2rel-def*)

lemma *take-convert-lits-lD*:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
(*take n M*, *take n M'*) ∈ *convert-lits-l N E*›
by (*auto simp*: *convert-lits-l-def list-rel-def*)

lemma *convert-lits-l-consE*:
‹(*Propagated L C* # *M*, *x*) ∈ *convert-lits-l N E* ⟹
(⋀*L' C' M'*. *x* = *Propagated L' C'* # *M'* ⟹ (*M*, *M'*) ∈ *convert-lits-l N E* ⟹
convert-lit N E (*Propagated L C*) (*Propagated L' C'*) ⟹ *P*) ⟹ *P*›
by (*cases x*) (*auto simp*: *convert-lit.simps*)

lemma *convert-lits-l-append*[*simp*]:
‹*length M1* = *length M1'* ⟹
(*M1* @ *M2*, *M1'* @ *M2'*) ∈ *convert-lits-l N E* ⟷ (*M1*, *M1'*) ∈ *convert-lits-l N E* ∧
(*M2*, *M2'*) ∈ *convert-lits-l N E* ›
by (*auto simp*: *convert-lits-l-def list-rel-append2 list-rel-imp-same-length*)

lemma *convert-lits-l-map-lit-of*: ‹(*ay*, *bq*) ∈ *convert-lits-l N e* ⟹ *map lit-of ay* = *map lit-of bq*›

apply (*induction ay arbitrary*: *bq*)
subgoal by *auto*
subgoal for *L M bq* **by** (*cases bq*) (*auto simp*: *convert-lit.simps*)
done

lemma *convert-lits-l-tlD*:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
(*tl M*, *tl M'*) ∈ *convert-lits-l N E*›
by (*cases M*; *cases M'*) *auto*

lemma *convert-lits-l-swap*:
‹*i* < *length* (*N* ∝ *C*) ⟹ *j* < *length* (*N* ∝ *C*) ⟹ *C* ∈# *dom-m N* ⟹
(*M*, *M'*) ∈ *convert-lits-l* (*N*(*C* ↪*swap* (*N* ∝ *C*) *i j*)) *E* ⟷
(*M*, *M'*) ∈ *convert-lits-l N E*›
by (*fastforce simp*: *convert-lits-l-def list-rel-def p2rel-def*
convert-lit.simps list-all2-conv-all-nth)

lemma *get-clauses-l-set-clauses-to-update-l*[*simp*]:
‹*get-clauses-l* (*set-clauses-to-update-l WC S*) = *get-clauses-l S*›
by (*cases S*) *auto*

lemma *get-trail-l-set-clauses-to-update-l*[*simp*]:
‹*get-trail-l* (*set-clauses-to-update-l WC S*) = *get-trail-l S*›
by (*cases S*) *auto*

lemma *get-trail-set-clauses-to-update*[*simp*]:
‹*get-trail* (*set-clauses-to-update WC S*) = *get-trail S*›
by (*cases S*) *auto*

lemma *convert-lits-l-add-mset*:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
(*M*, *M'*) ∈ *convert-lits-l N* (*add-mset C E*)›
by (*fastforce simp*: *convert-lits-l-def list-rel-def p2rel-def*
convert-lit.simps list-all2-conv-all-nth)

lemma *convert-lit-bind-new*:
‹*C* ∉# *dom-m N* ⟹ *C* > *0* ⟹
convert-lit N E L L'⟹
convert-lit (*N*(*C* ↪ *C'*)) *E L L'*›
‹*C* ∉# *dom-m N* ⟹ *C* > *0* ⟹
convert-lit N E L L'⟹
convert-lit (*fmupd C C'' N*) *E L L'*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lits-l-bind-new*:
‹*C* ∉# *dom-m N* ⟹ *C* > *0* ⟹
(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
(*M*, *M'*) ∈ *convert-lits-l* (*N*(*C* ↪ *C'*)) *E*›
‹*C* ∉# *dom-m N* ⟹ *C* > *0* ⟹
(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
(*M*, *M'*) ∈ *convert-lits-l* (*fmupd C C'' N*) *E*›
by (*auto simp*: *convert-lits-l-def list-rel-def p2rel-def*
list-all2-conv-all-nth convert-lit-bind-new)

abbreviation *resolve-cls-l* **where**
‹*resolve-cls-l L D' E* ≡ *union-mset-list* (*remove1* (−*L*) *D'*) (*remove1 L E*)›

lemma *mset-resolve-cls-l-resolve-cls*[*iff*]:
‹*mset* (*resolve-cls-l L D′ E*) = *cdcl*$_W$*-restart-mset.resolve-cls L* (*mset D′*) (*mset E*)›
by (*auto simp*: *union-mset-list*[*symmetric*])

lemma *resolve-cls-l-nil-iff*:
‹*resolve-cls-l L D′ E* = [] ⟷ *cdcl*$_W$*-restart-mset.resolve-cls L* (*mset D′*) (*mset E*) = {#}›
by (*metis mset-resolve-cls-l-resolve-cls mset-zero-iff*)

lemma *lit-of-convert-lit*[*simp*]:
‹*convert-lit N E L L′* ⟹ *lit-of L′* = *lit-of L*›
by (*auto simp*: *p2rel-def convert-lit.simps*)

lemma *is-decided-convert-lit*[*simp*]:
‹*convert-lit N E L L′* ⟹ *is-decided L′* ⟷ *is-decided L*›
by (*cases L*) (*auto simp*: *p2rel-def convert-lit.simps*)

lemma *defined-lit-convert-lits-l*[*simp*]: ‹(*M*, *M′*) ∈ *convert-lits-l N E* ⟹
defined-lit M′ = *defined-lit M*›
apply (*induction M arbitrary*: *M′*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *defined-lit-cons*)
done

lemma *no-dup-convert-lits-l*[*simp*]: ‹(*M*, *M′*) ∈ *convert-lits-l N E* ⟹
no-dup M′ ⟷ *no-dup M*›
apply (*induction M arbitrary*: *M′*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*) *auto*
done

lemma
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
count-decided-convert-lits-l[*simp*]:
‹*count-decided M′* = *count-decided M*›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def*)
done

lemma
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
get-level-convert-lits-l[*simp*]:
‹*get-level M′* = *get-level M*›
using *assms*

apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*fastforce simp*: *convert-lits-l-def p2rel-def get-level-cons-if split*: *if-splits*)+
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def get-level-cons-if*)
done

lemma
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
get-maximum-level-convert-lits-l[*simp*]:
‹*get-maximum-level M′* = *get-maximum-level M*›
by (*intro ext*, *rule get-maximum-level-cong*)
(*use assms* **in** *auto*)

lemma *list-of-l-convert-map-lit-of*:
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
‹*map lit-of M′* = *map lit-of M*›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def*)
done

lemma *list-of-l-convert-lits-l*[*simp*]:
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
‹*lits-of-l M′* = *lits-of-l M*›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def*)
done

lemma *is-proped-hd-convert-lits-l*[*simp*]:
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*› **and** ‹*M* ≠ []›
shows ‹*is-proped* (*hd M′*) ⟷ *is-proped* (*hd M*)›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*)

done

lemma *is-decided-hd-convert-lits-l*[*simp*]:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*› **and** ‹*M* ≠ []›
shows
‹*is-decided* (*hd M'*) ⟷ *is-decided* (*hd M*)›
by (*meson assms*(*1*) *assms*(*2*) *is-decided-no-proped-iff is-proped-hd-convert-lits-l*)

lemma *lit-of-hd-convert-lits-l*[*simp*]:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*› **and** ‹*M* ≠ []›
shows
‹*lit-of* (*hd M'*) = *lit-of* (*hd M*)›
by (*cases M*; *cases M'*) (*use assms* **in** *auto*)

lemma *lit-of-l-convert-lits-l*[*simp*]:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows
‹*lit-of* ‘ *set M'* = *lit-of* ‘ *set M*›
using *assms*
apply (*induction M arbitrary*: *M' rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M'*
by (*cases M'*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M'*
by (*cases M'*) (*auto simp*: *convert-lits-l-def p2rel-def*)
done

The order of the assumption is important for simpler use.

lemma *convert-lits-l-extend-mono*:
assumes ‹(*a*,*b*) ∈ *convert-lits-l N E*›
‹∀ *L i*. *Propagated L i* ∈ *set a* ∧ *i* ∈# *dom-m N* ⟶
mset (*N*∝*i*) = *mset* (*N'*∝*i*) ∧ *i* ∈# *dom-m N'*› **and**
‹*E* ⊆# *E'*›
shows
‹(*a*,*b*) ∈ *convert-lits-l N' E'*›
using *assms*
apply (*induction a arbitrary*: *b rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *l A b*
by (*cases b*)
(*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*)
subgoal for *l C A b*
by (*cases b*)
(*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*)
done

lemma *convert-lits-l-nil-iff*[*simp*]:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows
‹*M'* = [] ⟷ *M* = []›
using *assms* **by** *auto*

lemma *convert-lits-l-atm-lits-of-l*:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows ‹*atm-of* ‘ *lits-of-l M* = *atm-of* ‘ *lits-of-l M'*›

using *assms* **by** *auto*

lemma *convert-lits-l-true-clss-clss*[*simp*]:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹ *M'* ⊨*as C* ⟷ *M* ⊨*as C*›
unfolding *true-annots-true-cls*
by (*auto simp*: *p2rel-def*)

lemma *convert-lit-propagated-decided*[*iff*]:
‹*convert-lit b d* (*Propagated x21 x22*) (*Decided x1*) ⟷ *False*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lit-decided*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Decided x2*) ⟷ *x1* = *x2*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lit-decided-propagated*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Propagated x21 x22*) ⟷ *False*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lits-l-lit-of-mset*[*simp*]:
‹(*a*, *af*) ∈ *convert-lits-l N E* ⟹ *lit-of* ‘# *mset af* = *lit-of* ‘# *mset a*›
apply (*induction a arbitrary*: *af*)
subgoal by *auto*
subgoal for *L M af*
by (*cases af*) *auto*
done

lemma *convert-lits-l-imp-same-length*:
‹(*a*, *b*) ∈ *convert-lits-l N E* ⟹ *length a* = *length b*›
by (*auto simp*: *convert-lits-l-def list-rel-imp-same-length*)

lemma *convert-lits-l-decomp-ex*:
assumes
H: ‹(*Decided K* # *a*, *M2*) ∈ *set* (*get-all-ann-decomposition x*)› **and**
xxa: ‹(*x*, *xa*) ∈ *convert-lits-l aa ac*›
shows ‹∃ *M2*. (*Decided K* # *drop* (*length xa* − *length a*) *xa*, *M2*)
∈ *set* (*get-all-ann-decomposition xa*)› (**is** *?decomp*) **and**
‹(*a*, *drop* (*length xa* − *length a*) *xa*) ∈ *convert-lits-l aa ac*› (**is** *?a*)
proof −
from *H* **obtain** *M3* **where**
x: ‹*x* = *M3* @ *M2* @ *Decided K* # *a*›
by *blast*
obtain *M3'* *M2'* *a'* **where**
xa: ‹*xa* = *M3'* @ *M2'* @ *Decided K* # *a'*› **and**
‹(*M3*, *M3'*) ∈ *convert-lits-l aa ac*› **and**
‹(*M2*, *M2'*) ∈ *convert-lits-l aa ac*› **and**
aa': ‹(*a*, *a'*) ∈ *convert-lits-l aa ac*›
using *xxa* **unfolding** *x*
by (*auto simp*: *list-rel-append1 convert-lits-l-def p2rel-def convert-lit.simps*
list-rel-split-right-iff)
then have *a'*: ‹*a'* = *drop* (*length xa* − *length a*) *xa*› **and** [*simp*]: ‹*length xa* ≥ *length a*›
unfolding *xa* **by** (*auto simp*: *convert-lits-l-imp-same-length*)
show *?decomp*
using *get-all-ann-decomposition-ex*[*of K a'* ‹*M3'* @ *M2'*›]
unfolding *xa*

unfolding a'
by *auto*
show *?a*
using aa' **unfolding** a' **.**
qed

lemma *in-convert-lits-lD*:
‹$K \in$ *set* $TM \Longrightarrow$
$(M,\ TM) \in$ *convert-lits-l* N $NE \Longrightarrow$
$\exists K'.\ K' \in$ *set* $M \wedge$ *convert-lit* N NE K' K›
by (*auto 5 5 simp*: *convert-lits-l-def list-rel-append2 dest*!: *split-list p2relD elim*!: *list-relE*)

lemma *in-convert-lits-lD2*:
‹$K \in$ *set* $M \Longrightarrow$
$(M,\ TM) \in$ *convert-lits-l* N $NE \Longrightarrow$
$\exists K'.\ K' \in$ *set* $TM \wedge$ *convert-lit* N NE K K'›
by (*auto 5 5 simp*: *convert-lits-l-def list-rel-append1 dest*!: *split-list p2relD elim*!: *list-relE*)

lemma *convert-lits-l-filter-decided*: ‹$(S,\ S') \in$ *convert-lits-l* M $N \Longrightarrow$
map lit-of (*filter is-decided* S') = *map lit-of* (*filter is-decided* S)›
apply (*induction* S *arbitrary*: S')
subgoal by *auto*
subgoal for L S S'
by (*cases* S') *auto*
done

lemma *convert-lits-lI*:
‹*length* M = *length* $M' \Longrightarrow (\bigwedge i.\ i <$ *length* $M \Longrightarrow$ *convert-lit* N NE $(M!i)$ $(M'!i)) \Longrightarrow$
$(M,\ M') \in$ *convert-lits-l* N NE›
by (*auto simp*: *convert-lits-l-def list-rel-def p2rel-def list-all2-conv-all-nth*)

fun *get-learned-clauses-l* :: ‹'*v twl-st-l* ⇒ '*v clause-l multiset*› **where**
‹*get-learned-clauses-l* $(M,\ N,\ D,\ NE,\ UE,\ WS,\ Q)$ = *learned-clss-lf* N›

lemma *get-subsumed-clauses-l-simps*[*simp*]:
‹*get-subsumed-init-clauses-l* (*set-clauses-to-update-l* K S) = *get-subsumed-init-clauses-l* S›
‹*get-subsumed-learned-clauses-l* (*set-clauses-to-update-l* K S) = *get-subsumed-learned-clauses-l* S›
‹*get-subsumed-clauses-l* (*set-clauses-to-update-l* K S) = *get-subsumed-clauses-l* S›
by (*solves* ‹*cases* S; *auto*›)+

definition *twl-st-l* :: ‹- ⇒ ('*v twl-st-l* × '*v twl-st*) *set*› **where**
‹*twl-st-l* L =
{(($M,\ N,\ C,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$), ($M',\ N',\ U',\ C',\ NE',\ UE',\ NS',\ US',\ N0',\ U0',\ WS',\ Q'$)).
$(M,\ M') \in$ *convert-lits-l* N $(NEk+UEk) \wedge$
$N' =$ *twl-clause-of* ‘# *init-clss-lf* $N \wedge$
$U' =$ *twl-clause-of* ‘# *learned-clss-lf* $N \wedge$
$C' = C \wedge$
$NE' = NE+NEk \wedge$
$UE' = UE + UEk \wedge$
$NS' = NS \wedge$
$US' = US \wedge$
$N0' = N0 \wedge$
$U0' = U0 \wedge$

$WS' = (\mathit{case}\ L\ \mathit{of}\ \mathit{None} \Rightarrow \{\#\} \mid \mathit{Some}\ L \Rightarrow \mathit{image\text{-}mset}\ (\lambda j.\ (L,\ \mathit{twl\text{-}clause\text{-}of}\ (N \propto j)))\ WS) \wedge$
$Q' = Q$
}›

lemma *clss-state$_W$-of*[*twl-st*]:
assumes ‹$(S,\ R) \in \mathit{twl\text{-}st\text{-}l}\ L$›
shows
‹$\mathit{init\text{-}clss}\ (\mathit{state}_W\text{-}\mathit{of}\ R) = \mathit{mset}\ `\#\ (\mathit{init\text{-}clss\text{-}lf}\ (\mathit{get\text{-}clauses\text{-}l}\ S)) +$
$\mathit{get\text{-}unit\text{-}init\text{-}clauses\text{-}l}\ S + \mathit{get\text{-}subsumed\text{-}init\text{-}clauses\text{-}l}\ S + \mathit{get\text{-}init\text{-}clauses0\text{-}l}\ S$›
‹$\mathit{learned\text{-}clss}\ (\mathit{state}_W\text{-}\mathit{of}\ R) = \mathit{mset}\ `\#\ (\mathit{learned\text{-}clss\text{-}lf}\ (\mathit{get\text{-}clauses\text{-}l}\ S)) +$
$\mathit{get\text{-}unit\text{-}learned\text{-}clss\text{-}l}\ S + \mathit{get\text{-}subsumed\text{-}learned\text{-}clauses\text{-}l}\ S + \mathit{get\text{-}learned\text{-}clauses0\text{-}l}\ S$›
using *assms*
by (*cases S*; *cases R*; *cases L*; *auto simp*: *init-clss.simps learned-clss.simps twl-st-l-def*
mset-take-mset-drop-mset′)+

named-theorems *twl-st-l* ‹*Conversions simp rules*›

lemma [*twl-st-l*]:
assumes ‹$(S,\ T) \in \mathit{twl\text{-}st\text{-}l}\ L$›
shows
‹$(\mathit{get\text{-}trail\text{-}l}\ S,\ \mathit{get\text{-}trail}\ T) \in \mathit{convert\text{-}lits\text{-}l}\ (\mathit{get\text{-}clauses\text{-}l}\ S)\ (\mathit{get\text{-}kept\text{-}unit\text{-}clauses\text{-}l}\ S)$› **and**
‹$\mathit{get\text{-}clauses}\ T = \mathit{twl\text{-}clause\text{-}of}\ `\#\ \mathit{fst}\ `\#\ \mathit{ran\text{-}m}\ (\mathit{get\text{-}clauses\text{-}l}\ S)$› **and**
‹$\mathit{get\text{-}conflict}\ T = \mathit{get\text{-}conflict\text{-}l}\ S$› **and**
‹$L = \mathit{None} \Longrightarrow \mathit{clauses\text{-}to\text{-}update}\ T = \{\#\}$›
‹$L \neq \mathit{None} \Longrightarrow \mathit{clauses\text{-}to\text{-}update}\ T =$
$(\lambda j.\ (\mathit{the}\ L,\ \mathit{twl\text{-}clause\text{-}of}\ (\mathit{get\text{-}clauses\text{-}l}\ S \propto j)))\ `\#\ \mathit{clauses\text{-}to\text{-}update\text{-}l}\ S$› **and**
‹$\mathit{literals\text{-}to\text{-}update}\ T = \mathit{literals\text{-}to\text{-}update\text{-}l}\ S$›
‹$\mathit{backtrack\text{-}lvl}\ (\mathit{state}_W\text{-}\mathit{of}\ T) = \mathit{count\text{-}decided}\ (\mathit{get\text{-}trail\text{-}l}\ S)$›
‹$\mathit{unit\text{-}clss}\ T = \mathit{get\text{-}unit\text{-}clauses\text{-}l}\ S$›
‹$\mathit{cdcl}_W\text{-}\mathit{restart\text{-}mset.clauses}\ (\mathit{state}_W\text{-}\mathit{of}\ T) =$
$\mathit{mset}\ `\#\ \mathit{ran\text{-}mf}\ (\mathit{get\text{-}clauses\text{-}l}\ S) + \mathit{get\text{-}unit\text{-}clauses\text{-}l}\ S + \mathit{get\text{-}subsumed\text{-}clauses\text{-}l}\ S +$
$\mathit{get\text{-}clauses0\text{-}l}\ S$› **and**
‹$\mathit{no\text{-}dup}\ (\mathit{get\text{-}trail}\ T) \longleftrightarrow \mathit{no\text{-}dup}\ (\mathit{get\text{-}trail\text{-}l}\ S)$› **and**
‹$\mathit{lits\text{-}of\text{-}l}\ (\mathit{get\text{-}trail}\ T) = \mathit{lits\text{-}of\text{-}l}\ (\mathit{get\text{-}trail\text{-}l}\ S)$› **and**
‹$\mathit{count\text{-}decided}\ (\mathit{get\text{-}trail}\ T) = \mathit{count\text{-}decided}\ (\mathit{get\text{-}trail\text{-}l}\ S)$› **and**
‹$\mathit{get\text{-}trail}\ T = [] \longleftrightarrow \mathit{get\text{-}trail\text{-}l}\ S = []$› **and**
‹$\mathit{get\text{-}trail}\ T \neq [] \longleftrightarrow \mathit{get\text{-}trail\text{-}l}\ S \neq []$› **and**
‹$\mathit{get\text{-}trail}\ T \neq [] \Longrightarrow \mathit{is\text{-}proped}\ (\mathit{hd}\ (\mathit{get\text{-}trail}\ T)) \longleftrightarrow \mathit{is\text{-}proped}\ (\mathit{hd}\ (\mathit{get\text{-}trail\text{-}l}\ S))$›
‹$\mathit{get\text{-}trail}\ T \neq [] \Longrightarrow \mathit{is\text{-}decided}\ (\mathit{hd}\ (\mathit{get\text{-}trail}\ T)) \longleftrightarrow \mathit{is\text{-}decided}\ (\mathit{hd}\ (\mathit{get\text{-}trail\text{-}l}\ S))$›
‹$\mathit{get\text{-}trail}\ T \neq [] \Longrightarrow \mathit{lit\text{-}of}\ (\mathit{hd}\ (\mathit{get\text{-}trail}\ T)) = \mathit{lit\text{-}of}\ (\mathit{hd}\ (\mathit{get\text{-}trail\text{-}l}\ S))$›
‹$\mathit{get\text{-}level}\ (\mathit{get\text{-}trail}\ T) = \mathit{get\text{-}level}\ (\mathit{get\text{-}trail\text{-}l}\ S)$›
‹$\mathit{get\text{-}maximum\text{-}level}\ (\mathit{get\text{-}trail}\ T) = \mathit{get\text{-}maximum\text{-}level}\ (\mathit{get\text{-}trail\text{-}l}\ S)$›
‹$\mathit{get\text{-}trail}\ T \models\mathit{as}\ D \longleftrightarrow \mathit{get\text{-}trail\text{-}l}\ S \models\mathit{as}\ D$›
‹$\mathit{subsumed\text{-}init\text{-}clauses}\ T = \mathit{get\text{-}subsumed\text{-}init\text{-}clauses\text{-}l}\ S$›
‹$\mathit{subsumed\text{-}learned\text{-}clauses}\ T = \mathit{get\text{-}subsumed\text{-}learned\text{-}clauses\text{-}l}\ S$›
‹$\mathit{subsumed\text{-}clauses}\ T = \mathit{get\text{-}subsumed\text{-}clauses\text{-}l}\ S$›
‹$\mathit{get\text{-}all\text{-}clss}\ T = \mathit{get\text{-}all\text{-}clss\text{-}l}\ S$›
‹$\mathit{all\text{-}lits\text{-}of\text{-}st}\ T = \mathit{all\text{-}lits\text{-}of\text{-}st\text{-}l}\ S$›
using *assms* **unfolding** *twl-st-l-def all-clss-lf-ran-m*[*symmetric*]
apply (*auto split*: *option.splits simp*: *clauses-def mset-take-mset-drop-mset′*
all-lits-of-st-def all-lits-of-st-l-def; *fail*)+
using *assms* **unfolding** *twl-st-l-def all-clss-lf-ran-m*[*symmetric*]
all-lits-of-st-def all-lits-of-st-l-def image-mset-union
by (*auto split*: *option.splits simp*: *clauses-def mset-take-mset-drop-mset′*
all-lits-of-st-def all-lits-of-st-l-def; *auto simp*: *ac-simps*; *fail*)

lemma (**in** −) [*twl-st-l*]:
‹(*S*, *T*)∈*twl-st-l b* ⟹
get-all-init-clss T = *mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses-l S* +
subsumed-init-clauses T + *get-init-clauses0 T*›
‹(*S*, *T*)∈*twl-st-l b* ⟹ *get-all-learned-clss T* = *mset* ‘# *learned-clss-lf* (*get-clauses-l S*) +
get-unit-learned-clss-l S + *get-subsumed-learned-clauses-l S* + *get-learned-clauses0-l S*›
by (*cases S*; *cases T*; *cases b*; *auto simp*: *twl-st-l-def mset-take-mset-drop-mset*′; *fail*)+

lemma [*twl-st-l*]:
assumes ‹(*S*, *T*) ∈ *twl-st-l L*›
shows ‹*lit-of* ‘ *set* (*get-trail T*) = *lit-of* ‘ *set* (*get-trail-l S*)›
using *twl-st-l*[*OF assms*] **unfolding** *lits-of-def*
by *simp*

lemma [*twl-st-l*]:
‹*get-trail-l* (*set-literals-to-update-l D S*) = *get-trail-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *defined-lit* (*get-trail T*) = *defined-lit* (*get-trail-l S*)›
by (*cases S*; *cases L*) (*auto simp*: *twl-st-l-def*)

fun *remove-one-lit-from-wq* :: ‹*nat* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l*› **where**
‹*remove-one-lit-from-wq L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *remove1-mset L WS*, *Q*)›

lemma [*twl-st-l*]: ‹*get-conflict-l* (*set-clauses-to-update-l W S*) = *get-conflict-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*get-conflict-l* (*remove-one-lit-from-wq L S*) = *get-conflict-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*literals-to-update-l* (*set-clauses-to-update-l Cs S*) = *literals-to-update-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*get-unit-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-clauses-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*get-unit-clauses-l* (*remove-one-lit-from-wq L S*) = *get-unit-clauses-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]:
‹*get-unit-init-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-init-clauses-l S*›
by (*cases S*; *auto*; *fail*)+

lemma [*twl-st-l*]:
‹*get-unit-init-clauses-l* (*remove-one-lit-from-wq L S*) = *get-unit-init-clauses-l S*›
by (*cases S*; *auto*; *fail*)+

lemma [*twl-st-l*]:
‹*get-clauses-l* (*remove-one-lit-from-wq L S*) = *get-clauses-l S*›
‹*get-trail-l* (*remove-one-lit-from-wq L S*) = *get-trail-l S*›
by (*cases S*; *auto*; *fail*)+

lemma [*twl-st-l*]:

‹*get-unit-learned-clss-l* (*set-clauses-to-update-l Cs S*) = *get-unit-learned-clss-l S*›
by (*cases S*) *auto*

lemma [*twl-st-l*]:
‹*get-unit-learned-clss-l* (*remove-one-lit-from-wq L S*) = *get-unit-learned-clss-l S*›
by (*cases S*) *auto*

lemma *literals-to-update-l-remove-one-lit-from-wq*[*simp*]:
‹*literals-to-update-l* (*remove-one-lit-from-wq L T*) = *literals-to-update-l T*›
by (*cases T*) *auto*

lemma *clauses-to-update-l-remove-one-lit-from-wq*[*simp*]:
‹*clauses-to-update-l* (*remove-one-lit-from-wq L T*) = *remove1-mset L* (*clauses-to-update-l T*)›
by (*cases T*) *auto*

lemma *get-clauses0*[*twl-st-l*]:
‹*get-all-clauses0* (*set-clauses-to-update WS S*) = *get-all-clauses0 S*›
‹*get-init-clauses0* (*set-clauses-to-update WS S*) = *get-init-clauses0 S*›
‹*get-learned-clauses0* (*set-clauses-to-update WS S*) = *get-learned-clauses0 S*›
by (*cases S*; *auto*; *fail*)+

lemma *get-clauses0-l*[*twl-st-l*]:
‹*get-clauses0-l* (*set-clauses-to-update-l WS S*) = *get-clauses0-l S*›
‹*get-init-clauses0-l* (*set-clauses-to-update-l WS S*) = *get-init-clauses0-l S*›
‹*get-learned-clauses0-l* (*set-clauses-to-update-l WS S*) = *get-learned-clauses0-l S*›
‹*get-clauses0-l* (*set-literals-to-update-l WS′ S*) = *get-clauses0-l S*›
‹*get-init-clauses0-l* (*set-literals-to-update-l WS′ S*) = *get-init-clauses0-l S*›
‹*get-learned-clauses0-l* (*set-literals-to-update-l WS′ S*) = *get-learned-clauses0-l S*›
‹*get-clauses0-l* (*remove-one-lit-from-wq L S*) = *get-clauses0-l S*›
‹*get-init-clauses0-l* (*remove-one-lit-from-wq L S*) = *get-init-clauses0-l S*›
‹*get-learned-clauses0-l* (*remove-one-lit-from-wq L S*) = *get-learned-clauses0-l S*›
by (*cases S*; *auto*; *fail*)+

declare *twl-st-l*[*simp*]

lemma *unit-init-clauses-get-unit-init-clauses-l*[*twl-st-l*]:
‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *unit-init-clauses T* = *get-unit-init-clauses-l S*› **and**
get-init-learned-clss-get-init-learned-clss-l[*twl-st-l*]:
‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *get-init-learned-clss T* = *get-unit-learned-clss-l S*›
by (*cases S*) (*auto simp*: *twl-st-l-def init-clss.simps*)

lemma *get-clauses0-l-get-clauses0*[*twl-st*,*simp*]:
‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *get-all-clauses0 T* = *get-clauses0-l S*›
‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *get-init-clauses0 T* = *get-init-clauses0-l S*›
‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *get-learned-clauses0 T* = *get-learned-clauses0-l S*›
by (*cases S*) (*auto simp*: *twl-st-l-def init-clss.simps*)

lemma *clauses-state-to-l*[*twl-st-l*]: ‹(*S*, *S′*) ∈ *twl-st-l L* ⟹
cdcl$_W$*-restart-mset.clauses* (*state*$_W$*-of S′*) = *mset* ‘# *ran-mf* (*get-clauses-l S*) +
get-unit-init-clauses-l S + *get-unit-learned-clss-l S* + *get-subsumed-init-clauses-l S* +
get-subsumed-learned-clauses-l S + *get-init-clauses0-l S* + *get-learned-clauses0-l S*›
apply (*subst all-clss-l-ran-m*[*symmetric*])
unfolding *image-mset-union*
by (*cases S*) (*auto simp*: *twl-st-l-def init-clss.simps mset-take-mset-drop-mset′ clauses-def*)

lemma *clauses-to-update-l-set-clauses-to-update-l*[*twl-st-l*]:

‹*clauses-to-update-l* (*set-clauses-to-update-l WS S*) = *WS*›
by (*cases S*) *auto*

lemma *hd-get-trail-twl-st-of-get-trail-l*:
‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *get-trail-l S* ≠ [] ⟹
lit-of (*hd* (*get-trail T*)) = *lit-of* (*hd* (*get-trail-l S*))›
by (*cases S*; *cases* ‹*get-trail-l S*›; *cases* ‹*get-trail T*›) (*auto simp*: *twl-st-l-def*)

lemma *twl-st-l-mark-of-hd*:
‹(*x*, *y*) ∈ *twl-st-l b* ⟹
get-trail-l x ≠ [] ⟹
is-proped (*hd* (*get-trail-l x*)) ⟹
mark-of (*hd* (*get-trail-l x*)) > *0* ⟹
mark-of (*hd* (*get-trail y*)) = *mset* (*get-clauses-l x* ∝ *mark-of* (*hd* (*get-trail-l x*)))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;
cases ‹*hd* (*get-trail y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma *twl-st-l-lits-of-tl*:
‹(*x*, *y*) ∈ *twl-st-l b* ⟹
lits-of-l (*tl* (*get-trail y*)) = (*lits-of-l* (*tl* (*get-trail-l x*)))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;
cases ‹*hd* (*get-trail y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma *twl-st-l-mark-of-is-decided*:
‹(*x*, *y*) ∈ *twl-st-l b* ⟹
get-trail-l x ≠ [] ⟹
is-decided (*hd* (*get-trail y*)) = *is-decided* (*hd* (*get-trail-l x*))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;
cases ‹*hd* (*get-trail y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma *twl-st-l-mark-of-is-proped*:
‹(*x*, *y*) ∈ *twl-st-l b* ⟹
get-trail-l x ≠ [] ⟹
is-proped (*hd* (*get-trail y*)) = *is-proped* (*hd* (*get-trail-l x*))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;
cases ‹*hd* (*get-trail y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma [*simp*]:
‹*get-clauses-l* (*set-literals-to-update-l L T*) = *get-clauses-l T*›
‹*get-unit-clauses-l* (*set-literals-to-update-l L T*) = *get-unit-clauses-l T*›
‹*get-subsumed-clauses-l* (*set-literals-to-update-l L T*) = *get-subsumed-clauses-l T*›
by (*cases T*; *auto*; *fail*)+

fun *equality-except-trail* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
‹*equality-except-trail* (*M*, *N*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*) (*M'*, *N'*, *D'*, *NE'*, *UE'*, *NS'*, *US'*, *WS'*, *Q'*)
⟷
N = *N'* ∧ *D* = *D'* ∧ *NE* = *NE'* ∧ *UE* = *UE'* ∧ *NS* = *NS'* ∧ *US* = *US'* ∧ *WS* = *WS'* ∧ *Q* = *Q'*›

fun *equality-except-conflict-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
‹*equality-except-conflict-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*) (*M'*, *N'*, *D'*, *NE'*, *UE'*, *NEk'*,
UEk', *NS'*, *US'*, *WS'*, *Q'*) ⟷

$M = M' \wedge N = N' \wedge NE = NE' \wedge UE = UE' \wedge NEk = NEk' \wedge UEk = UEk' \wedge NS = NS' \wedge US = US' \wedge WS = WS' \wedge Q = Q'$⟩

lemma *equality-except-conflict-l-rewrite*:
assumes ‹*equality-except-conflict-l S T*›
shows
‹*get-trail-l S = get-trail-l T*› **and**
‹*get-clauses-l S = get-clauses-l T*›
using *assms* **by** (*cases S*; *cases T*; *auto*; *fail*)+

lemma *equality-except-conflict-l-alt-def*:
‹*equality-except-conflict-l S T* ⟷
get-trail-l S = get-trail-l T ∧ *get-clauses-l S = get-clauses-l T* ∧
get-kept-init-clauses-l S = get-kept-init-clauses-l T ∧
get-kept-learned-clss-l S = get-kept-learned-clss-l T ∧
get-unkept-init-clauses-l S = get-unkept-init-clauses-l T ∧
get-unkept-learned-clss-l S = get-unkept-learned-clss-l T ∧
get-unit-learned-clss-l S = get-unit-learned-clss-l T ∧
literals-to-update-l S = literals-to-update-l T ∧
clauses-to-update-l S = clauses-to-update-l T ∧
get-subsumed-learned-clauses-l S = get-subsumed-learned-clauses-l T ∧
get-subsumed-init-clauses-l S = get-subsumed-init-clauses-l T ∧
get-init-clauses0-l S = get-init-clauses0-l T ∧
get-learned-clauses0-l S = get-learned-clauses0-l T›
by (*cases S*, *cases T*) *auto*

lemma *equality-except-conflict-alt-def*:
‹*equality-except-conflict S T* ⟷
get-trail S = get-trail T ∧ *get-init-clauses S = get-init-clauses T* ∧
get-learned-clss S = get-learned-clss T ∧
get-init-learned-clss S = get-init-learned-clss T ∧
unit-init-clauses S = unit-init-clauses T ∧
literals-to-update S = literals-to-update T ∧
clauses-to-update S = clauses-to-update T ∧
subsumed-learned-clauses S = subsumed-learned-clauses T ∧
subsumed-init-clauses S = subsumed-init-clauses T ∧
get-init-clauses0 S = get-init-clauses0 T ∧
get-learned-clauses0 S = get-learned-clauses0 T›
by (*cases S*, *cases T*) *auto*

lemma *all-lits-of-st-simps*[*simp*]:
‹*all-lits-of-st* (*set-clauses-to-update C S*)= *all-lits-of-st S*›
by (*cases S*; *auto simp*: *all-lits-of-st-def*)

lemma *all-lits-of-st-l-simps*[*simp*]:
‹*all-lits-of-st-l* (*set-clauses-to-update-l C T*)= *all-lits-of-st-l T*›
‹*all-lits-of-st-l* (*set-literals-to-update-l C' T*)= *all-lits-of-st-l T*›
‹*all-lits-of-st-l* (*remove-one-lit-from-wq n T*)= *all-lits-of-st-l T*›
‹−*L* ∈# *all-lits-of-st-l T* ⟷ *L* ∈# *all-lits-of-st-l T*›
by (*cases T*; *auto simp*: *all-lits-of-st-l-def in-all-lits-of-mm-ain-atms-of-iff*; *fail*)+

5.2 Additional Invariants and Definitions

definition *twl-list-invs* **where**
‹*twl-list-invs S* ⟷

(∀ C ∈# clauses-to-update-l S. C ∈# dom-m (get-clauses-l S)) ∧
0 ∉# dom-m (get-clauses-l S) ∧
(∀ L C. Propagated L C ∈ set (get-trail-l S) ⟶ (C > 0 ⟶ C ∈# dom-m (get-clauses-l S) ∧
(C > 0 ⟶ L ∈ set (watched-l (get-clauses-l S ∝ C)) ∧
(length (get-clauses-l S ∝ C) > 2 ⟶ L = get-clauses-l S ∝ C ! 0)))) ∧
distinct-mset (clauses-to-update-l S) ∧
(∀ C∈#ran-mf (get-clauses-l S). ¬tautology (mset C))›

definition *polarity* **where**
‹polarity M L =
(if undefined-lit M L then None else if L ∈ lits-of-l M then Some True else Some False)›

lemma *polarity-None-undefined-lit*: ‹is-None (polarity M L) ⟹ undefined-lit M L›
by (*auto simp*: *polarity-def split*: *if-splits*)

lemma *polarity-spec*:
assumes ‹no-dup M›
shows
‹RETURN (polarity M L) ≤ SPEC(λv. (v = None ⟷ undefined-lit M L) ∧
(v = Some True ⟷ L ∈ lits-of-l M) ∧ (v = Some False ⟷ −L ∈ lits-of-l M))›
unfolding *polarity-def*
by *refine-vcg*
(*use assms* **in** ‹*auto simp*: *defined-lit-map lits-of-def atm-of-eq-atm-of uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus
split: *option.splits*›)

lemma *polarity-spec′*:
assumes ‹no-dup M›
shows
‹polarity M L = None ⟷ undefined-lit M L› **and**
‹polarity M L = Some True ⟷ L ∈ lits-of-l M› **and**
‹polarity M L = Some False ⟷ −L ∈ lits-of-l M›
unfolding *polarity-def*
by (*use assms* **in** ‹*auto simp*: *defined-lit-map lits-of-def atm-of-eq-atm-of uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus
split: *option.splits*›)

definition *mop-polarity-l* :: ‹'v twl-st-l ⇒ 'v literal ⇒ bool option nres› **where**
‹mop-polarity-l S L = do {
ASSERT(L ∈# all-lits-of-st-l S);
ASSERT(no-dup (get-trail-l S));
RETURN(polarity (get-trail-l S) L)
}›

definition *find-unwatched-l* :: ‹('v, -) ann-lits ⇒ - ⇒ nat ⇒ nat option nres› **where**
‹find-unwatched-l M N C = do {
ASSERT(C ∈# dom-m N ∧ length (N ∝ C) ≥ 2 ∧ distinct (N ∝ C) ∧ no-dup M);
SPEC (λ(found).
(found = None ⟷ (∀ L∈set (unwatched-l (N ∝ C)). −L ∈ lits-of-l M)) ∧
(∀ j. found = Some j ⟶ (j < length (N ∝ C) ∧ (undefined-lit M (N ∝ C!j) ∨ N ∝ C!j ∈ lits-of-l
M) ∧ j ≥ 2)))
}›

definition *set-conflict-l-pre* :: ‹nat ⇒ 'v twl-st-l ⇒ bool› **where**
‹set-conflict-l-pre C S ⟷
get-conflict-l S = None ∧ C ∈# dom-m (get-clauses-l S) ∧ ¬tautology (mset (get-clauses-l S ∝ C))

$\wedge$ *distinct* (*get-clauses-l S* $\propto$ *C*) $\wedge$
get-trail-l S $\models$*as CNot* (*mset* (*get-clauses-l S* $\propto$ *C*)) $\wedge$ *twl-list-invs S* $\wedge$
($\exists$ *S'* *b*. (*set-clauses-to-update-l* (*clauses-to-update-l S* + {#*C*#}) *S*, *S'*) $\in$ *twl-st-l b* $\wedge$ *twl-struct-invs S'* $\wedge$ *twl-stgy-invs S'*)›

definition *set-conflict-l* :: ‹*nat* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *'v twl-st-l nres*› **where**
‹*set-conflict-l* = (*λC* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*set-conflict-l-pre C* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
RETURN (*M*, *N*, *Some* (*mset* (*N* $\propto$ *C*)), *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})
})›

definition *cons-trail-propagate-l* **where**
‹*cons-trail-propagate-l L C M* = *do* {
ASSERT(*undefined-lit M L*);
RETURN (*Propagated L C* # *M*)
}›

definition *propagate-lit-l* :: ‹*'v literal* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *'v twl-st-l nres*› **where**
‹*propagate-lit-l* = (*λL' C i* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*C* $\in$# *dom-m N*);
ASSERT(*L'* $\in$# *all-lits-of-st-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
ASSERT(*i* $\leq$ *1*);
M $\leftarrow$ *cons-trail-propagate-l L' C M*;
N $\leftarrow$ (*if length* (*N* $\propto$ *C*) > *2 then mop-clauses-swap N C 0* (*Suc 0* $-$ *i*) *else RETURN N*);
RETURN (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* ($-$*L'*) *Q*)})›

definition *update-clause-l-pre* :: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *bool*› **where**
‹*update-clause-l-pre C i f S* $\longleftrightarrow$
($\exists$ *S' L*. (*S*, *S'*) $\in$ *twl-st-l* (*Some L*) $\wedge$ *C* $\in$# *dom-m* (*get-clauses-l S*) $\wedge$
i < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$ *f* < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
update-clauseS-pre (*get-clauses-l S* $\propto$ *C* ! *i*) (*twl-clause-of* (*get-clauses-l S* $\propto$ *C*)) *S'*)›

definition *update-clause-l* :: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *'v twl-st-l nres*› **where**
‹*update-clause-l* = (*λC i f* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*). *do* {
ASSERT(*update-clause-l-pre C i f* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*));
N' $\leftarrow$ *mop-clauses-swap N C i f*;
RETURN (*M*, *N'*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*)
})›

definition *unit-propagation-inner-loop-body-l-inv*
:: ‹*'v literal* $\Rightarrow$ *nat* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *bool*›
where
‹*unit-propagation-inner-loop-body-l-inv L C S* $\longleftrightarrow$
($\exists$ *S'*. (*set-clauses-to-update-l* (*clauses-to-update-l S* + {#*C*#}) *S*, *S'*) $\in$ *twl-st-l* (*Some L*) $\wedge$
twl-struct-invs S' $\wedge$
twl-stgy-invs S' $\wedge$
C $\in$# *dom-m* (*get-clauses-l S*) $\wedge$
C > *0* $\wedge$
0 < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
no-dup (*get-trail-l S*) $\wedge$
(*if* (*get-clauses-l S* $\propto$ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
1 $-$ (*if* (*get-clauses-l S* $\propto$ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
L $\in$ *set* (*watched-l* (*get-clauses-l S* $\propto$ *C*)) $\wedge$
get-conflict-l S = *None*
)

```
  ›

definition pos-of-watched :: ‹'v clauses-l ⇒ nat ⇒ 'v literal ⇒ nat nres› where
  ‹pos-of-watched N C L = do {
     ASSERT(length (N ∝ C) > 0 ∧ C ∈# dom-m N);
     RETURN (if (N ∝ C) ! 0 = L then 0 else 1)
  }›

definition other-watched-l :: ‹'v twl-st-l ⇒ 'v literal ⇒ nat ⇒ nat ⇒ 'v literal nres› where
  ‹other-watched-l S L C i = do {
    ASSERT(get-clauses-l S ∝ C ! i = L ∧ i < length (get-clauses-l S ∝ C) ∧ i < 2 ∧
      C ∈# dom-m (get-clauses-l S) ∧ 1−i < length (get-clauses-l S ∝ C));
    mop-clauses-at (get-clauses-l S) C (1 − i)
  }›

definition unit-propagation-inner-loop-body-l :: ‹'v literal ⇒ nat ⇒
  'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹unit-propagation-inner-loop-body-l L C S = do {
      ASSERT(unit-propagation-inner-loop-body-l-inv L C S);
      K ← SPEC(λK. K ∈ set (get-clauses-l S ∝ C));
      val-K ← mop-polarity-l S K;
      if val-K = Some True
      then RETURN S
      else do {
        i ← pos-of-watched (get-clauses-l S) C L;
        L' ← other-watched-l S L C i;
        val-L' ← mop-polarity-l S L';
        if val-L' = Some True
        then RETURN S
        else do {
            f ← find-unwatched-l (get-trail-l S) (get-clauses-l S) C;
            case f of
              None ⇒
                if val-L' = Some False
                then set-conflict-l C S
                else propagate-lit-l L' C i S
            | Some f ⇒ do {
                ASSERT(f < length (get-clauses-l S ∝ C));
                K ← mop-clauses-at (get-clauses-l S) C f;
                val-K ← mop-polarity-l S K;
                if val-K = Some True then
                  RETURN S
                else
                  update-clause-l C i f S
              }
          }
      }
   }›

lemma refine-add-invariants:
  assumes
    ‹(f S) ≤ SPEC(λS'. Q S')› and
    ‹y ≤ ⇓ {(S, S'). P S S'} (f S)›
  shows ‹y ≤ ⇓ {(S, S'). P S S' ∧ Q S'} (f S)›
  using assms unfolding pw-le-iff pw-conc-inres pw-conc-nofail by force
```

lemma *clauses-tuple*[*simp*]:
‹*cdcl*$_W$*-restart-mset.clauses* (*M*, {#*f x* . *x* ∈# *init-clss-l N*#} + *NE*,
{#*f x* . *x* ∈# *learned-clss-l N*#} + *UE*, *D*) = {#*f x*. *x* ∈# *all-clss-l N*#} + *NE* + *UE*›
by (*auto simp*: *clauses-def simp flip*: *image-mset-union*)

lemma *valid-enqueued-alt-simps*[*simp*]:
‹*valid-enqueued S* ⟷
(∀ (*L*, *C*) ∈# *clauses-to-update S*. *L* ∈# *watched C* ∧ *C* ∈# *get-clauses S* ∧
−*L* ∈ *lits-of-l* (*get-trail S*) ∧ *get-level* (*get-trail S*) *L* = *count-decided* (*get-trail S*)) ∧
(∀ *L* ∈# *literals-to-update S*.
−*L* ∈ *lits-of-l* (*get-trail S*) ∧ *get-level* (*get-trail S*) *L* = *count-decided* (*get-trail S*))›
by (*cases S*) *auto*

declare *valid-enqueued.simps*[*simp del*]

lemma *unit-propagation-inner-loop-body-alt-def*:
‹*unit-propagation-inner-loop-body L C S* = *do* {
bL′ ← *SPEC* (λ*K*. *K* ∈# *clause C*);
val-bL′ ← *mop-lit-is-pos bL*′ *S*;
if val-bL′
then RETURN S
else do {
i ← *RETURN* ();
L′ ← *SPEC* (λ*K*. *K* ∈# *watched C* − {#*L*#});
ASSERT (*watched C* = {#*L*, *L*′#});
val-L′ ← *mop-lit-is-pos L*′ *S*;
if val-L′
then RETURN S
else
if ∀ *L* ∈# *unwatched C*. −*L* ∈ *lits-of-l* (*get-trail S*)
then
if −*L*′ ∈ *lits-of-l* (*get-trail S*)
then do {*mop-set-conflicting C S*}
else do {*mop-propagate-lit L*′ *C S*}
else do {
update-clauseS L C S
}
}
}›
unfolding *unit-propagation-inner-loop-body-def bind-to-let-conv Let-def*
by (*auto intro*!: *ext*)

lemma [*twl-st*, *simp*]:
‹*get-clauses* (*set-clauses-to-update C S*′) = *get-clauses S*′›
‹*unit-clss* (*set-clauses-to-update C S*′) = *unit-clss S*′›
‹(*S*, *S*′) ∈ *twl-st-l* (*Some L*) ⟹
clauses (*get-clauses S*′) = {#*mset* (*fst x*). *x* ∈# *ran-m* (*get-clauses-l S*)#}›
apply (*cases S*′; *auto simp*: *twl-st-l-def clauses-def*)
apply (*cases S*′; *auto simp*: *twl-st-l-def clauses-def*)
apply (*cases S*′; *auto simp*: *twl-st-l-def clauses-def mset-take-mset-drop-mset*′
simp flip: *image-mset-union*)
done

lemma *in-set-takeI*:
‹*i* < *j* ⟹ *i* < *length xs* ⟹ *xs* ! *i* ∈ *set* (*take j xs*)›
by (*auto simp*: *in-set-take-conv-nth intro*!: *exI*[*of* - *i*])

lemma *unit-propagation-inner-loop-body-l*:
  fixes *i C* :: *nat* **and** *S* :: ‹*'v twl-st-l*› **and** *S'* :: ‹*'v twl-st*› **and** *L* :: ‹*'v literal*›
  defines
    C'[*simp*]: ‹*C'* ≡ *get-clauses-l S* ∝ *C*›
  assumes
    SS': ‹(*S*, *S'*) ∈ *twl-st-l* (*Some L*)› **and**
    WS: ‹*C* ∈# *clauses-to-update-l S*› **and**
    struct-invs: ‹*twl-struct-invs S'*› **and**
    add-inv: ‹*twl-list-invs S*› **and**
    stgy-inv: ‹*twl-stgy-invs S'*›
  shows
    ‹*unit-propagation-inner-loop-body-l L C*
       (*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*) ≤
       ⇓ {(*S*, *S'*). ((*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*) ∧
          (*twl-stgy-invs S'* ∧ *twl-struct-invs S'*)}
         (*unit-propagation-inner-loop-body L* (*twl-clause-of C'*)
           (*set-clauses-to-update* (*clauses-to-update* (*S'*) − {#(*L*, *twl-clause-of C'*)#}) *S'*))›
    (**is** ‹*?A* ≤ ⇓ - *?B*›)
proof −

  have *C-0*: ‹*C* > *0*› **and** *C-neq-0*[*iff*]: ‹*C* ≠ *0*›
    using *assms*(*3*,*5*) **unfolding** *twl-list-invs-def* **by** (*auto dest*!: *multi-member-split*)

  have *C-N-U*: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
    using *add-inv WS SS'* **by** (*auto simp*: *twl-list-invs-def*)

  let *?S* = ‹*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*›
  define *i* :: *nat* **where** ‹*i* ≡ (*if get-clauses-l S*∝*C*!*0* = *L then 0 else 1*)›
  let *?L* = ‹*C'* ! *i*›
  let *?L'* = ‹*C'* ! (*Suc 0* − *i*)›
  have *inv*: ‹*twl-st-inv S'*› **and**
    cdcl-inv: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of S'*)› **and**
    valid: ‹*valid-enqueued S'*›
    using *struct-invs WS* **by** (*auto simp*: *twl-struct-invs-def pcdcl-all-struct-invs-def*)
  have
    w-q-inv: ‹*clauses-to-update-inv S'*› **and**
    dist: ‹*distinct-queued S'*› **and**
    no-dup: ‹*no-duplicate-queued S'*› **and**
    confl: ‹*get-conflict S'* ≠ *None* ⟹ *clauses-to-update S'* = {#} ∧ *literals-to-update S'* = {#}› **and**
    st-inv: ‹*twl-st-inv S'*›
    using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*+
  have *dist-C*: ‹*distinct* (*get-clauses-l S* ∝ *C*)›
    using *cdcl-inv SS' C-N-U* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
      cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-alt-def pcdcl-all-struct-invs-def*
    by (*auto simp*: *ran-m-def dest*!: *multi-member-split*)

  let *?M* = ‹*get-trail-l S*›
  let *?N* = ‹*get-clauses-l S*›
  let *?WS* = ‹*clauses-to-update-l S*›
  let *?Q* = ‹*literals-to-update-l S*›

  have *n-d*: ‹*no-dup ?M*› **and** *confl-inv*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting* (*state*$_W$*-of S'*)›
    using *cdcl-inv SS'* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
      cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*

by (*auto simp*: *trail.simps comp-def twl-st*)

then have *consistent*: ‹− *L* ∉ *lits-of-l ?M*› **if** ‹*L* ∈ *lits-of-l ?M*› **for** *L*
using *consistent-interp-def distinct-consistent-interp that* **by** *blast*

have *cons-M*: ‹*consistent-interp* (*lits-of-l ?M*)›
using *n-d distinct-consistent-interp* **by** *fast*
let *?C′* = ‹*twl-clause-of C′*›
have *C′-N-U-or*: ‹*?C′* ∈# *twl-clause-of* ‘# (*init-clss-lf ?N*) ∨
?C′ ∈# *twl-clause-of* ‘# *learned-clss-lf ?N*›
using *WS valid SS′*
unfolding *union-iff*[*symmetric*] *image-mset-union*[*symmetric*] *mset-append*[*symmetric*]
by (*auto simp*: *twl-struct-invs-def*
split: *prod.splits simp del*: *twl-clause-of.simps*)
have *struct*: ‹*struct-wf-twl-cls ?C′*›
using *C-N-U inv SS′ WS valid* **unfolding** *valid-enqueued-alt-simps*
by (*auto simp*: *twl-st-inv-alt-def Ball-ran-m-dom-struct-wf*
simp del: *twl-clause-of.simps*)
have *C′-N-U*: ‹*?C′* ∈# *twl-clause-of* ‘# *all-clss-lf ?N*›
using *C′-N-U-or*
unfolding *union-iff*[*symmetric*] *image-mset-union*[*symmetric*] *mset-append*[*symmetric*] **.**
have *watched-C′*: ‹*mset* (*watched-l C′*) = {#*?L*, *?L′*#}›
using *struct i-def SS′* **by** (*cases C*) (*auto simp*: *length-list-2 take-2-if*)
then have *mset-watched-C*: ‹*mset* (*watched-l C′*) = {#*watched-l C′* ! *i*, *watched-l C′* ! (*Suc 0* − *i*)#}›
using *i-def* **by** (*cases* ‹*twl-clause-of* (*get-clauses-l S* ∝ *C*)›) (*auto simp*: *take-2-if*)
have *two-le-length-C*: ‹*2* ≤ *length C′*›
by (*metis length-take linorder-not-le min-less-iff-conj numeral-2-eq-2 order-less-irrefl*
size-add-mset size-eq-0-iff-empty size-mset watched-C′)
then have *i-le-length-C*: ‹*i* < *length C′*›
by (*auto simp*: *i-def*)
obtain *WS′* **where** *WS′-def*: ‹*?WS* = *add-mset C WS′*›
using *multi-member-split*[*OF WS*] **by** *auto*
then have *WS′-def′*: ‹*?WS* = *add-mset C WS′*›
by *auto*
have *L*: ‹*L* ∈ *set* (*watched-l C′*)› **and** *uL-M*: ‹−*L* ∈ *lits-of-l* (*get-trail-l S*)›
using *valid SS′* **by** (*auto simp*: *WS′-def*)
have *C′-i*[*simp*]: ‹*C′*!*i* = *L*›
using *L two-le-length-C* **by** (*auto simp*: *take-2-if i-def split*: *if-splits*)
then have [*simp*]: ‹*?N*∝*C*!*i* = *L*›
by *auto*
have *C-neq-0*[*iff*]: ‹*C* ≠ *0*›
using *assms*(*3*,*5*) **unfolding** *twl-list-invs-def* **by** (*auto dest*!: *multi-member-split*)
then have *C-0*: ‹*C* > *0*›
by *linarith*
have *pre-inv*: ‹*unit-propagation-inner-loop-body-l-inv L C ?S*›
unfolding *unit-propagation-inner-loop-body-l-inv-def*
proof (*rule exI*[*of - S′*], *intro conjI*)
have *S-readd-C-S*: ‹*set-clauses-to-update-l* (*clauses-to-update-l ?S* + {#*C*#}) *?S* = *S*›
using *WS* **by** (*cases S*) *auto*
show ‹(*set-clauses-to-update-l*
(*clauses-to-update-l ?S* + {#*C*#})
(*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*),
S′) ∈ *twl-st-l* (*Some L*)›
using *SS′* **unfolding** *S-readd-C-S* **.**
show ‹*twl-stgy-invs S′*› ‹*twl-struct-invs S′*›
using *assms* **by** *fast*+

```
  show ‹C ∈# dom-m (get-clauses-l ?S)›
    using assms C-N-U by auto
  show ‹C > 0›
    by (rule C-0)
  show ‹(if get-clauses-l ?S ∝ C ! 0 = L then 0 else 1) < length (get-clauses-l ?S ∝ C)›
    using two-le-length-C by auto
  show ‹1 − (if get-clauses-l ?S ∝ C ! 0 = L then 0 else 1) < length (get-clauses-l ?S ∝ C)›
    using two-le-length-C by auto
  show ‹length (get-clauses-l ?S ∝ C) > 0›
    using two-le-length-C by auto
  show ‹no-dup (get-trail-l ?S)›
    using n-d by auto
  show ‹L ∈ set (watched-l (get-clauses-l ?S ∝ C))›
    using L by auto
  show ‹get-conflict-l ?S = None›
    using confl SS' WS by (cases ‹get-conflict-l S›) (auto dest: in-diffD)
qed

define pol-spec where
  ‹pol-spec bL' = {(b, b'). (b = None ⟶ ¬b') ∧ (b = Some True ⟷ b') ∧
    (b' ⟷ bL'
        ∈ lits-of-l
         (get-trail
          (set-clauses-to-update
           (remove1-mset (L, twl-clause-of C')
            (clauses-to-update S'))
           S'))) ∧
    (b = Some False ⟷ −bL'
        ∈ lits-of-l
         (get-trail
          (set-clauses-to-update
           (remove1-mset (L, twl-clause-of C')
            (clauses-to-update S'))
           S')))}› for bL'

have [refine]: ‹mop-polarity-l
   (set-clauses-to-update-l
     (remove1-mset C (clauses-to-update-l S)) S)
   K
  ≤ ⇓(pol-spec L')
      (mop-lit-is-pos L'
        (set-clauses-to-update
          (remove1-mset (L, twl-clause-of C')
            (clauses-to-update S'))
          S'))›
  if ‹(K,L') ∈ Id› for K L'
  using that SS' unfolding mop-polarity-l-def mop-lit-is-pos-def
  by refine-rcg
    (auto simp: polarity-def mop-polarity-l-def twl-st mset-take-mset-drop-mset'
      Decided-Propagated-in-iff-in-lits-of-l pol-spec-def dest: no-dup-consistentD
    intro!: ASSERT-leI)

let ?S = ‹set-clauses-to-update-l (clauses-to-update-l S − {#C#}) S›
obtain M N D NE UE NEk UEk NS US N0 U0 WS Q where
  S: ‹S = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q)›
  by (cases S) auto
```

have *C-N-U*: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
  using *add-inv WS SS′* **by** (*auto simp*: *twl-list-invs-def*)
let *?M* = ‹*get-trail-l S*›
let *?N* = ‹*get-clauses-l S*›
let *?WS* = ‹*clauses-to-update-l S*›
let *?Q* = ‹*literals-to-update-l S*›

define *i* :: *nat* **where** ‹*i* ≡ (*if get-clauses-l S*∝*C*!*0* = *L then 0 else 1*)›
let *?L* = ‹*C′* ! *i*›
let *?L′* = ‹*C′* ! (*Suc 0* − *i*)›
have *inv*: ‹*twl-st-inv S′*› **and**
  cdcl-inv: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of S′*)› **and**
  valid: ‹*valid-enqueued S′*›
  using *struct-invs WS* **by** (*auto simp*: *twl-struct-invs-def pcdcl-all-struct-invs-def*)
have
  w-q-inv: ‹*clauses-to-update-inv S′*› **and**
  dist: ‹*distinct-queued S′*› **and**
  no-dup: ‹*no-duplicate-queued S′*› **and**
  confl: ‹*get-conflict S′* ≠ *None* ⟹ *clauses-to-update S′* = {#} ∧ *literals-to-update S′* = {#}› **and**
  st-inv: ‹*twl-st-inv S′*›
  using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*+
have *dist-C*: ‹*distinct* (*get-clauses-l S* ∝ *C*)›
  using *cdcl-inv SS′ C-N-U* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
    cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-alt-def*
  by (*auto simp*: *ran-m-def dest*!: *multi-member-split*)
have *n-d*: ‹*no-dup ?M*› **and** *confl-inv*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting* (*state*$_W$*-of S′*)›
  using *cdcl-inv SS′* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
    cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
  by (*auto simp*: *trail.simps comp-def twl-st*)

then have *consistent*: ‹− *L* ∉ *lits-of-l ?M*› **if** ‹*L* ∈ *lits-of-l ?M*› **for** *L*
  using *consistent-interp-def distinct-consistent-interp that* **by** *blast*

have *cons-M*: ‹*consistent-interp* (*lits-of-l ?M*)›
  using *n-d distinct-consistent-interp* **by** *fast*
let *?C′* = ‹*twl-clause-of C′*›
have *C′-N-U-or*: ‹*?C′* ∈# *twl-clause-of* ‘# (*init-clss-lf ?N*) ∨
    ?C′ ∈# *twl-clause-of* ‘# *learned-clss-lf ?N*›
  using *WS valid SS′*
  unfolding *union-iff*[*symmetric*] *image-mset-union*[*symmetric*] *mset-append*[*symmetric*]
  by (*auto simp*: *twl-struct-invs-def*
      split: *prod.splits simp del*: *twl-clause-of.simps*)
have *struct*: ‹*struct-wf-twl-cls ?C′*›
  using *C-N-U inv SS′ WS valid* **unfolding** *valid-enqueued-alt-simps*
  by (*auto simp*: *twl-st-inv-alt-def Ball-ran-m-dom-struct-wf*
    simp del: *twl-clause-of.simps*)
have *C′-N-U*: ‹*?C′* ∈# *twl-clause-of* ‘# *all-clss-lf ?N*›
  using *C′-N-U-or*
  unfolding *union-iff*[*symmetric*] *image-mset-union*[*symmetric*] *mset-append*[*symmetric*] **.**
have *watched-C′*: ‹*mset* (*watched-l C′*) = {#*?L*, *?L′*#}›
  using *struct i-def SS′* **by** (*cases C*) (*auto simp*: *length-list-2 take-2-if*)
then have *mset-watched-C*: ‹*mset* (*watched-l C′*) = {#*watched-l C′* ! *i*, *watched-l C′* ! (*Suc 0* − *i*)#}›
  using *i-def* **by** (*cases* ‹*twl-clause-of* (*get-clauses-l S* ∝ *C*)›) (*auto simp*: *take-2-if*)
have *two-le-length-C*: ‹*2* ≤ *length C′*›
  by (*metis length-take linorder-not-le min-less-iff-conj numeral-2-eq-2 order-less-irrefl*

size-add-mset size-eq-0-iff-empty size-mset watched-C′)
then have *i-le-length-C*: ‹*i* < *length C′*›
by (*auto simp*: *i-def*)
obtain *WS′* **where** *WS′-def*: ‹*?WS* = *add-mset C WS′*›
using *multi-member-split*[*OF WS*] **by** *auto*
then have *WS′-def′*: ‹*WS* = *add-mset C WS′*›
unfolding *S* **by** *auto*
have *L*: ‹*L* ∈ *set* (*watched-l C′*)› **and** *uL-M*: ‹−*L* ∈ *lits-of-l* (*get-trail-l S*)›
using *valid SS′* **by** (*auto simp*: *WS′-def*)
have *C′-i*[*simp*]: ‹*C′*!*i* = *L*›
using *L two-le-length-C* **by** (*auto simp*: *take-2-if i-def split*: *if-splits*)
then have [*simp*]: ‹*?N*∝*C*!*i* = *L*›
by *auto*

have *i-def′*: ‹*i* = (*if get-clauses-l ?S* ∝ *C* ! *0* = *L then 0 else 1*)›
unfolding *i-def* **by** *auto*
have ‹*twl-list-invs ?S*›
using *add-inv C-N-U* **unfolding** *twl-list-invs-def S*
by (*auto dest*: *in-diffD*)
then have *upd-rel*: ‹(*?S*,
set-clauses-to-update (*remove1-mset* (*L*, *twl-clause-of C′*) (*clauses-to-update S′*)) *S′*)
∈ {(*S*, *S′*). (*S*, *S′*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*}›
using *SS′ WS*
by (*auto simp*: *twl-st-l-def image-mset-remove1-mset-if*)
have *D-None*: ‹*get-conflict-l S* = *None*›
using *confl SS′* **by** (*cases* ‹*get-conflict-l S*›) (*auto simp*: *S WS′-def′*)
have [*simp*]: ‹*twl-st-inv* (*set-conflicting C′* (*set-clauses-to-update*
(*remove1-mset LC* (*clauses-to-update S′*)) *S′*))›
if ‹*twl-st-inv S′*› **for** *S′ C′ LC*
using *that*
by (*cases S′*) (*auto simp*: *twl-st-inv.simps set-conflicting-def*)

have *pre*: ‹*set-conflict-l-pre C* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *remove1-mset C WS*, *Q*)›
if ‹*M* ⊨*as CNot* (*mset* (*N* ∝ *C*))›
using *pre-inv C-N-U dist-C that assms*(*5*)
unfolding *set-conflict-l-pre-def unit-propagation-inner-loop-body-l-inv-def* **apply** −
apply *normalize-goal*+
apply (*intro conjI*)
apply (*auto simp*: *S true-annots-true-cls twl-list-invs-def dest*: *consistent-CNot-not-tautology distinct-consistent-interp dest*: *in-diffD*)[*6*]
apply (*rule-tac x*=*x* **in** *exI*, *rule-tac x* = ‹*Some L*› **in** *exI*)
apply (*simp add*: *S*)
done
have ‹*set-conflict-l C ?S* ≤ *SPEC twl-list-invs*› **if** ‹*M* ⊨*as CNot* (*mset* (*N* ∝ *C*))›
using *add-inv C-N-U D-None pre that* **unfolding** *twl-list-invs-def*
by (*auto dest*: *in-diffD simp*: *set-conflicting-def S*
set-conflict-l-def mset-take-mset-drop-mset′ intro!: *ASSERT-leI*)
then have *confl-rel*: ‹*set-conflict-l C ?S* ≤
⇓ {(*S*, *S′*).
(*S*, *S′*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*}
(*mop-set-conflicting* (*twl-clause-of C′*)
(*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of C′*)
(*clauses-to-update S′*))
S′))›

```
  using SS' WS D-None C-N-U pre
  by (auto simp: twl-st-l-def image-mset-remove1-mset-if set-conflicting-def S Let-def
    set-conflict-l-def mset-take-mset-drop-mset' mop-set-conflicting-def set-conflict-pre-def
    intro!: ASSERT-leI ASSERT-refine-right)

have propa-rel:
  ‹propagate-lit-l K C j
   (set-clauses-to-update-l
     (remove1-mset C (clauses-to-update-l S)) S)
  ≤ ⇓ {(S, S'). 
       (S, S') ∈ twl-st-l (Some L) ∧
       twl-list-invs S}
      (mop-propagate-lit L' (twl-clause-of C')
        (set-clauses-to-update
          (remove1-mset (L, twl-clause-of C')
            (clauses-to-update S'))
          S'))›
  if
    ‹(K, L') ∈ Id› and ‹L' ∈ {K. K ∈# remove1-mset L {#L, L'a#}}› and
    ‹watched (twl-clause-of C') = {#L, L'a#}›
   ‹(j, j') ∈ {(j, j'). j = i ∧ i < 2}›
  for L' K L'a j j'
  unfolding S clauses-to-update-l.simps set-clauses-to-update-l.simps
    propagate-lit-l-def mop-propagate-lit-def prod.case
proof refine-rcg
  assume ‹propagate-lit-pre L' (twl-clause-of C')
   (set-clauses-to-update
     (remove1-mset (L, twl-clause-of C') (clauses-to-update S'))
     S')›
  then have L'-undef: ‹- L' ∉ lits-of-l
     (get-trail
       (set-clauses-to-update
         (remove1-mset (L, twl-clause-of C') (clauses-to-update S')) S')) ›
      ‹L' ∉ lits-of-l
         (get-trail
           (set-clauses-to-update
             (remove1-mset (L, twl-clause-of C') (clauses-to-update S'))
             S'))›
    unfolding propagate-lit-pre-def Let-def
    by (auto simp: Decided-Propagated-in-iff-in-lits-of-l)
  show ‹C ∈# dom-m N›
    using C-N-U by (auto simp: S)

  have [simp]: ‹L' = L'a› ‹K = L'› ‹L'a = (N ∝ C ! (Suc 0 - i))› ‹j = i›
    using that two-le-length-C unfolding i-def
    by (auto simp: take-2-if S add-mset-eq-add-mset split: if-splits)
  have [simp]: ‹mset (swap (N ∝ C) 0 (Suc 0 - i)) = mset (N ∝ C)›
    apply (subst swap-multiset)
    using two-le-length-C unfolding i-def
    by (auto simp: S)
  have mset-un-watched-swap:
      ‹mset (watched-l (swap (N ∝ C) 0 (Suc 0 - i))) = mset (watched-l (N ∝ C))›
      ‹mset (unwatched-l (swap (N ∝ C) 0 (Suc 0 - i))) = mset (unwatched-l (N ∝ C))›
    using two-le-length-C unfolding i-def
    apply (auto simp: S take-2-if)
    by (auto simp: S swap-def)
```

```
  have irred-init: ‹irred N C ⟹ (N ∝ C, True) ∈# init-clss-l N›
    using C-N-U by (auto simp: S ran-def)
  have init-unchanged: ‹{#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
  . x ∈# init-clss-l (N(C ↪ swap (N ∝ C) 0 (Suc 0 − i)))#} =
  {#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
  . x ∈# init-clss-l N#}›
    using C-N-U
    by (cases ‹irred N C›) (auto simp: init-clss-l-mapsto-upd S image-mset-remove1-mset-if
      mset-un-watched-swap init-clss-l-mapsto-upd-irrel
      dest: multi-member-split[OF irred-init])


  have irred-init: ‹¬irred N C ⟹ (N ∝ C, False) ∈# learned-clss-l N›
    using C-N-U by (auto simp: S ran-def)
  have learned-unchanged: ‹{#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
  . x ∈# learned-clss-l (N(C ↪ swap (N ∝ C) 0 (Suc 0 − i)))#} =
   {#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
  . x ∈# learned-clss-l N#}›
    using C-N-U
    by (cases ‹irred N C›) (auto simp: init-clss-l-mapsto-upd S image-mset-remove1-mset-if
      mset-un-watched-swap learned-clss-l-mapsto-upd
      learned-clss-l-mapsto-upd-irrel
      dest: multi-member-split[OF irred-init])
  have [simp]: ‹{#(L, TWL-Clause (mset (watched-l
              (fst (the (if C = x
                        then Some (swap (N ∝ C) 0 (Suc 0 − i), irred N C)
                        else fmlookup N x)))))
         (mset (unwatched-l
              (fst (the (if C = x
                        then Some (swap (N ∝ C) 0 (Suc 0 − i), irred N C)
                        else fmlookup N x))))))
   . x ∈# WS#} = {#(L, TWL-Clause (mset (watched-l (N ∝ x))) (mset (unwatched-l (N ∝ x))))
   . x ∈# WS#}›
    by (rule image-mset-cong) (auto simp: mset-un-watched-swap)
  have C'-0i: ‹C' ! (Suc 0 − i) ∈ set (watched-l C')›
    using two-le-length-C by (auto simp: take-2-if S i-def)

  have nth-swap-isabelle: ‹length a ≥ 2 ⟹ swap a 0 (Suc 0 − i) ! 0 = a ! (Suc 0 − i)›
    for a :: ‹'a list›
    using two-le-length-C that apply (auto simp: swap-def S i-def)
    by (metis (full-types) le0 neq0-conv not-less-eq-eq nth-list-update-eq numeral-2-eq-2)
  have [simp]: ‹Propagated La C ∉ set M› for La
  proof (rule ccontr)
    assume H:‹¬ ?thesis›
    then have ‹La ∈ set (watched-l (N ∝ C))› and
      ‹2 < length (N ∝ C) ⟶ La = N ∝ C ! 0›
      using add-inv C-N-U two-le-length-C mset-un-watched-swap C'-0i
      unfolding twl-list-invs-def S by auto
    moreover have ‹La ∈ lits-of-l M›
      using H by (force simp: lits-of-def)
    ultimately show False
      using L'-undef that SS' uL-M n-d C'-i S watched-C' two-le-length-C
      apply (auto simp: S i-def dest: no-dup-consistentD split: if-splits)
apply (metis in-multiset-nempty member-add-mset no-dup-consistentD  set-mset-mset)
by (metis (mono-tags) in-multiset-nempty member-add-mset no-dup-consistentD set-mset-mset)
  qed
```

have ‹*twl-list-invs*
(*Propagated* (*N* ∝ *C* ! (*Suc* 0 − *i*)) *C* # *M*, *N*(*C* ↪ *swap* (*N* ∝ *C*) 0 (*Suc* 0 − *i*)),
D, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *remove1-mset C WS*, *add-mset* (− *N* ∝ *C* ! (*Suc* 0 − *i*)) *Q*)›
using *add-inv C-N-U two-le-length-C mset-un-watched-swap C′-0i*
unfolding *twl-list-invs-def*
by (*auto dest*: *in-diffD simp*: *set-conflicting-def ran-m-clause-upd*
set-conflict-l-def mset-take-mset-drop-mset′ S nth-swap-isabelle
dest!: *mset-eq-setD*)
moreover have
‹*convert-lit* (*N*(*C* ↪ *swap* (*N* ∝ *C*) 0 (*Suc* 0 − *i*))) (*NEk* + *UEk*)
(*Propagated* (*N* ∝ *C* ! (*Suc* 0 − *i*)) *C*)
(*Propagated* (*N* ∝ *C* ! (*Suc* 0 − *i*)) (*mset* (*N* ∝ *C*)))›
by (*auto simp*: *convert-lit.simps C-0*)
moreover have ‹(*M*, *x*) ∈ *convert-lits-l N* (*NEk* + *UEk*) ⟹
(*M*, *x*) ∈ *convert-lits-l* (*N*(*C* ↪ *swap* (*N* ∝ *C*) 0 (*Suc* 0 − *i*))) (*NEk* + *UEk*)› **for** *x*
apply (*rule convert-lits-l-extend-mono*)
apply *assumption*
apply *auto*
done
moreover have
‹*convert-lit N* (*NEk* + *UEk*)
(*Propagated* (*N* ∝ *C* ! (*Suc* 0 − *i*)) *C*)
(*Propagated* (*N* ∝ *C* ! (*Suc* 0 − *i*)) (*mset* (*N* ∝ *C*)))›
using *C-N-U* **by** (*auto simp*: *S convert-lit.simps C-0*)
moreover have ‹*twl-list-invs*
(*Propagated* (*N* ∝ *C* ! (*Suc* 0 − *i*)) *C* # *M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*,
NS, *US*, *N0*, *U0*, *remove1-mset C WS*, *add-mset* (− *N* ∝ *C* ! (*Suc* 0 − *i*)) *Q*)›
if ‹¬ 2 < *length* (*N* ∝ *C*)›
using *add-inv C-N-U two-le-length-C mset-un-watched-swap C′-0i that*
unfolding *twl-list-invs-def*
by (*auto dest*: *in-diffD simp*: *set-conflicting-def*
set-conflict-l-def mset-take-mset-drop-mset′ S nth-swap-isabelle
dest!: *mset-eq-setD*)
moreover have ‹*swap* (*N* ∝ *C*) 0 (*Suc* 0) = *swap* (*N* ∝ *C*) *i* (1 − *i*)›
using *i-def two-le-length-C* **by** (*cases* ‹*N* ∝ *C*›)(*auto simp*: *swap-def*)
moreover have ‹*i* < *length* (*N* ∝ *C*)› ‹*i* ≤ 1›
using *i-def two-le-length-C* **by** (*auto simp*: *S*)
moreover have ‹*undefined-lit M L′*›
using *L′-undef SS′* **by** (*auto simp*: *S Decided-Propagated-in-iff-in-lits-of-l*)
moreover have ‹*N* ∝ *C* ! (*Suc* 0 − *i*)
∈# *all-lits-of-st-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *remove1-mset C WS*, *Q*)›
using *C-N-U two-le-length-C* **by** (*auto simp*: *S ran-m-def all-lits-of-mm-add-mset i-def*
all-lits-of-st-l-def
intro!: *in-clause-in-all-lits-of-m nth-mem dest*!: *multi-member-split*)
moreover have ‹*add-mset L* (*add-mset* (*N* ∝ *C* ! (*Suc* 0 − *i*)) (*mset* (*unwatched-l* (*N* ∝ *C*)))) =
mset (*N* ∝ *C*)›
apply (*subst* (*4*) *append-take-drop-id*[*of* 2, *symmetric*])
using *that* **unfolding** *mset-append*
by (*auto simp*: *S*)
ultimately show
‹*K* ∈# *all-lits-of-st-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *remove1-mset C WS*, *Q*)›
‹*j* ≤ 1›
‹*do* {

```
        M ← cons-trail-propagate-l K C M;
        N ← if 2 < length (N ∝ C)
            then mop-clauses-swap N C 0 (Suc 0 − j) else RETURN N;
        RETURN
         (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, remove1-mset C WS,
          add-mset (− K) Q)
      } ≤ SPEC(λC.
          (C,
           (propagate-lit L′ (twl-clause-of C′)
            (set-clauses-to-update
             (remove1-mset (L, twl-clause-of C′)
              (clauses-to-update S′))
             S′))) ∈ {(S, S′).
           (S, S′) ∈ twl-st-l (Some L) ∧
           twl-list-invs S})›
    using SS′ WS C-N-U that unfolding propagate-lit-l-def apply (auto simp: S)
    by (auto simp: twl-st-l-def image-mset-remove1-mset-if propagate-lit-def
        propagate-lit-l-def mset-take-mset-drop-mset′ S learned-unchanged mop-clauses-swap-def
        cons-trail-propagate-l-def mop-propagate-lit-def
      init-unchanged mset-un-watched-swap intro: convert-lit.intros
      intro!: ASSERT-leI ASSERT-refine-right)
  qed
  have update-clause-rel: ‹do {
    K′ ← mop-clauses-at
        (get-clauses-l
          (set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S))
            S))
        C (the K);
   val-K ← mop-polarity-l
     (set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S)) S) K′;
   (if val-K = Some True
   then RETURN (set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S)) S)
    else update-clause-l C j (the K) (set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S))
S))}
   ≤ ⇓ {(S, S′). (S, S′) ∈ twl-st-l (Some L) ∧ twl-list-invs S}
      (update-clauseS L (twl-clause-of C′)
        (set-clauses-to-update (remove1-mset (L, twl-clause-of C′) (clauses-to-update S′)) S′))›
   (is ‹?update-clss ≤ ⇓ - -›)
  if
   L′: ‹L′ ∈ {K. K ∈# clause (twl-clause-of C′)}› and
   K: ‹K ∈ {found. (found = None) =
        (∀ L∈set (unwatched-l (get-clauses-l ?S ∝ C)).
           − L ∈ lits-of-l (get-trail-l ?S)) ∧
        (∀ j. found = Some j ⟶
            j < length (get-clauses-l ?S ∝ C) ∧
            (undefined-lit (get-trail-l ?S) (get-clauses-l ?S ∝ C ! j) ∨
             get-clauses-l ?S ∝ C ! j ∈ lits-of-l (get-trail-l ?S)) ∧
            2 ≤ j)}› and
   K-None: ‹K ≠ None› and
   ‹watched (twl-clause-of C′) = {#L, L′a#}› and
   val: ‹(val-L′, False) ∈ pol-spec L′a› and
   ‹(j, j′) ∈ {(j, j′). j = i ∧ i < 2}›
   for L′ and K and L′a and j j′ and val-L′
  proof −
   have L′a: ‹L′a
     ∉ lits-of-l
```

```
    (get-trail
      (set-clauses-to-update
        (remove1-mset (L, twl-clause-of C') (clauses-to-update S')) S'))›
    using n-d val SS' by (auto simp: pol-spec-def dest: no-dup-consistentD)
  have [simp]: ‹j = i›
    using that by auto
  obtain K' where [simp]: ‹K = Some K'›
    using K-None by auto
  have
    K'-le: ‹K' < length (N ∝ C)› and
    K'-2: ‹2 ≤ K'› and
    K'-M: ‹undefined-lit M (N ∝ C ! K') ∨
      N ∝ C ! K' ∈ lits-of-l (get-trail-l S) ›
    using K by (auto simp: S)
  have [simp]: ‹N ∝ C ! K' ∈ set (unwatched-l (N ∝ C))›
    using K'-le K'-2 by (auto simp: set-drop-conv S)
  have [simp]: ‹- N ∝ C ! K' ∉ lits-of-l M ›
    using n-d K'-M by (auto simp: S Decided-Propagated-in-iff-in-lits-of-l
      dest: no-dup-consistentD)

  have irred-init: ‹irred N C ⟹ (N ∝ C, True) ∈# init-clss-l N›
    using C-N-U by (auto simp: S)
  have init-unchanged: ‹update-clauses
   ({#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
    . x ∈# init-clss-l N#},
    {#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
    . x ∈# learned-clss-l N#})
   (TWL-Clause (mset (watched-l (N ∝ C))) (mset (unwatched-l (N ∝ C)))) L
   (N ∝ C ! K')
   ({#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
    . x ∈# init-clss-l (N(C ↪ swap (N ∝ C) i K'))#},
    {#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
    . x ∈# learned-clss-l (N(C ↪ swap (N ∝ C) i K'))#})›
  proof (cases ‹irred N C›)
    case J-NE: True
    have L-L'-UW-N: ‹C' ∈# init-clss-lf N›
      using C-N-U J-NE unfolding take-set
      by (auto simp: S ran-m-def)

    let ?UW = ‹unwatched-l C'›
    have TWL-L-L'-UW-N: ‹TWL-Clause {#?L, ?L'#} (mset ?UW) ∈# twl-clause-of `# init-clss-lf
N›
      using imageI[OF L-L'-UW-N, of twl-clause-of] watched-C' by force
    let ?k' = ‹the K - 2›
    have ‹?k' < length (unwatched-l C')›
      using K'-le two-le-length-C K'-2 by (auto simp: S)
    then have H0: ‹TWL-Clause {#?UW ! ?k', ?L'#} (mset (list-update ?UW ?k' ?L)) =
      update-clause (TWL-Clause {#?L, ?L'#} (mset ?UW)) ?L (?UW ! ?k')›
      by (auto simp: mset-update)

    have H3: ‹{#L, C' ! (Suc 0 - i)#} = mset (watched-l (N ∝ C))›
      using K'-2 K'-le ‹C > 0› C'-i by (auto simp: S take-2-if C-N-U nth-tl i-def)
    have H4: ‹mset (unwatched-l C') = mset (unwatched-l (N ∝ C))›
      by (auto simp: S take-2-if C-N-U nth-tl)

    let ?New-C = ‹(TWL-Clause {#L, C' ! (Suc 0 - i)#} (mset (unwatched-l C')))›
```

```
    have wo: a = a' ⟹ b = b' ⟹ L = L' ⟹ K = K' ⟹ c = c' ⟹
      update-clauses a K L b c ⟹
      update-clauses a' K' L' b' c' for a a' b b' K L K' L' c c'
      by auto
    have [simp]: ‹C' ∈ fst ` {a. a ∈# ran-m N ∧ snd a} ⟷ irred N C›
      using C-N-U J-NE unfolding C' S ran-m-def
      by auto
    have C'-ran-N: ‹(C', True) ∈# ran-m N›
      using C-N-U J-NE unfolding C' S S
      by auto
    have upd: ‹update-clauses
       (twl-clause-of `# init-clss-lf N, twl-clause-of `# learned-clss-lf N)
       (TWL-Clause {#C' ! i, C' ! (Suc 0 − i)#} (mset (unwatched-l C'))) (C' ! i) (C' ! the K)
         (add-mset (update-clause (TWL-Clause {#C' ! i, C' ! (Suc 0 − i)#}
           (mset (unwatched-l C'))) (C' ! i) (C' ! the K))
          (remove1-mset
            (TWL-Clause {#C' ! i, C' ! (Suc 0 − i)#} (mset (unwatched-l C')))
            (twl-clause-of `# init-clss-lf N)), twl-clause-of `# learned-clss-lf N)›
      by (rule update-clauses.intros(1)[OF TWL-L-L'-UW-N])
    have K1: ‹mset (watched-l (swap (N∝C) i K')) = {#N∝C!K', N∝C!(1 − i)#}›
      using J-NE C-N-U C' K'-2 K'-le two-le-length-C
       by (auto simp: init-clss-l-mapsto-upd S image-mset-remove1-mset-if
        take-2-if swap-def i-def)
    have K2: ‹mset (unwatched-l (swap (N∝C) i K')) = add-mset (N∝C ! i)
           (remove1-mset (N∝C ! K') (mset (unwatched-l (N∝C))))›
      using J-NE C-N-U C' K'-2 K'-le two-le-length-C
      by (auto simp: init-clss-l-mapsto-upd S image-mset-remove1-mset-if mset-update
        take-2-if swap-def i-def drop-upd-irrelevant drop-Suc drop-update-swap)
    have K3: ‹mset (watched-l (N∝C)) = {#N∝C!i, N∝C!(1 − i)#}›
      using J-NE C-N-U C' K'-2 K'-le two-le-length-C
       by (auto simp: init-clss-l-mapsto-upd S image-mset-remove1-mset-if
        take-2-if swap-def i-def)

    show ?thesis
      apply (rule wo[OF - - - - - upd])
      subgoal by auto
      subgoal by (auto simp: S)
      subgoal by auto
      subgoal unfolding S H3[symmetric] H4[symmetric] by auto
      subgoal
      using J-NE C-N-U C' K'-2 K'-le two-le-length-C K1 K2 K3 C'-ran-N
       by (auto simp: init-clss-l-mapsto-upd S image-mset-remove1-mset-if
        learned-clss-l-mapsto-upd-irrel)
      done
  next
    assume J-NE: ‹¬irred N C›
    have L-L'-UW-N: ‹C' ∈# learned-clss-lf N›
      using C-N-U J-NE unfolding take-set
      by (auto simp: S ran-m-def)

    let ?UW = ‹unwatched-l C'›
   have TWL-L-L'-UW-N: ‹TWL-Clause {#?L, ?L'#} (mset ?UW) ∈# twl-clause-of `# learned-clss-lf
N›
      using imageI[OF L-L'-UW-N, of twl-clause-of] watched-C' by force
    let ?k' = ‹the K − 2›
```

have ‹*?k′ < length (unwatched-l C′)*›
  using *K′-le two-le-length-C K′-2* **by** (*auto simp*: *S*)
then have *H0*: ‹*TWL-Clause {#?UW ! ?k′, ?L′#} (mset (list-update ?UW ?k′ ?L)) =*
  update-clause (TWL-Clause {#?L, ?L′#} (mset ?UW)) ?L (?UW ! ?k′)›
   by (*auto simp*: *mset-update*)

have *H3*: ‹*{#L, C′ ! (Suc 0 − i)#} = mset (watched-l (N ∝ C))*›
  using *K′-2 K′-le* ‹*C > 0*› *C′-i* **by** (*auto simp*: *S take-2-if C-N-U nth-tl i-def*)
have *H4*: ‹*mset (unwatched-l C′) = mset (unwatched-l (N ∝ C))*›
  by (*auto simp*: *S take-2-if C-N-U nth-tl*)

let *?New-C* = ‹(*TWL-Clause {#L, C′ ! (Suc 0 − i)#} (mset (unwatched-l C′))*)›

have *wo*: *a = a′ ⟹ b = b′ ⟹ L = L′ ⟹ K = K′ ⟹ c = c′ ⟹*
  update-clauses a K L b c ⟹
  update-clauses a′ K′ L′ b′ c′ **for** *a a′ b b′ K L K′ L′ c c′*
  by *auto*
have [*simp*]: ‹*C′ ∈ fst ‘ {a. a ∈# ran-m N ∧ ¬snd a} ⟷ ¬irred N C*›
  using *C-N-U J-NE* **unfolding** *C′ S ran-m-def*
  by *auto*
have *C′-ran-N*: ‹(*C′*, *False*) *∈# ran-m N*›
  using *C-N-U J-NE* **unfolding** *C′ S S*
  by *auto*
have *upd*: ‹*update-clauses*
  (*twl-clause-of ‘# init-clss-lf N*, *twl-clause-of ‘# learned-clss-lf N*)
  (*TWL-Clause {#C′ ! i, C′ ! (Suc 0 − i)#} (mset (unwatched-l C′))) (C′ ! i)*
  (*C′ ! the K*)
  (*twl-clause-of ‘# init-clss-lf N*,
  add-mset
    (*update-clause*
      (*TWL-Clause {#C′ ! i, C′ ! (Suc 0 − i)#} (mset (unwatched-l C′))) (C′ ! i)*
      (*C′ ! the K*))
    (*remove1-mset*
      (*TWL-Clause {#C′ ! i, C′ ! (Suc 0 − i)#} (mset (unwatched-l C′))*)
      (*twl-clause-of ‘# learned-clss-lf N*)))
  ›
  by (*rule update-clauses.intros*(*2*)[*OF TWL-L-L′-UW-N*])
have *K1*: ‹*mset (watched-l (swap (N∝C) i K′)) = {#N∝C!K′, N∝C!(1 − i)#}*›
  using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
    by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
      take-2-if swap-def i-def)
have *K2*: ‹*mset (unwatched-l (swap (N∝C) i K′)) = add-mset (N∝C ! i)*
          (*remove1-mset (N∝C ! K′) (mset (unwatched-l (N∝C))))*›
  using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
  by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if mset-update*
      take-2-if swap-def i-def drop-upd-irrelevant drop-Suc drop-update-swap)
have *K3*: ‹*mset (watched-l (N∝C)) = {#N∝C!i, N∝C!(1 − i)#}*›
  using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
    by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
      take-2-if swap-def i-def)

show *?thesis*
  apply (*rule wo*[*OF - - - - - upd*])
  subgoal by *auto*
  subgoal by (*auto simp*: *S*)
  subgoal by *auto*

subgoal unfolding *S H3*[*symmetric*] *H4*[*symmetric*] **by** *auto*
subgoal
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C K1 K2 K3 C′-ran-N*
by (*auto simp*: *learned-clss-l-mapsto-upd S image-mset-remove1-mset-if*
init-clss-l-mapsto-upd-irrel)
done
qed
have ‹*distinct-mset WS*›
by (*metis* (*full-types*) *WS′-def WS′-def′ add-inv twl-list-invs-def*)
then have [*simp*]: ‹*C* ∉# *WS′*›
by (*auto simp*: *WS′-def′*)
have *H*: ‹{#(*L*, *TWL-Clause*
(*mset* (*watched-l*
(*fst* (*the* (*if C* = *x then Some* (*swap* (*N* ∝ *C*) *i K′*, *irred N C*)
else fmlookup N x)))))
(*mset* (*unwatched-l*
(*fst* (*the* (*if C* = *x then Some* (*swap* (*N* ∝ *C*) *i K′*, *irred N C*)
else fmlookup N x)))))). *x* ∈# *WS′*#} =
{#(*L*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *x*))) (*mset* (*unwatched-l* (*N* ∝ *x*)))). *x* ∈# *WS′*#}›
by (*rule image-mset-cong*) *auto*
have [*simp*]: ‹*Propagated La C* ∉ *set M*› **for** *La*
proof (*rule ccontr*)
assume *H*:‹¬ *?thesis*›
then have ‹*length* (*N* ∝ *C*) > *2* ⟹ *La* = *N* ∝ *C* ! *0*› **and**
‹*La* ∈ *set* (*watched-l* (*N* ∝ *C*))›
using *add-inv C-N-U two-le-length-C*
unfolding *twl-list-invs-def S* **by** *auto*
moreover have ‹*La* ∈ *lits-of-l M*›
using *H* **by** (*force simp*: *lits-of-def*)
ultimately show *False*
using *SS′ uL-M n-d K′-2 K′-le two-le-length-C arg-cong*[*OF that*(*4*), *of set-mset*] *L′a*
by (*auto simp*: *S i-def polarity-spec′ dest*: *no-dup-consistentD split*: *if-splits*)
qed
have *A*: ‹*?update-clss* = *do* {*x* ← *mop-clauses-at N C K′*;
if x ∈ *lits-of-l* (*get-trail-l* (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*))
then RETURN (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*)
else update-clause-l C i
(*the K*) (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*)}›
using *C-N-U K′-le n-d* **unfolding** *i-def*
by (*auto simp add*: *S i-def polarity-def mop-clauses-at-def mop-polarity-l-def all-lits-of-st-l-def*
ran-m-def all-lits-of-mm-add-mset in-clause-in-all-lits-of-m dest!: *multi-member-split*[*of C*]
dest: *in-lits-of-l-defined-litD*)

have *alt-defs*: ‹*C′* = *N* ∝ *C*›
unfolding *C′ S* **by** *auto*
have *list-invs-blit*: ‹*twl-list-invs* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS′*, *Q*)›
using *add-inv C-N-U two-le-length-C*
unfolding *twl-list-invs-def*
by (*auto dest*: *in-diffD simp*: *S WS′-def′*)
define *pick-lit* **where**
‹*pick-lit a* = *SPEC* (λ*L*. *L* ∈# *unwatched* (*twl-clause-of C′*) ∧ − *L* ∉ *lits-of-l a*)›
for *a* :: ‹('*v*, '*v clause*) *ann-lit list*›
have [*refine*]:
‹*a* = *get-trail S′* ⟹ *mop-clauses-at N C K′* ≤ ⇓ {(*L*, *L′*). *L* = *L′* ∧ *L′* = *op-clauses-at N C K′*}
(*pick-lit a*)› **for** *a*
using *C-N-U K′-le K′-M n-d SS′* **unfolding** *C′ S* **by** (*auto simp*: *mop-clauses-at-def twl-st-l-def*

pick-lit-def
RETURN-RES-refine-iff S op-clauses-at-def)

have ‹*twl-list-invs* (*M*, *N*(*C* ↪ *swap* (*N* ∝ *C*) *i K′*), *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS′*, *Q*)›
using *add-inv C-N-U two-le-length-C i-le-length-C K′-le*
unfolding *twl-list-invs-def*
by (*auto dest*: *in-diffD simp*: *set-conflicting-def ran-m-clause-upd*
set-conflict-l-def mset-take-mset-drop-mset′ S WS′-def′
dest!: *mset-eq-setD*)
moreover have ‹(*M*, *x*) ∈ *convert-lits-l N* (*NEk* + *UEk*) ⟹
(*M*, *x*) ∈ *convert-lits-l* (*N*(*C* ↪ *swap* (*N* ∝ *C*) *i K′*)) (*NEk* + *UEk*)› **for** *x*
apply (*rule convert-lits-l-extend-mono*)
by *auto*
ultimately show *?thesis*
apply (*cases S′*)
unfolding *update-clauseS-def* **unfolding** *pick-lit-def*[*symmetric*]
apply (*clarsimp simp only*: *clauses-to-update.simps set-clauses-to-update.simps*)
apply (*subst A*)
apply *refine-vcg*
subgoal using *C-N-U K′-le K′-M n-d SS′* **unfolding** *C′ S* **by** (*auto simp*: *mop-clauses-at-def twl-st-l-def*)
subgoal using *SS′ K′-M* **unfolding** *C′ S* **by** (*auto simp*: *twl-st-l-def*)
subgoal using *SS′ K′-M add-inv list-invs-blit* **unfolding** *C′ S*
by (*auto simp*: *twl-st-l-def WS′-def′*)
subgoal for *a b c d e f g h x Ka*
using *SS′ init-unchanged C-N-U i-le-length-C K′-le C′-i*
unfolding *i-def*[*symmetric*] *get-clauses-l-set-clauses-to-update-l*
unfolding *update-clause-l-pre-def*
by (*auto simp*: *S update-clause-l-def update-clauseS-def twl-st-l-def WS′-def′*
RETURN-SPEC-refine RES-RES-RETURN-RES RETURN-def RES-RES2-RETURN-RES H
mop-clauses-swap-def op-clauses-at-def WS′-def′ update-clause-l-pre-def
all-lits-of-st-l-def
intro!: *RES-refine exI*[*of* - ‹*N* ∝ *C* ! *the K*›] *ASSERT-leI*
exI[*of* - ‹*set-clauses-to-update* (*remove1-mset* (*L*, *twl-clause-of* (*N* ∝ *C*))
(*clauses-to-update S′*)) *S′*›] *exI*[*of* - *L*] *image-mset-cong*)
done
qed

have *pos-of-watched*: ‹*unit-propagation-inner-loop-body-l-inv L C*
(*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*) ⟹ *pos-of-watched*
(*get-clauses-l*
(*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*))
C L
≤ *SPEC* (λ*c*. (*c*, ()) ∈ {(*j*, *j′*). *j* = *i* ∧ *i* < *2*})›
unfolding *pos-of-watched-def*
by (*auto simp*: *S unit-propagation-inner-loop-body-l-inv-def i-def*
intro!: *ASSERT-leI*)

have [*refine*]: ‹*unit-propagation-inner-loop-body-l-inv L C*
(*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*) ⟹
(*K*, *bL′*) ∈ *Id* ⟹
bL′ ∈ { *K*. *K*∈# *clause* (*twl-clause-of C′*)} ⟹
mop-polarity-l
(*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*)

```
K
≤ SPEC
 (λ c. (c, bL′
      ∈ lits-of-l
       (get-trail
        (set-clauses-to-update
         (remove1-mset (L, twl-clause-of C′)
          (clauses-to-update S′))
         S′)))
    ∈ pol-spec bL′)› for bL′ K
 using assms n-d L C-N-U consistent unfolding pol-spec-def
 by (auto simp: mop-polarity-l-def polarity-def ran-m-def all-lits-of-mm-add-mset
   all-lits-of-st-l-def
   true-annot-iff-decided-or-true-lit dest!: in-set-takeD in-set-dropD
   dest!: multi-member-split[of C ‹dom-m (get-clauses-l S)›]
   intro!: ASSERT-leI intro!: in-clause-in-all-lits-of-m)

have I: ‹(x, ()) ∈ {-. True}› for x
 by auto
have L-i0: ‹L = get-clauses-l S ∝ C ! 0 ⟹ Suc 0 − i = Suc 0›
 by (auto simp: i-def)
have other-watched-l: ‹(i′, ia) ∈ {(j, j′). j = i ∧ i < 2} ⟹
 other-watched-l
 (set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S)) S) L
 C i′
 ≤ SPEC (λK. K ∈# remove1-mset L (watched (twl-clause-of C′)))› for i′ ia
 unfolding other-watched-l-def
 by (refine-vcg mop-clauses-at[THEN fref-to-Down-curry2, unfolded comp-def, THEN order-trans])
 (use mset-watched-C pre-inv in ‹auto simp: op-clauses-at-def C-N-U
       unit-propagation-inner-loop-body-l-inv-def L-i0
      intro!: ASSERT-leI split: if-splits
      intro!: mop-clauses-at[THEN fref-to-Down-curry2, unfolded comp-def, THEN order-trans]
      simp: other-watched-l-def›)

have H: ‹?A ≤ ⇓ {(S, S′). (S, S′) ∈ twl-st-l (Some L) ∧ twl-list-invs S} ?B›
 unfolding unit-propagation-inner-loop-body-l-def unit-propagation-inner-loop-body-alt-def
   option.case-eq-if find-unwatched-l-def op-clauses-at-def[symmetric]
   nres-monad3
 apply (refine-vcg pos-of-watched
    case-prod-bind[of - ‹If - -›]; remove-dummy-vars)
 subgoal by (rule pre-inv)
 subgoal unfolding C′ clause-twl-clause-of by auto
 subgoal using SS′ by (auto simp: pol-spec-def)
 subgoal by (rule upd-rel)
 subgoal
   by (rule other-watched-l)
 subgoal for L′
   using assms by (auto simp: pol-spec-def)
 subgoal by (rule upd-rel)
 subgoal using SS′ C-N-U by auto
 subgoal using SS′ C-N-U two-le-length-C by auto
 subgoal using SS′ C-N-U dist-C by auto
 subgoal using SS′ C-N-U dist-C n-d by auto
 subgoal using SS′ by auto
 subgoal using SS′ by (auto simp: pol-spec-def)
 subgoal by (rule confl-rel)
```

subgoal for *K bL′ j j′ L′ L′a2 L′a3* **unfolding** *i-def*[*symmetric*] *op-clauses-at-def i-def′*[*symmetric*]
by (*rule propa-rel*)
subgoal by *auto*
subgoal for *L′ K* **unfolding** *i-def*[*symmetric*] *i-def′*[*symmetric*] *op-clauses-at-def Let-def*
by (*rule update-clause-rel*)
done
have ∗: ‹*unit-propagation-inner-loop-body* (*C′* ! *i*) (*twl-clause-of C′*)
(*set-clauses-to-update* (*remove1-mset* (*C′* ! *i*, *twl-clause-of C′*) (*clauses-to-update S′*)) *S′*)
≤ *SPEC* (λ*S″*. *twl-struct-invs S″* ∧
twl-stgy-invs S″ ∧
*cdcl-twl-cp*** *S′ S″* ∧
(*S″*, *S′*) ∈ *measure* (*size* ∘ *clauses-to-update*))›
apply (*rule unit-propagation-inner-loop-body*(*1*)[*of S′* ‹*C′* ! *i*› ‹*twl-clause-of C′*›])
using *imageI*[*OF WS*, *of* ‹(λ*j*. (*L*, *twl-clause-of* (*N* ∝ *j*)))›]
struct-invs stgy-inv C-N-U WS SS′ D-None **by** *auto*
have *H′*: ‹*?B* ≤ *SPEC* (λ*S′*. *twl-stgy-invs S′* ∧ *twl-struct-invs S′*)›
using ∗
by (*simp add*: *weaken-SPEC*)
have ‹*?A*
≤ ⇓ {(*S*, *S′*). ((*S*, *S′*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*) ∧
(*twl-stgy-invs S′* ∧ *twl-struct-invs S′*)}
?B›
apply (*rule refine-add-invariants*)
apply (*rule H′*)
by (*rule H*)
then show *?thesis* **by** *simp*
qed

lemma *unit-propagation-inner-loop-body-l2*:
assumes
SS′: ‹(*S*, *S′*) ∈ *twl-st-l* (*Some L*)› **and**
WS: ‹*C* ∈# *clauses-to-update-l S*› **and**
struct-invs: ‹*twl-struct-invs S′*› **and**
add-inv: ‹*twl-list-invs S*› **and**
stgy-inv: ‹*twl-stgy-invs S′*›
shows
‹(*unit-propagation-inner-loop-body-l L C*
(*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*),
unit-propagation-inner-loop-body L (*twl-clause-of* (*get-clauses-l S* ∝ *C*))
(*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of* (*get-clauses-l S* ∝ *C*))
(*clauses-to-update S′*)) *S′*))
∈ ⟨{(*S*, *S′*). (*S*, *S′*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S* ∧ *twl-stgy-invs S′* ∧
twl-struct-invs S′}⟩*nres-rel*›
using *unit-propagation-inner-loop-body-l*[*OF assms*]
by (*auto simp*: *nres-rel-def*)

This a work around equality: it allows to instantiate variables that appear in goals by hand in a reasonable way (*rule\-tac I=x in EQI*).

definition *EQ* **where**
[*simp*]: ‹*EQ* = (=)›

lemma *EQI*: *EQ I I*
by *auto*

lemma *unit-propagation-inner-loop-body-l-unit-propagation-inner-loop-body*:

‹*EQ L'' L''* ⟹
(*uncurry2 unit-propagation-inner-loop-body-l*, *uncurry2 unit-propagation-inner-loop-body*) ∈
{(((*L*, *C*), *S0*), ((*L'*, *C'*), *S0'*)). ∃ *S S'*. *L* = *L'* ∧ *C'* = (*twl-clause-of* (*get-clauses-l S* ∝ *C*)) ∧
S0 = (*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*) ∧
S0' = (*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of* (*get-clauses-l S* ∝ *C*))
(*clauses-to-update S'*)) *S'*) ∧
(*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *L* = *L''* ∧
C ∈# *clauses-to-update-l S* ∧ *twl-struct-invs S'* ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'*} →*f*
⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l* (*Some L''*) ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'* ∧
twl-struct-invs S'}⟩*nres-rel*›
apply (*intro frefI nres-relI*)
using *unit-propagation-inner-loop-body-l*
by *fastforce*

definition *select-from-clauses-to-update* :: ‹'*v twl-st-l* ⇒ ('*v twl-st-l* × *nat*) *nres*› **where**
‹*select-from-clauses-to-update S* = *SPEC* (λ(*S'*, *C*). *C* ∈# *clauses-to-update-l S* ∧
S' = *set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*)›

definition *unit-propagation-inner-loop-l-inv* **where**
‹*unit-propagation-inner-loop-l-inv L* = (λ(*S*, *n*).
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-struct-invs S'* ∧ *twl-stgy-invs S'* ∧
twl-list-invs S ∧ (*clauses-to-update S'* ≠ {#} ∨ *n* > 0 ⟶ *get-conflict S'* = *None*) ∧
−*L* ∈ *lits-of-l* (*get-trail-l S*)))›

definition *unit-propagation-inner-loop-body-l-with-skip* **where**
‹*unit-propagation-inner-loop-body-l-with-skip L* = (λ(*S*, *n*). *do* {
ASSERT (*clauses-to-update-l S* ≠ {#} ∨ *n* > 0);
ASSERT(*unit-propagation-inner-loop-l-inv L* (*S*, *n*));
b ← *SPEC*(λ*b*. (*b* ⟶ *n* > 0) ∧ (¬*b* ⟶ *clauses-to-update-l S* ≠ {#}));
if ¬*b* *then do* {
ASSERT (*clauses-to-update-l S* ≠ {#});
(*S'*, *C*) ← *select-from-clauses-to-update S*;
T ← *unit-propagation-inner-loop-body-l L C S'*;
RETURN (*T*, *if get-conflict-l T* = *None then n else* 0)
} *else RETURN* (*S*, *n*−1)
})›

definition *unit-propagation-inner-loop-l* :: ‹'*v literal* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*unit-propagation-inner-loop-l L* S_0 = *do* {
ASSERT(*L* ∈# *all-lits-of-st-l* S_0);
n ← *SPEC*(λ-::*nat*. *True*);
(*S*, *n*) ← $WHILE_T$*unit-propagation-inner-loop-l-inv L*
(λ(*S*, *n*). *clauses-to-update-l S* ≠ {#} ∨ *n* > 0)
(*unit-propagation-inner-loop-body-l-with-skip L*)
(S_0, *n*);
RETURN S
}›

lemma *set-mset-clauses-to-update-l-set-mset-clauses-to-update-spec*:
assumes ‹(*S*, *S'*) ∈ *twl-st-l* (*Some L*)›
shows
‹*RES* (*set-mset* (*clauses-to-update-l S*)) ≤ ⇓ {(*C*, (*L'*, *C'*)). *L'* = *L* ∧
C' = *twl-clause-of* (*get-clauses-l S* ∝ *C*)}
(*RES* (*set-mset* (*clauses-to-update S'*)))›
proof −

obtain *M N D NE UE WS Q* **where**
S: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
by (*cases S*) *auto*
show *?thesis*
using *assms* **unfolding** *S* **by** (*auto simp add*: *Bex-def twl-st-l-def intro*!: *RES-refine*)
qed

lemma *clauses-to-update-l-empty-tw-st-of-Some-None*[*simp*]:
‹*clauses-to-update-l S* = {#} ⟹ (*S*, *S'*)∈ *twl-st-l* (*Some L*) ⟷ (*S*, *S'*) ∈ *twl-st-l None*›
by (*cases S*) (*auto simp*: *twl-st-l-def*)

lemma *cdcl-twl-cp-in-trail-stays-in*:
‹*cdcl-twl-cp*** *S' aa* ⟹ − *x1* ∈ *lits-of-l* (*get-trail S'*) ⟹ − *x1* ∈ *lits-of-l* (*get-trail aa*)›
by (*induction rule*: *rtranclp-induct*)
(*auto elim*!: *cdcl-twl-cpE*)

lemma *cdcl-twl-cp-in-trail-stays-in-l*:
‹(*x2*, *S'*) ∈ *twl-st-l* (*Some x1*) ⟹ *cdcl-twl-cp*** *S' aa* ⟹ − *x1* ∈ *lits-of-l* (*get-trail-l x2*) ⟹
(*a*, *aa*) ∈ *twl-st-l* (*Some x1*) ⟹ − *x1* ∈ *lits-of-l* (*get-trail-l a*)›
using *cdcl-twl-cp-in-trail-stays-in*[*of S' aa* ‹*x1*›]
by (*auto simp*: *twl-st twl-st-l*)

lemma *unit-propagation-inner-loop-l*:
‹(*uncurry unit-propagation-inner-loop-l*, *unit-propagation-inner-loop*) ∈
{((*L*, *S*), *S'*). (*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S* ∧ −*L* ∈ *lits-of-l* (*get-trail-l S*) ∧
L ∈# *all-lits-of-st-l S*} →*f*
⟨{(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧ *clauses-to-update-l T* = {#} ∧
twl-list-invs T}⟩ *nres-rel*›
(**is** ‹*?unit-prop-inner* ∈ *?A* →*f* ⟨*?B*⟩*nres-rel*›)
proof −
have *SPEC-remove*: ‹*select-from-clauses-to-update S*
≤ ⇓ {((*T'*, *C*), *C'*).
(*T'*, *set-clauses-to-update* (*clauses-to-update S''* − {#*C'*#}) *S''*) ∈ *twl-st-l* (*Some L*) ∧
T' = *set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S* ∧
C' ∈# *clauses-to-update S''* ∧
C ∈# *clauses-to-update-l S* ∧
snd C' = *twl-clause-of* (*get-clauses-l S* ∝ *C*)}
(*SPEC* (λ*C*. *C* ∈# *clauses-to-update S''*))›
if ‹(*S*, *S''*) ∈ {(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs T*}›
for *S* :: ‹*'v twl-st-l*› **and** *S''* *L*
using *that* **unfolding** *select-from-clauses-to-update-def*
by (*auto simp*: *conc-fun-def image-mset-remove1-mset-if twl-st-l-def*)
show *?thesis*
unfolding *unit-propagation-inner-loop-l-def unit-propagation-inner-loop-def uncurry-def*
unit-propagation-inner-loop-body-l-with-skip-def
apply (*intro frefI nres-relI*)
subgoal for *LS S'*
apply (*rewrite* **in** ‹*let* - = *set-clauses-to-update* - - *in* -› *Let-def*)
apply (*refine-vcg set-mset-clauses-to-update-l-set-mset-clauses-to-update-spec*
WHILEIT-refine-genR[**where**
R = ‹{(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs T* ∧ *clauses-to-update-l T* = {#}}
×*f* *nat-rel*› **and**
R' = ‹{(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l* (*Some* (*fst LS*)) ∧ *twl-list-invs T*}
×*f* *nat-rel*›]
unit-propagation-inner-loop-body-l-unit-propagation-inner-loop-body[*THEN fref-to-Down-curry2*]
SPEC-remove;

```
      remove-dummy-vars)
    subgoal for x1 x2
      by (auto simp add: in-all-lits-of-mm-uminus-iff twl-st-l
       mset-take-mset-drop-mset')
    subgoal by simp
    subgoal for x1 x2 n na x x' unfolding unit-propagation-inner-loop-l-inv-def
      apply (case-tac x; case-tac x')
      apply (simp only: prod.simps)
      by (rule exI[of - ‹fst x'›]) (auto intro: cdcl-twl-cp-in-trail-stays-in-l)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
       apply (subst (asm) prod-rel-iff)
       apply normalize-goal
        apply assumption
    apply (rule-tac I=x1 in EQI)
    subgoal for x1 x2 n na x1a x2a x1b x2b b ba x1c x2c x1d x2d
      apply (subst in-pair-collect-simp)
      apply (subst prod.case)+
      apply (rule-tac x = x1b in exI)
      apply (rule-tac x = x1a in exI)
      apply (intro conjI)
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      done
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
  done
qed

definition clause-to-update :: ‹'v literal ⇒ 'v twl-st-l ⇒ 'v clauses-to-update-l›where
  ‹clause-to-update L S =
    filter-mset
      (λC::nat. L ∈ set (watched-l (get-clauses-l S ∝ C)))
      (dom-m (get-clauses-l S))›

lemma distinct-mset-clause-to-update: ‹distinct-mset (clause-to-update L C)›
  unfolding clause-to-update-def
  apply (rule distinct-mset-filter)
  using distinct-mset-dom by blast

lemma in-clause-to-updateD: ‹b ∈# clause-to-update L' T ⟹ b ∈# dom-m (get-clauses-l T)›
```

by (*auto simp*: *clause-to-update-def*)

lemma *in-clause-to-update-iff*:
‹*C* ∈# *clause-to-update L S* ⟷
C ∈# *dom-m* (*get-clauses-l S*) ∧ *L* ∈ *set* (*watched-l* (*get-clauses-l S* ∝ *C*))›
by (*auto simp*: *clause-to-update-def*)

definition *select-and-remove-from-literals-to-update* :: ‹'*v twl-st-l* ⇒
('*v twl-st-l* × '*v literal*) *nres*› **where**
‹*select-and-remove-from-literals-to-update S* = *SPEC*(λ(*S'*, *L*). *L* ∈# *literals-to-update-l S* ∧
S' = *set-clauses-to-update-l* (*clause-to-update L S*)
(*set-literals-to-update-l* (*literals-to-update-l S* − {#*L*#}) *S*))›

definition *unit-propagation-outer-loop-l-inv* **where**
‹*unit-propagation-outer-loop-l-inv S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'* ∧ *twl-stgy-invs S'* ∧
clauses-to-update-l S = {#})›

definition *unit-propagation-outer-loop-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*unit-propagation-outer-loop-l S*$_0$ =
WHILE$_T$*unit-propagation-outer-loop-l-inv*
(λ*S*. *literals-to-update-l S* ≠ {#})
(λ*S*. *do* {
ASSERT(*literals-to-update-l S* ≠ {#});
(*S'*, *L*) ← *select-and-remove-from-literals-to-update S*;
unit-propagation-inner-loop-l L S'
})
(*S*$_0$ :: '*v twl-st-l*)
›

lemma *watched-twl-clause-of-watched*: ‹*watched* (*twl-clause-of x*) = *mset* (*watched-l x*)›
by (*cases x*) *auto*

lemma *twl-st-of-clause-to-update*:
assumes
TT': ‹(*T*, *T'*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs T'*›
shows
‹(*set-clauses-to-update-l*
(*clause-to-update L' T*)
(*set-literals-to-update-l* (*remove1-mset L'* (*literals-to-update-l T*)) *T*),
set-clauses-to-update
(*Pair L'* '# {#*C* ∈# *get-clauses T'*. *L'* ∈# *watched C*#})
(*set-literals-to-update* (*remove1-mset L'* (*literals-to-update T'*))
T'))
∈ *twl-st-l* (*Some L'*)›
proof −
obtain *M N D NE UE NEk UEk NS US N0 U0 WS Q* **where**
T: ‹*T* = (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases T*) *auto*

have
‹{#(*L'*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *x*)))
(*mset* (*unwatched-l* (*N* ∝ *x*)))).
x ∈# {#*C* ∈# *dom-m N*. *L'* ∈ *set* (*watched-l* (*N* ∝ *C*))#}#} =
Pair L' '#

```
    {#C ∈# {#TWL-Clause (mset (watched-l x)) (mset (unwatched-l x)). x ∈# init-clss-lf N#} +
          {#TWL-Clause (mset (watched-l x)) (mset (unwatched-l x)). x ∈# learned-clss-lf N#}.
     L' ∈# watched C#}›
   (is ‹{#(L', ?C x). x ∈# ?S#} = Pair L' `# ?C'›)
  proof −
   have H: ‹{#f (N ∝ x). x ∈# {#x ∈# dom-m N. P (N ∝ x)#}#} =
     {#f (fst x). x ∈# {#C ∈# ran-m N. P (fst C)#}#}› for P and f :: ‹'a literal list ⇒ 'b›
      unfolding ran-m-def image-mset-filter-swap2 by auto

   have H: ‹{#f (N∝x). x ∈# ?S#} =
      {#f (fst x). x ∈# {#C ∈# init-clss-l N. L' ∈ set (watched-l (fst C))#}#} +
      {#f (fst x). x ∈# {#C ∈# learned-clss-l N. L' ∈ set (watched-l (fst C))#}#}›
     for f :: ‹'a literal list ⇒ 'b›
    unfolding image-mset-union[symmetric] filter-union-mset[symmetric]
    apply auto
    apply (subst H)
    ..

   have L'': ‹{#(L', ?C x). x ∈# ?S#} = Pair L' `# {#?C x. x ∈# ?S#}›
    by auto
   also have ‹... = Pair L' `# ?C'›
    apply (rule arg-cong[of - - ‹(`#) (Pair L')›])
    unfolding image-mset-union[symmetric] mset-append[symmetric] drop-Suc H
    apply simp
    apply (subst H)
    unfolding image-mset-union[symmetric] mset-append[symmetric] drop-Suc H
      filter-union-mset[symmetric] image-mset-filter-swap2
    by auto
   finally show ?thesis .
  qed
  then show ?thesis
   using TT'
   by (cases T') (auto simp del: filter-union-mset
     simp: T split-beta clause-to-update-def twl-st-l-def
     split: if-splits)
qed

lemma twl-list-invs-set-clauses-to-update-iff:
  assumes ‹twl-list-invs T›
  shows ‹twl-list-invs (set-clauses-to-update-l WS (set-literals-to-update-l Q T)) ⟷
    ((∀x∈#WS. case x of C ⇒ C ∈# dom-m (get-clauses-l T)) ∧
    distinct-mset WS)›
proof −
  obtain M N C NE UE NEk UEk WS NS US N0 U0 Q where
    T: ‹T = (M, N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q)›
    by (cases T) auto
  show ?thesis
    using assms
    unfolding twl-list-invs-def T by auto
qed


lemma unit-propagation-outer-loop-l-spec:
  ‹(unit-propagation-outer-loop-l, unit-propagation-outer-loop) ∈
  {(S, S'). (S, S') ∈ twl-st-l None ∧ twl-list-invs S ∧ clauses-to-update-l S = {#}} →f
  ⟨{(S, S').
```

```
        (S, S') ∈ twl-st-l None ∧
        clauses-to-update-l S = {#} ∧
        twl-list-invs S}⟩ nres-rel›
  (is ‹- ∈ ?R →f ?I› is ‹- ∈ - →f ⟨?B⟩ nres-rel›)
proof −
  have twl-struct-invs-no-alien-in-trail: ‹unit-propagation-outer-loop-l-inv T ⟹
    −x2a ∈ lits-of-l (get-trail-l T) ⟹
    x2a ∈# all-lits-of-st-l T› for T x2a
    unfolding unit-propagation-outer-loop-l-inv-def
    apply normalize-goal+
    by (drule  twl-struct-invs-no-alien-in-trail[of - ‹−x2a›])
      (simp-all add: mset-take-mset-drop-mset' in-all-lits-of-mm-uminus-iff)

  have H:
    ‹select-and-remove-from-literals-to-update x
      ≤ ⇓ {((S', L'), (L, S)). L = L' ∧  S' = set-clauses-to-update-l (clause-to-update L x)
           (set-literals-to-update-l (remove1-mset L (literals-to-update-l x)) x) ∧
          L' ∈# literals-to-update-l x ∧
          S = set-clauses-to-update {#(L, C)|C ∈# get-clauses x'. L ∈# watched C#}
         (set-literals-to-update (literals-to-update x' − {#L#}) x')}
         (mop-pop-literal-to-update x')›
    if ‹(x, x') ∈ twl-st-l None› for x :: ‹'v twl-st-l› and x' :: ‹'v twl-st›
    using that unfolding select-and-remove-from-literals-to-update-def mop-pop-literal-to-update-def
      Let-def RES-RETURN-RES
    apply (cases x; cases x')
    apply refine-rcg
    by (clarsimp simp add: twl-st-l-def intro!: RES-refine)
  have H': ‹unit-propagation-outer-loop-l-inv T ⟹
    x2 ∈# literals-to-update-l T ⟹ − x2 ∈ lits-of-l (get-trail-l T)›
    for S S' T T' L L' C x2
    by (auto simp: unit-propagation-outer-loop-l-inv-def twl-st-l-def twl-struct-invs-def)
  show H:
    ‹(unit-propagation-outer-loop-l, unit-propagation-outer-loop) ∈?R →f
      ⟨{(S, S').
        (S, S') ∈ twl-st-l None ∧
        clauses-to-update-l S = {#} ∧
        twl-list-invs S}⟩ nres-rel›
    unfolding unit-propagation-outer-loop-l-def unit-propagation-outer-loop-def fref-param1[symmetric]
    apply (refine-vcg unit-propagation-inner-loop-l[THEN fref-to-Down-curry-left]
      H)
    subgoal by simp
    subgoal unfolding unit-propagation-outer-loop-l-inv-def by fastforce
    subgoal by auto
    subgoal by simp
    subgoal for S S' T T' L L' C x2
      by (auto simp add: twl-st-of-clause-to-update twl-list-invs-set-clauses-to-update-iff
        intro: cdcl-twl-cp-twl-struct-invs cdcl-twl-cp-twl-stgy-invs
        distinct-mset-clause-to-update H'
        dest: in-clause-to-updateD)
    subgoal for S S' T T' L L' C x2 x1a x2a
      using twl-struct-invs-no-alien-in-trail[of T ‹snd L›] H'[of T ‹snd L›]
      by (auto simp add: twl-st-of-clause-to-update twl-list-invs-set-clauses-to-update-iff
        intro: cdcl-twl-cp-twl-struct-invs cdcl-twl-cp-twl-stgy-invs
        distinct-mset-clause-to-update
        dest: in-clause-to-updateD)
    done
```

qed

lemma *get-conflict-l-get-conflict-state-spec*:
assumes ‹$(S, S') \in$ *twl-st-l None*› **and** ‹*twl-list-invs S*› **and** ‹*clauses-to-update-l S* $= \{\#\}$›
shows ‹$((\mathit{False}, S), (\mathit{False}, S'))$
$\in \{((\mathit{brk}, S), (\mathit{brk}', S')).\ \mathit{brk} = \mathit{brk}' \wedge (S, S') \in \mathit{twl\text{-}st\text{-}l\ None} \wedge \mathit{twl\text{-}list\text{-}invs}\ S \wedge$
clauses-to-update-l S $= \{\#\}\}$›
using *assms* **by** *auto*

fun *lit-and-ann-of-propagated* **where**
‹*lit-and-ann-of-propagated* (*Propagated L C*) $= (L, C)$› |
‹*lit-and-ann-of-propagated* (*Decided* -) = *undefined*›
— we should never call the function in that context

definition *tl-state-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*tl-state-l* $= (\lambda(M, N, D, NE, UE, NEk, UEk, NS, US, WS, Q).$ (*tl M, N, D, NE, UE, NEk, UEk, NS, US, WS, Q*))›

definition *resolve-cls-l'* :: ‹*'v twl-st-l* ⇒ *nat* ⇒ *'v literal* ⇒ *'v clause*› **where**
‹*resolve-cls-l' S C L* =
remove1-mset L (*remove1-mset* $(-L)$ (*the* (*get-conflict-l S*) $\cup\#$ *mset* (*get-clauses-l S* $\propto$ *C*)))›

definition *update-confl-tl-l* :: ‹*'v literal* ⇒ *nat* ⇒ *'v twl-st-l* ⇒ *bool* × *'v twl-st-l*› **where**
‹*update-confl-tl-l* $= (\lambda L\ C\ (M, N, D, NE, UE, NEk, UEk, NS, US, WS, Q).$
let D = *resolve-cls-l'* (*M, N, D, NE, UE, NEk, UEk, NS, US, WS, Q*) *C L in*
(*False*, (*tl M, N, Some D, NE, UE, NEk, UEk, NS, US, WS, Q*)))›

definition *update-confl-tl-l-pre* :: ‹*'v literal* ⇒ *nat* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*update-confl-tl-l-pre L C S* ⟷
($\exists S'.$ $(S, S') \in$ *twl-st-l None* $\wedge$ *update-confl-tl-pre L* (*mset* (*get-clauses-l S* $\propto$ *C*)) *S'* $\wedge$
twl-list-invs S $\wedge$ *C* $\in\#$ *dom-m* (*get-clauses-l S*) $\wedge$ *get-trail-l S* $\neq$ [] $\wedge$ *hd* (*get-trail-l S*) = *Propagated L C* $\wedge$ *C* > 0)›

definition *mop-update-confl-tl-l* :: ‹*'v literal* ⇒ *nat* ⇒ *'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres*› **where**
‹*mop-update-confl-tl-l* $= (\lambda L\ C\ S.$ *do* {
ASSERT(*update-confl-tl-l-pre L C S*);
RETURN (*update-confl-tl-l L C S*)})›

definition *mop-hd-trail-l-pre* :: ‹*'v twl-st-l* ⇒ *bool*› **where**
‹*mop-hd-trail-l-pre S* ⟷
($\exists S'.$ $(S, S') \in$ *twl-st-l None* $\wedge$ *mop-hd-trail-pre S'* $\wedge$
twl-list-invs S $\wedge$ *mark-of* (*hd* (*get-trail-l S*)) > 0)›

definition *mop-hd-trail-l* :: ‹*'v twl-st-l* ⇒ (*'v literal* × *nat*) *nres*› **where**
‹*mop-hd-trail-l S* = *do*{
ASSERT(*mop-hd-trail-l-pre S*);
SPEC($\lambda(L, C).$ *Propagated L C* = *hd* (*get-trail-l S*))
}›

definition *mop-lit-notin-conflict-l* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *bool nres*› **where**
‹*mop-lit-notin-conflict-l L S* = *do* {
ASSERT(*get-conflict-l S* $\neq$ *None* $\wedge$ $-L \notin\#$ *the* (*get-conflict-l S*) $\wedge$ *L* $\in\#$ *all-lits-of-st-l S*);
RETURN (*L* $\notin\#$ *the* (*get-conflict-l S*))
}›

definition *mop-tl-state-l-pre* :: ‹*'v twl-st-l* ⇒ *bool*› **where**

‹*mop-tl-state-l-pre* $S \longleftrightarrow$
  ($\exists S'$. $(S, S') \in$ *twl-st-l None* $\wedge$ *mop-tl-state-pre* $S' \wedge$
    twl-list-invs S)›

definition *mop-tl-state-l* :: ‹*'v twl-st-l* $\Rightarrow$ (*bool* $\times$ *'v twl-st-l*) *nres*› **where**
‹*mop-tl-state-l* = (λS. *do* {
  ASSERT(*mop-tl-state-l-pre S*);
  RETURN(*False*, *tl-state-l S*)})›

definition *mop-maximum-level-removed-l-pre* :: ‹*'v literal* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *bool*› **where**
‹*mop-maximum-level-removed-l-pre L S* $\longleftrightarrow$
  ($\exists S'$. $(S, S') \in$ *twl-st-l None* $\wedge$ *mop-maximum-level-removed-pre L* $S' \wedge$
    twl-list-invs S)›

definition *mop-maximum-level-removed-l* :: ‹*'v literal* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *bool nres*› **where**
‹*mop-maximum-level-removed-l L S* = *do* {
  ASSERT (*mop-maximum-level-removed-l-pre L S*);
  RETURN (*get-maximum-level* (*get-trail-l S*) (*remove1-mset* ($-L$) (*the* (*get-conflict-l S*))) =
    count-decided (*get-trail-l S*))
}›

definition *skip-and-resolve-loop-inv-l* **where**
‹*skip-and-resolve-loop-inv-l* S_0 *brk S* $\longleftrightarrow$
  ($\exists S'$ S_0'. $(S, S') \in$ *twl-st-l None* $\wedge$ $(S_0, S_0') \in$ *twl-st-l None* $\wedge$
    skip-and-resolve-loop-inv S_0' (*brk*, S') $\wedge$
      twl-list-invs S $\wedge$ *clauses-to-update-l* $S = \{\#\}$ $\wedge$
      ($\neg$*is-decided* (*hd* (*get-trail-l S*)) $\longrightarrow$ *mark-of* (*hd*(*get-trail-l S*)) $>$ *0*))›

definition *skip-and-resolve-loop-l* :: ‹*'v twl-st-l* $\Rightarrow$ *'v twl-st-l nres*› **where**
‹*skip-and-resolve-loop-l* S_0 =
  do {
    ASSERT(*get-conflict-l* $S_0 \neq$ *None*);
    (-, S) $\leftarrow$
      $WHILE_T^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\text{-}l\ S_0\ brk\ S}$
      (λ(*brk*, S). $\neg$*brk* $\wedge$ $\neg$*is-decided* (*hd* (*get-trail-l S*)))
      (λ(-, S).
        do {
          (L, C) $\leftarrow$ *mop-hd-trail-l S*;
          b $\leftarrow$ *mop-lit-notin-conflict-l* ($-L$) S;
          if b then
            mop-tl-state-l S
          else do {
            b $\leftarrow$ *mop-maximum-level-removed-l L S*;
            if b
            then
              mop-update-confl-tl-l L C S
            else
              RETURN (*True*, S)
          }
        }
)
      (*False*, S_0);
    RETURN S
  }
›

lemma *get-level-same-lits-cong*:
assumes
‹*map* (*atm-of o lit-of*) *M* = *map* (*atm-of o lit-of*) *M'*› **and**
‹*map is-decided M* = *map is-decided M'*›
shows ‹*get-level M L* = *get-level M' L*›
proof −
have [*dest*]: ‹*map is-decided M* = *map is-decided zsa* ⟹
length (*filter is-decided M*) = *length* (*filter is-decided zsa*)›
for *M* :: ‹('*d*, '*e*, '*f*) *annotated-lit list*› **and** *zsa* :: ‹('*g*, '*h*, '*i*) *annotated-lit list*›
by (*induction M arbitrary*: *zsa*) (*auto simp*: *get-level-def*)

show *?thesis*
using *assms*
by (*induction M arbitrary*: *M'*) (*auto simp*: *get-level-def*)
qed

lemma *clauses-in-unit-clss-have-level0*:
assumes
struct-invs: ‹*twl-struct-invs T*› **and**
C: ‹*C* ∈# *unit-clss T*› **and**
LC-T: ‹*Propagated L C* ∈ *set* (*get-trail T*)› **and**
count-dec: ‹0 < *count-decided* (*get-trail T*)›
shows
‹*get-level* (*get-trail T*) *L* = 0› (**is** *?lev-L*) **and**
‹∀ *K*∈# *C*. *get-level* (*get-trail T*) *K* = 0› (**is** *?lev-K*)
proof −
have
all-struct: ‹$cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv* ($state_W$-*of T*)› **and**
ent: ‹*entailed-clss-inv* ($pstate_W$-*of T*)›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv-def*
pcdcl-all-struct-invs-def $state_W$-*of-def*[*symmetric*]
by *fast+*
obtain *K* **where**
‹*K* ∈# *C*› **and** *lev-K*: ‹*get-level* (*get-trail T*) *K* = 0› **and** *K-M*: ‹*K* ∈ *lits-of-l* (*get-trail T*)›
using *ent C count-dec* **by** (*cases T*; *cases* ‹*get-conflict T*›) (*auto simp*: *entailed-clss-inv.simps*)

obtain *M1 M2* **where**
M: ‹*get-trail T* = *M2* @ *Propagated L C* # *M1*›
using *LC-T* **by** (*blast elim*: *in-set-list-format*)
have ‹$cdcl_W$-*restart-mset*.$cdcl_W$-*conflicting* ($state_W$-*of T*)› **and**
lev-inv: ‹$cdcl_W$-*restart-mset*.$cdcl_W$-*M-level-inv* ($state_W$-*of T*) ›
using *all-struct* **unfolding** $cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv-def*
by *fast+*
then have *M1*: ‹*M1* ⊨*as CNot* (*remove1-mset L C*)› **and** ‹*L* ∈# *C*›
using *M* **unfolding** $cdcl_W$-*restart-mset*.$cdcl_W$-*conflicting-def*
by (*auto simp*: *twl-st*)
moreover have *n-d*: ‹*no-dup* (*get-trail T*)›
using *lev-inv* **unfolding** $cdcl_W$-*restart-mset*.$cdcl_W$-*M-level-inv-def* **by** *simp*
ultimately have ‹*L* = *K*›
using ‹*K* ∈# *C*› *M K-M*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset*
dest: *in-lits-of-l-defined-litD no-dup-uminus-append-in-atm-notin*
no-dup-appendD no-dup-consistentD)
then show *?lev-L*
using *lev-K* **by** *simp*
have *count-dec-M1*: ‹*count-decided M1* = 0›

```
    using M n-d ‹?lev-L› by auto
  have ‹get-level (get-trail T) K = 0› if ‹K ∈# C› for K
  proof −
    have ‹−K ∈ lits-of-l (Propagated (−L) C # M1)›
    using M1 M that by (auto simp: true-annots-true-cls-def-iff-negation-in-model remove1-mset-add-mset-If
      dest!: multi-member-split dest: in-diffD split: if-splits)
    then have ‹get-level (get-trail T) K = get-level (Propagated (−L) C # M1) K›
      apply −
      apply (subst (2) get-level-skip[symmetric, of M2])
      using n-d M by (auto dest: no-dup-uminus-append-in-atm-notin
        intro: get-level-same-lits-cong)
    then show ?thesis
      using count-decided-ge-get-level[of ‹Propagated (−L) C # M1› K] count-dec-M1
      by (auto simp: get-level-cons-if split: if-splits)
  qed
  then show ?lev-K
    by fast
qed

lemma clauses-clss-have-level1-notin-unit:
  assumes
    struct-invs: ‹twl-struct-invs T› and
    LC-T: ‹Propagated L C ∈ set (get-trail T)› and
    count-dec: ‹0 < count-decided (get-trail T)› and
    ‹get-level (get-trail T) L > 0›
  shows
    ‹C ∉# unit-clss T›
  using clauses-in-unit-clss-have-level0[of T C, OF struct-invs - LC-T count-dec] assms
  by linarith

lemma skip-and-resolve-loop-l-spec:
  ‹(skip-and-resolve-loop-l, skip-and-resolve-loop) ∈
    {(S::'v twl-st-l, S'). (S, S') ∈ twl-st-l None ∧ twl-struct-invs S' ∧
      twl-stgy-invs S' ∧
      twl-list-invs S ∧ clauses-to-update-l S = {#} ∧ literals-to-update-l S = {#} ∧
      get-conflict S' ≠ None ∧
      0 < count-decided (get-trail-l S)} →f
  ⟨{(T, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T ∧
    (twl-struct-invs T' ∧ twl-stgy-invs T' ∧
    no-step cdcl_W-restart-mset.skip (state_W-of T') ∧
    no-step cdcl_W-restart-mset.resolve (state_W-of T') ∧
    literals-to-update T' = {#} ∧
    clauses-to-update-l T = {#} ∧ get-conflict T' ≠ None)}⟩ nres-rel›
  (is ‹- ∈ ?R →f -›)
proof −
  let ?R' = ‹{(T::'v twl-st-l, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T ∧
      clauses-to-update-l T = {#}}›
  have H: ‹((False, x), False, y) ∈ bool-rel ×f ?R'› if ‹(x, y) ∈ ?R'› for x y
    using that by auto

  have
    mark-ge-0: ‹0 < mark-of (hd (get-trail-l T))› (is ?ge) and
    nempty: ‹get-trail-l T ≠ []› ‹get-trail (snd brkT') ≠ []› (is ?nempty)
  if
    SS': ‹(S, S') ∈ ?R› and
    ‹get-conflict-l S ≠ None› and
```

brk-TT': ‹(*brkT*, *brkT'*)
∈ {((*brk*, *S*), *brk'*, *S'*). *brk* = *brk'* ∧ (*S*, *S'*) ∈ *twl-st-l None* ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#}}› (**is** ‹- ∈ *?brk*›) **and**
loop-inv: ‹*skip-and-resolve-loop-inv S' brkT'*› **and**
brkT: ‹*brkT* = (*brk*, *T*)› **and**
dec: ‹¬ *is-decided* (*hd* (*get-trail-l T*))›
for *S S' brkT brkT' brk T*
proof −
obtain *brk' T'* **where** *brkT'*: ‹*brkT'* = (*brk'*, *T'*)› **by** (*cases brkT'*)
have ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of T'*)› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of T'*)› **and**
tr: ‹*get-trail T'* ≠ []› ‹*get-trail-l T* ≠ []› **and**
count-dec: ‹*count-decided* (*get-trail-l T*) ≠ *0*› ‹*count-decided* (*get-trail T'*) ≠ *0*› **and**
TT': ‹(*T*,*T'*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T'*›
using *loop-inv brk-TT'* **unfolding** *twl-struct-invs-def skip-and-resolve-loop-inv-def brkT brkT'*
prod.simps state$_W$*-of-def*[*symmetric*] *pcdcl-all-struct-invs-def*
by *auto*
moreover have ‹*Suc 0* ≤ *backtrack-lvl* (*state*$_W$*-of T'*)›
using *count-dec TT'* **by** (*auto simp*: *trail.simps*)
moreover have *proped*: ‹*is-proped* (*cdcl*$_W$*-restart-mset.hd-trail* (*state*$_W$*-of T'*))›
using *dec tr TT'* **by** (*cases* ‹*get-trail-l T*›)
(*auto simp*: *trail.simps is-decided-no-proped-iff twl-st*)
moreover have ‹*mark-of* (*hd* (*get-trail T'*)) ∉# *get-kept-unit-clauses-l T*›
using *clauses-clss-have-level1-notin-unit*(*1*)[*of T'* ‹*lit-of* (*hd* (*get-trail T'*))›
‹*mark-of* (*hd* (*get-trail T'*))›] *dec struct-invs count-dec tr proped TT'*
by (*cases* ‹*get-trail T'*›; *cases* ‹*hd* (*get-trail T'*)›)
(*auto simp*: *twl-st get-unit-clauses-l-alt-def*)
moreover have ‹*convert-lit* (*get-clauses-l T*) (*get-kept-unit-clauses-l T*) (*hd* (*get-trail-l T*))
(*hd* (*get-trail T'*))›
using *tr dec TT'*
by (*cases* ‹*get-trail T'*›; *cases* ‹*get-trail-l T*›)
(*auto simp*: *twl-st-l-def*)
ultimately have ‹*mark-of* (*hd* (*get-trail-l T*)) = *0* ⟹ *False*›
using *tr dec TT'* **apply** (*cases* ‹*get-trail-l T*›; *cases* ‹*hd* (*get-trail-l T*)›)
by (*auto simp*: *trail.simps twl-st convert-lit.simps*)
then show *?ge* **by** *blast*
show ‹*get-trail-l T* ≠ []› ‹*get-trail* (*snd brkT'*) ≠ []›
using *tr TT' brkT'* **by** *auto*
qed

have *mop-hd-trail-l*:
‹*mop-hd-trail-l S* ≤ ⇓{((*L*, *C*), (*L'*, *C'*)). *L* = *L'* ∧
C' = *mset* (*get-clauses-l S* ∝ *C*) ∧ *C* ∈# *dom-m* (*get-clauses-l S*) ∧ *get-trail-l S* ≠ [] ∧
hd (*get-trail-l S*) = *Propagated L C* ∧ *C* > *0*} (*mop-hd-trail S'*)›
(**is** ‹- ≤ ⇓ *?hd* -›)
if ‹(*brkS*, *brkS'*) ∈ *bool-rel* ×$_f$ *?R'*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ *skip-and-resolve-loop-inv-l S*$_0$ *brk S*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ ¬*brk* ∧ ¬ *is-decided* (*hd* (*get-trail-l S*))› **and**
‹*brkS* = (*brk*, *S*)› **and**
‹*brkS'* = (*brk'*, *S'*)›
for *S S' S*$_0$ *brk brkS brkS' brk'*
using *that*
unfolding *mop-hd-trail-l-def mop-hd-trail-def*
apply *refine-rcg*
subgoal unfolding *mop-hd-trail-l-pre-def skip-and-resolve-loop-inv-l-def*

mop-hd-trail-def
by (*rule exI*[*of* - *S'*]) *auto*
subgoal
apply (*rule RES-refine*)
using *mark-ge-0*[*of S S' brkS brkS'*]
apply (*cases* ‹*get-trail S'*›; *cases* ‹*get-trail-l S*›;
cases ‹*hd* (*get-trail S'*)›)
by (*auto simp*: *mop-hd-trail-pre-def twl-st-l-def*
convert-lit.simps mop-hd-trail-l-pre-def)
done

have *mop-lit-notin-conflict-l*:
‹*mop-lit-notin-conflict-l* (−*L*) *S* ≤ ⇓*Id* (*mop-lit-notin-conflict* (−*L'*) *S'*)›
if ‹(*brkS*, *brkS'*) ∈ *bool-rel* ×$_f$ *?R'*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ *skip-and-resolve-loop-inv-l* S_0 *brk S*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ ¬*brk* ∧ ¬ *is-decided* (*hd* (*get-trail-l S*))› **and**
‹*brkS* = (*brk*, *S*)› **and**
‹*brkS'* = (*brk'*, *S'*)› **and**
‹*LC* = (*L*, *C*)› **and**
‹*LC'* = (*L'*, *C'*)› **and**
‹((*LC*), (*LC'*)) ∈ *?hd S*›
for *S S' S*$_0$ *brk brkS brkS' brk' L L' C C' LC LC'*
using *that*
unfolding *mop-lit-notin-conflict-l-def mop-lit-notin-conflict-def*
apply *refine-rcg*
subgoal
by *auto*
subgoal
by *auto*
subgoal
by (*auto simp*: *mset-take-mset-drop-mset'*)
subgoal
by *auto*
done

have *mop-maximum-level-removed-l*:
‹*mop-maximum-level-removed-l L S* ≤ ⇓*bool-rel* (*mop-maximum-level-removed L' S'*)›
if ‹(*brkS*, *brkS'*) ∈ *bool-rel* ×$_f$ *?R'*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ *skip-and-resolve-loop-inv-l* S_0 *brk S*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ ¬*brk* ∧ ¬ *is-decided* (*hd* (*get-trail-l S*))› **and**
‹*brkS* = (*brk*, *S*)› **and**
‹*brkS'* = (*brk'*, *S'*)› **and**
‹*LC* = (*L*, *C*)› **and**
‹*LC'* = (*L'*, *C'*)› **and**
‹((*LC*), (*LC'*)) ∈ *?hd S*›
for *S S' S*$_0$ *brk brkS brkS' brk' L L' C C' LC LC'*
using *that*
unfolding *mop-maximum-level-removed-l-def mop-maximum-level-removed-def*
apply *refine-rcg*
subgoal
unfolding *mop-maximum-level-removed-l-pre-def*
by (*rule exI*[*of* - ‹*S'*›]) *auto*
subgoal
by *auto*
done

have *mop-tl-state-l*:
‹*mop-tl-state-l S* ≤ ⇓ (*bool-rel* ×$_f$ *?R'*) (*mop-tl-state S'*)›
if ‹(*brkS*, *brkS'*) ∈ *bool-rel* ×$_f$ *?R'*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ *skip-and-resolve-loop-inv-l S*$_0$ *brk S*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ ¬*brk* ∧ ¬ *is-decided* (*hd* (*get-trail-l S*))› **and**
‹*brkS* = (*brk*, *S*)› **and**
‹*brkS'* = (*brk'*, *S'*)› **and**
‹*LC* = (*L*, *C*)› **and**
‹*LC'* = (*L'*, *C'*)› **and**
‹((*LC*), (*LC'*)) ∈ *?hd S*›
for *S S' S*$_0$ *brk brkS brkS' brk' L L' C C' LC LC'*
using *that* **unfolding** *mop-tl-state-l-def mop-tl-state-def*
apply *refine-rcg*
subgoal
unfolding *mop-tl-state-l-pre-def*
by (*rule exI*[*of* - ‹*S'*›]) *auto*
subgoal
by (*cases S*; *cases S'*; *cases* ‹*get-trail-l S*›; *cases* ‹*get-trail S'*›)
(*auto simp*: *tl-state-l-def tl-state-def twl-st-l-def*
convert-lits-l-tlD mop-tl-state-pre-def twl-list-invs-def)
done

have *mop-update-confl-tl-l*:
‹*mop-update-confl-tl-l L C S* ≤ ⇓ (*bool-rel* ×$_f$ *?R'*) (*mop-update-confl-tl L' C' S'*)›
if ‹(*brkS*, *brkS'*) ∈ *bool-rel* ×$_f$ *?R'*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ *skip-and-resolve-loop-inv-l S*$_0$ *brk S*› **and**
‹*case brkS of* (*brk*, *S*) ⇒ ¬*brk* ∧ ¬ *is-decided* (*hd* (*get-trail-l S*))› **and**
‹*brkS* = (*brk*, *S*)› **and**
‹*brkS'* = (*brk'*, *S'*)› **and**
‹*LC* = (*L*, *C*)› **and**
‹*LC'* = (*L'*, *C'*)› **and**
‹((*LC*), (*LC'*)) ∈ *?hd S*›
for *S S' S*$_0$ *brk brkS brkS' brk' L L' C C' LC LC'*
using *that* **unfolding** *mop-update-confl-tl-l-def mop-update-confl-tl-def*
apply *refine-rcg*
subgoal
unfolding *update-confl-tl-l-pre-def*
by (*rule exI*[*of* - ‹*S'*›]) (*auto simp*: *twl-struct-invs-def*)
subgoal
by (*cases S*; *cases S'*; *cases* ‹*get-trail-l S*›; *cases* ‹*get-trail S'*›)
(*auto simp*: *update-confl-tl-l-def update-confl-tl-def twl-st-l-def*
convert-lits-l-tlD mop-tl-state-pre-def twl-list-invs-def resolve-cls-l'-def)
done

have *H*:
‹(*skip-and-resolve-loop-l*, *skip-and-resolve-loop*) ∈ *?R* →$_f$
⟨{(*T*::'*v twl-st-l*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs T* ∧
clauses-to-update-l T = {#}}⟩ *nres-rel*›
apply (*intro frefI nres-relI*)
unfolding *skip-and-resolve-loop-l-def skip-and-resolve-loop-def*
apply (*refine-rcg H*)
subgoal by *auto*
subgoal by *auto*
subgoal for *x y xa x' x1 x2*
unfolding *skip-and-resolve-loop-inv-l-def*

```
    apply (cases x')
    apply(rule exI[of - ‹snd x'›])
    apply(rule exI[of - y])
    apply (intro conjI impI)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by (rule mark-ge-0) auto
    done
  subgoal by (auto simp: twl-st-l twl-st skip-and-resolve-loop-inv-def)
  apply (rule mop-hd-trail-l; assumption)
  apply (rule mop-lit-notin-conflict-l; assumption)
  subgoal by auto
  apply (rule mop-tl-state-l; assumption)
  apply (rule mop-maximum-level-removed-l; assumption)
  subgoal by auto
  apply (rule mop-update-confl-tl-l; assumption)
  subgoal by auto
  subgoal by auto
  done

 have H: ‹(skip-and-resolve-loop-l, skip-and-resolve-loop)
  ∈ ?R →f
    ⟨{(T::'v twl-st-l, T').
      (T, T') ∈ {(T, T'). (T, T') ∈ twl-st-l None ∧ (twl-list-invs T ∧
      clauses-to-update-l T = {#})} ∧
      T' ∈ {T'. twl-struct-invs T' ∧ twl-stgy-invs T' ∧
      (no-step cdcl_W-restart-mset.skip (state_W-of T')) ∧
      (no-step cdcl_W-restart-mset.resolve (state_W-of T')) ∧
      literals-to-update T' = {#} ∧
      get-conflict T' ≠ None}}⟩nres-rel›
  apply (rule refine-add-inv-generalised)
  subgoal by (rule H)
  subgoal for S S'
    apply (rule order-trans)
    apply (rule skip-and-resolve-loop-spec[of S'])
    by auto
  done
 show ?thesis
  using H apply −
  apply (match-spec; (match-fun-rel; match-fun-rel?)+)
  by blast+
qed

definition find-decomp :: ‹'v literal ⇒ 'v twl-st-l ⇒ 'v twl-st-l nres› where
 ‹find-decomp =  (λL (M, N, D, NE, NEk, UEk, UE, NS, US, WS, Q).
  SPEC(λS. ∃K M2 M1. S = (M1, N, D, NE, NEk, UEk, UE, NS, US, WS, Q) ∧
    (Decided K # M1, M2) ∈ set (get-all-ann-decomposition M) ∧
      get-level M K = get-maximum-level M (the D − {#−L#}) + 1))›

lemma find-decomp-alt-def:
 ‹find-decomp L S =
   SPEC(λT. ∃K M2 M1. equality-except-trail S T ∧ get-trail-l T = M1 ∧
    (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (get-trail-l S)) ∧
```

get-level (*get-trail-l S*) *K* =
get-maximum-level (*get-trail-l S*) (*the* (*get-conflict-l S*) − {#−*L*#}) + *1*)›
unfolding *find-decomp-def*
by (*cases S*) *force*

definition *find-lit-of-max-level-l* :: ‹'*v twl-st-l* ⇒ '*v literal* ⇒ '*v literal nres*› **where**
‹*find-lit-of-max-level-l* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *WS*, *Q*) *L*.
SPEC(λ*L'*. *L'* ∈# *the D* − {#−*L*#} ∧ *get-level M L'* = *get-maximum-level M* (*the D* − {#−*L*#})))›

lemma *find-lit-of-max-level-l-alt-def*:
‹*find-lit-of-max-level-l* = (λ*S L*.
SPEC(λ*L'*. *L'* ∈# *the* (*get-conflict-l S*) − {#−*L*#} ∧
get-level (*get-trail-l S*) *L'* = *get-maximum-level* (*get-trail-l S*) (*the* (*get-conflict-l S*) − {#−*L*#})))›
by (*auto simp*: *find-lit-of-max-level-l-def intro*!: *ext*)

definition *ex-decomp-of-max-lvl* :: ‹('*v*, *nat*) *ann-lits* ⇒ '*v cconflict* ⇒ '*v literal* ⇒ *bool*› **where**
‹*ex-decomp-of-max-lvl M D L* ⟷
(∃ *K M1 M2*. (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
get-level M K = *get-maximum-level M* (*remove1-mset* (−*L*) (*the D*)) + *1*)›

fun *add-mset-list* :: ‹'*a list* ⇒ '*a multiset multiset* ⇒ '*a multiset multiset*› **where**
‹*add-mset-list L UE* = *add-mset* (*mset L*) *UE*›

definition (**in** −)*list-of-mset* :: ‹'*v clause* ⇒ '*v clause-l nres*› **where**
‹*list-of-mset D* = *SPEC*(λ*D'*. *D* = *mset D'*)›

definition *extract-shorter-conflict-l-pre* :: ‹'*v twl-st-l* ⇒ *bool*› **where**
‹*extract-shorter-conflict-l-pre S* ⟷ (∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧
extract-shorter-conflict-pre S')›

fun *extract-shorter-conflict-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l nres*›
where
‹*extract-shorter-conflict-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) = *do* {
ASSERT(*extract-shorter-conflict-l-pre* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
SPEC(λ*S*.
∃ *D'*. *D'* ⊆# *the D* ∧ *S* = (*M*, *N*, *Some D'*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ∧
clause ‘# *twl-clause-of* ‘# *ran-mf N* + (*NE* + *NEk*) + (*UE* + *UEk*) + *NS* + *US* + *N0* + *U0*
⊨*pm D'* ∧ −(*lit-of* (*hd M*)) ∈# *D'*)
}›

declare *extract-shorter-conflict-l.simps*[*simp del*]
lemmas *extract-shorter-conflict-l-def* = *extract-shorter-conflict-l.simps*

lemma *extract-shorter-conflict-l-alt-def*:
‹*extract-shorter-conflict-l S* = *do* {
ASSERT(*extract-shorter-conflict-l-pre S*);
SPEC(λ*T*.
∃ *D'*. *D'* ⊆# *the* (*get-conflict-l S*) ∧ *equality-except-conflict-l S T* ∧
get-conflict-l T = *Some D'* ∧
get-all-clss-l S ⊨*pm D'* ∧
−*lit-of* (*hd* (*get-trail-l S*)) ∈# *D'*) }›
by (*cases S*) (*auto simp*: *extract-shorter-conflict-l-def*
mset-take-mset-drop-mset mset-take-mset-drop-mset' Un-assoc
intro: *bind-cong*)

definition *backtrack-l-inv* **where**

```
  ‹backtrack-l-inv S ⟷
    (∃S'. (S, S') ∈ twl-st-l None ∧
      backtrack-inv S' ∧
      twl-list-invs S ∧
      literals-to-update-l S = {#})
  ›

definition get-fresh-index :: ‹'v clauses-l ⇒ nat nres› where
‹get-fresh-index N = SPEC(λi. i > 0 ∧ i ∉# dom-m N)›

definition propagate-bt-l-pre where
  ‹propagate-bt-l-pre L L' S ⟷
    (∃S'. (S, S') ∈ twl-st-l None ∧ propagate-bt-pre L L' S')›

definition propagate-bt-l :: ‹'v literal ⇒ 'v literal ⇒ 'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹propagate-bt-l = (λL L' (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q). do {
    ASSERT(propagate-bt-l-pre L L' (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q));
    ASSERT(L ∈# all-lits-of-st-l (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q));
    ASSERT(L' ∈# all-lits-of-st-l (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q));
    D'' ← list-of-mset (the D);
    i ← get-fresh-index N;
    M ← cons-trail-propagate-l (−L) i M;
    RETURN (M,
       fmupd i ([−L, L'] @ (remove1 (−L) (remove1 L' D'')), False) N,
         None, NE, UE, NEk, UEk, NS, US, N0, U0, WS, {#L#})
     })›

definition propagate-unit-bt-l-pre where
  ‹propagate-unit-bt-l-pre L S ⟷
     (∃S'. (S, S') ∈ twl-st-l None ∧ propagate-unit-bt-pre L S')›

definition propagate-unit-bt-l :: ‹'v literal ⇒ 'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹propagate-unit-bt-l = (λL (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q). do {
    ASSERT(L ∈# all-lits-of-st-l (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q));
    ASSERT(propagate-unit-bt-l-pre L (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q));
    M ← cons-trail-propagate-l (−L) 0 M;
    RETURN (M, N, None, NE, UE, NEk, add-mset (the D) UEk, NS, US, N0, U0, WS, {#L#})})›

definition mop-lit-hd-trail-l-pre :: ‹'v twl-st-l ⇒ bool› where
‹mop-lit-hd-trail-l-pre S ⟷
  (∃S'. (S, S') ∈ twl-st-l None ∧ mop-lit-hd-trail-pre S' ∧
     twl-list-invs S)›

definition mop-lit-hd-trail-l :: ‹'v twl-st-l ⇒ ('v literal) nres› where
  ‹mop-lit-hd-trail-l S = do{
     ASSERT(mop-lit-hd-trail-l-pre S);
     SPEC(λL. L = lit-of (hd (get-trail-l S)))
  }›

definition backtrack-l :: ‹'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹backtrack-l S =
    do {
      ASSERT(backtrack-l-inv S);
      L ← mop-lit-hd-trail-l S;
      S ← extract-shorter-conflict-l S;
      S ← find-decomp L S;
```

```
    if size (the (get-conflict-l S)) > 1
    then do {
      L' ← find-lit-of-max-level-l S L;
      propagate-bt-l L L' S
    }
    else do {
      propagate-unit-bt-l L S
    }
  }›

lemma backtrack-l-spec:
  ‹(backtrack-l, backtrack) ∈
    {(S::'v twl-st-l, S'). (S, S') ∈ twl-st-l None ∧ get-conflict-l S ≠ None ∧
      get-conflict-l S ≠ Some {#} ∧
      clauses-to-update-l S = {#} ∧ literals-to-update-l S = {#} ∧ twl-list-invs S ∧
      twl-struct-invs S' ∧ twl-stgy-invs S' ∧
      no-step cdcl_W-restart-mset.skip (state_W-of S') ∧
      no-step cdcl_W-restart-mset.resolve (state_W-of S')} →_f
    ⟨{(T::'v twl-st-l, T'). (T, T') ∈ twl-st-l None ∧ get-conflict-l T = None ∧ twl-list-invs T ∧
      twl-struct-invs T' ∧ twl-stgy-invs T' ∧ clauses-to-update-l T = {#} ∧
      literals-to-update-l T ≠ {#}}⟩ nres-rel›
  (is ‹ - ∈ ?R →_f ?I›)
proof −
  let ?J = ‹{(T::'v twl-st-l, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T ∧
      clauses-to-update-l T = {#} ∧ literals-to-update-l T ≠ {#}}›
  let ?J' = ‹{(T::'v twl-st-l, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T ∧
      clauses-to-update-l T = {#} ∧ literals-to-update-l T = {#}}›
  have mop-lit-hd-trail-l:
    ‹mop-lit-hd-trail-l x ≤ ⇓ Id (mop-lit-hd-trail y)›
    if ‹(x,y) ∈ ?R›
    for x y
    using that
    unfolding mop-lit-hd-trail-l-def mop-lit-hd-trail-def
    apply refine-vcg
    subgoal
      unfolding mop-lit-hd-trail-l-pre-def
      by (rule exI[of - y]) auto
    subgoal by (auto simp: mop-lit-hd-trail-pre-def)
    done
  have [simp]: ‹(λx. mset (fst x)) ‘ {a. a ∈# ran-m aa ∧ snd a} ∪
      (λx. mset (fst x)) ‘ {a. a ∈# ran-m aa ∧ ¬ snd a} ∪ A =
     (λx. mset (fst x)) ‘ {a. a ∈# ran-m aa}∪ A› for aa A
    by auto
  have get-all-clss-alt-def:
    ‹get-all-clss y = clauses (get-clauses y) + unit-clss y + subsumed-clauses y + get-all-clauses0 y› for
y
    by (cases y) auto
  have entailement-cong:‹(x,y) ∈ ?R ⟹ (get-all-clss-l x) ⊨pm D ⟷
    (get-all-clss y) ⊨pm D› for x y D
    by auto
  have extract-shorter-conflict-l: ‹extract-shorter-conflict-l x
      ≤ ⇓ ?J'
          (extract-shorter-conflict y)›
    if ‹(x,y) ∈ ?R›
    for x y
```

using *that*
unfolding *extract-shorter-conflict-l-alt-def extract-shorter-conflict-alt-def*
apply *refine-vcg*
subgoal
unfolding *extract-shorter-conflict-l-pre-def*
by (*rule exI*[*of - y*]) *auto*
subgoal
unfolding *get-all-clss-alt-def*[*symmetric*]
apply (*subst entailement-cong*, *assumption*)
by (*rule RES-refine*, *rule-tac x* = ‹*set-conflict* (*the* (*get-conflict-l s*)) *y*› **in** *bexI*)
(*clarsimp simp*: *twl-st-l-def image-image image-Un mset-take-mset-drop-mset′*
extract-shorter-conflict-pre-def twl-list-invs-def
simp del: *get-all-clss.simps*)+
done

have *find-decomp*: ‹(*L*, *La*) ∈ *Id* ⟹
(*S*, *Sa*) ∈ *?J′* ⟹
find-decomp L S
≤ ⇓ *?J′* (*reduce-trail-bt La Sa*)› **for** *L La S Sa*
unfolding *find-decomp-def reduce-trail-bt-def*
RES-RETURN-RES
apply *refine-vcg*
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f*
x1g x2g x1h x2h x1i x2i x1j x2j x1k x2k x1l x2l
apply (*rule RES-refine*)
apply (*rule-tac x*=‹(*drop* (*length* (*get-trail Sa*) − *length* (*get-trail-l s*)) (*get-trail Sa*),
x1a, *x1b*, *x1c*, *x1d*, *x1e*, *x1f*, *x2f*)› **in** *bexI*)
apply (*auto simp*: *twl-st-l-def image-iff twl-list-invs-def*
intro: *convert-lits-l-decomp-ex*)
apply (*rule-tac x*=*K* **in** *exI*)
apply (*auto simp*: *twl-st-l-def image-image image-Un mset-take-mset-drop-mset′*
extract-shorter-conflict-pre-def convert-lits-l-decomp-ex)
done
done

have *find-lit-of-max-level-l*: ‹(*L*, *La*) ∈ *Id* ⟹
(*S*, *Sa*) ∈ *?J′* ⟹
(*Sb*, *Sc*) ∈ *?J′* ⟹
find-lit-of-max-level-l Sb L
≤ ⇓ *Id* (*find-lit-of-max-level Sc La*)›
for *L La S Sa Sb Sc*
unfolding *find-lit-of-max-level-l-alt-def*
find-lit-of-max-level-def
by *refine-rcg auto*

have [*intro*!]: ‹*mset x* =
add-mset (*La*) (*add-mset L′a* (*mset x* − {#*La*, *L′a*#}))›
if ‹*La* ∈ *set x*› **and** ‹*L′a* ∈ *set x*› ‹*La* ≠ *L′a*› **for** *x La L′a*
using *that*
by (*metis add-mset-add-single diff-diff-add-mset in-multiset-in-set mset-add*
remove1-mset-add-mset-If)
have *propagate-bt-l-preD*:
‹ ¬*tautology* (*add-mset* (− *L*) (*add-mset L′* (*the* (*get-conflict-l S*) − {#− *L*, *L′*#})))›
if *pre*: ‹*propagate-bt-l-pre L L′ S*› **for** *L L′ S*
proof −
obtain *M N U D NE UE NEk UEk NS US WS Q M′ D′ T* **where**

```
    T: ‹T = (M, N, U, Some D, NE, UE, NEk, UEk, NS, US, WS, Q)› and
    struct: ‹twl-struct-invs (M' @ M, N, U, Some D', NE, UE, NEk, UEk, NS, US, WS, Q)› and
    ST: ‹(S, T) ∈ twl-st-l None› and
    DD': ‹D ⊆# D'› and
    LL': ‹− L ∈# D ∧ L' ∈# remove1-mset (− L) D›
    using pre that
    unfolding propagate-bt-l-pre-def propagate-bt-pre-def
    by auto
  have ‹M' @ M ⊨as CNot D'› and ‹no-dup (M' @ M)›
    using struct unfolding
      pcdcl-all-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-conflicting-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def
      by simp-all
  then have ‹¬tautology D'›
    using consistent-CNot-not-tautology[of ‹lits-of-l (M'@M)› D']
    by (auto simp: true-annots-true-cls dest: distinct-consistent-interp)
   then show ‹?thesis›
     using ST DD' LL' by (simp only: twl-st-l[symmetric], auto  dest!: multi-member-split
       simp add: T not-tautology-mono)
 qed
have propagate-bt-l: ‹(Sb, Sc) ∈ ?J' ⟹
    (L', L'a) ∈ Id ⟹
    (L, La) ∈ Id ⟹
    propagate-bt-l L L' Sb
    ≤ ⇓ ?J
        (mop-propagate-bt La L'a Sc)›
  for L La S Sa Sb Sc L' L'a
  unfolding propagate-bt-l-def
   mop-propagate-bt-def list-of-mset-def
   get-fresh-index-def cons-trail-propagate-l-def
  apply refine-vcg
  subgoal unfolding propagate-bt-l-pre-def by fast
  subgoal by (auto simp: propagate-bt-pre-def twl-st-l-def all-lits-of-st-def
    all-lits-of-st-l-def mset-take-mset-drop-mset' simp flip: image-mset-union) (simp add: ac-simps)
  subgoal by (auto simp: propagate-bt-pre-def twl-st-l-def all-lits-of-st-def
    all-lits-of-st-l-def mset-take-mset-drop-mset' simp flip: image-mset-union) (simp add: ac-simps)
  subgoal by (auto simp: propagate-bt-pre-def twl-st-l-def
     mset-take-mset-drop-mset' simp flip: image-mset-union)
  subgoal
    apply (frule propagate-bt-l-preD)
    by (clarsimp simp add: twl-list-invs-def propagate-bt-def twl-st-l-def
          propagate-bt-pre-def init-clss-l-mapsto-upd-irrel-notin ran-m-mapsto-upd-notin
          learned-clss-l-mapsto-upd-notin)
     (auto 4 3 simp: twl-list-invs-def propagate-bt-def twl-st-l-def
          propagate-bt-pre-def init-clss-l-mapsto-upd-irrel-notin
          learned-clss-l-mapsto-upd-notin
        intro!: convert-lit.intros(2)
      intro!: convert-lits-l-bind-new
      dest: in-diffD)
  done


have propagate-unit-bt-l: ‹(Sb, Sc) ∈ ?J' ⟹
    (L, La) ∈ Id ⟹
    propagate-unit-bt-l L Sb
```

```
    ≤ ⇓ ?J
        (mop-propagate-unit-bt La Sc)›
  for L La S Sa Sb Sc L' L'a
  unfolding propagate-unit-bt-l-def
   mop-propagate-unit-bt-def cons-trail-propagate-l-def
  apply refine-vcg
  subgoal by (auto simp: propagate-unit-bt-pre-def twl-st-l-def all-lits-of-st-def
    all-lits-of-st-l-def
     mset-take-mset-drop-mset' simp flip: image-mset-union) (simp add: ac-simps)
  subgoal
    unfolding propagate-unit-bt-l-pre-def
    by blast
  subgoal by (auto simp: propagate-unit-bt-pre-def twl-st-l-def
     mset-take-mset-drop-mset' simp flip: image-mset-union)
  subgoal by (auto simp: twl-list-invs-def propagate-unit-bt-def twl-st-l-def
    convert-lits-l-add-mset intro!: convert-lit.intros)
  done

have bt: ‹(backtrack-l, backtrack) ∈
    ?R →f ⟨?J⟩ nres-rel›
  unfolding backtrack-l-def backtrack-def
  apply (intro frefI nres-relI)
  apply (refine-rcg mop-lit-hd-trail-l)
  subgoal for x y
    unfolding backtrack-l-inv-def
    by (rule exI[of - y]) auto
  apply (rule extract-shorter-conflict-l; assumption)
  apply (rule find-decomp; assumption)
  subgoal by auto
  apply (rule find-lit-of-max-level-l; assumption)
  apply (rule propagate-bt-l; assumption)
  apply (rule propagate-unit-bt-l; assumption)
  done

have SPEC-Id: ‹SPEC Φ = ⇓ {(T, T'). Φ T} (SPEC Φ)› for Φ
  unfolding conc-fun-RES
  by auto
have ‹(backtrack-l S, backtrack S') ∈ ?I› if ‹(S, S') ∈ ?R› for S S'
proof −
  have ‹backtrack-l S ≤ ⇓ ?J (backtrack S')›
    by (rule bt[unfolded fref-param1[symmetric], to-⇓, rule-format, of S S'])
      (use that in auto)
  moreover have ‹backtrack S' ≤ SPEC (λT. cdcl-twl-o S' T ∧
          get-conflict T = None ∧
          (∀ S'. ¬ cdcl-twl-o T S') ∧
          twl-struct-invs T ∧
          twl-stgy-invs T ∧ clauses-to-update T = {#} ∧ literals-to-update T ≠ {#})›
    by (rule backtrack-spec[to-⇓, of S']) (use that in ‹auto simp: twl-st-l›)
  ultimately show ?thesis
    apply −
    apply (unfold refine-rel-defs nres-rel-def in-pair-collect-simp;
       (unfold Ball2-split-def all-to-meta)?;
       (intro allI impI)?)
    apply (subst (asm) SPEC-Id)
    apply unify-Down-invs2+
    unfolding nofail-simps
```

apply *unify-Down-invs2-normalisation-post*
apply (*rule weaken-⇓*)
prefer *2* **apply** *assumption*
subgoal premises *p* **by** (*auto simp*: *twl-st-l-def*)
done
qed
then show *?thesis*
by (*intro frefI*)
qed

definition *find-unassigned-lit-l* :: ‹*'v twl-st-l* ⇒ *'v literal option nres*› **where**
‹*find-unassigned-lit-l* = (λ*S*.
SPEC (λ*L*.
(*L* ≠ *None* ⟶
undefined-lit (*get-trail-l S*) (*the L*) ∧
(*the L*) ∈# *all-lits-of-st-l S*) ∧
(*L* = *None* ⟶ (∄ *L'*. *undefined-lit* (*get-trail-l S*) *L'* ∧
L' ∈# *all-lits-of-st-l S*)))
)›

definition *decide-l-or-skip-pre* **where**
‹*decide-l-or-skip-pre S* ⟷ (∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧
decide-or-skip-pre S' ∧
twl-list-invs S ∧
clauses-to-update-l S = {#} ∧
literals-to-update-l S = {#})
›

definition *decide-lit-l* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*decide-lit-l* = (λ*L'* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*).
(*Decided L'* # *M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, {#− *L'*#}))›

definition *decide-l-or-skip* :: ‹*'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres*› **where**
‹*decide-l-or-skip S* = (*do* {
ASSERT(*decide-l-or-skip-pre S*);
L ← *find-unassigned-lit-l S*;
case L of
None ⇒ *RETURN* (*True*, *S*)
| *Some L* ⇒ *RETURN* (*False*, *decide-lit-l L S*)
})
›
method *match-⇓* =
(*match* **conclusion in** ‹*f* ≤ ⇓ *R g*› **for** *f* :: ‹*'a nres*› **and** *R* :: ‹(*'a* × *'b*) *set*› **and**
g :: ‹*'b nres*› ⇒
‹*match premises in*
I[*thin*,*uncurry*]: ‹*f* ≤ ⇓ *R' g*› *for R'* :: ‹(*'a* × *'b*) *set*›
⇒ ‹*rule refinement-trans-long*[*of f f g g R' R*, *OF refl refl - I*]›
| *I*[*thin*,*uncurry*]: ‹- ⟹ *f* ≤ ⇓ *R' g*› *for R'* :: ‹(*'a* × *'b*) *set*›
⇒ ‹*rule refinement-trans-long*[*of f f g g R' R*, *OF refl refl - I*]›
›)

lemma *decide-l-or-skip-spec*:
‹(*decide-l-or-skip*, *decide-or-skip*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *get-conflict-l S* = *None* ∧
clauses-to-update-l S = {#} ∧ *literals-to-update-l S* = {#} ∧ *no-step cdcl-twl-cp S'* ∧

```
      twl-struct-invs S' ∧ twl-stgy-invs S' ∧ twl-list-invs S} →_f
    ⟨{((brk, T), (brk', T')). (T, T') ∈ twl-st-l None ∧ brk = brk' ∧ twl-list-invs T ∧
      clauses-to-update-l T = {#} ∧
      (get-conflict-l T ≠ None ⟶ get-conflict-l T = Some {#})∧
        twl-struct-invs T' ∧ twl-stgy-invs T' ∧
        (¬brk ⟶ literals-to-update-l T ≠ {#})∧
        (brk ⟶ literals-to-update-l T = {#})}⟩ nres-rel›
  (is ‹- ∈ ?R →_f ⟨?S⟩nres-rel›)
proof −
  have find-unassigned-lit-l: ‹find-unassigned-lit-l S ≤ ⇓ Id (find-unassigned-lit S')›
    if SS': ‹(S, S') ∈ ?R›
    for S S'
    using that
    unfolding find-unassigned-lit-l-def find-unassigned-lit-def state-decomp-to-state-l
    by (auto simp:)

  have I: ‹(x, x') ∈ Id ⟹ (x, x') ∈ ⟨Id⟩option-rel› for x x' by auto
  have dec: ‹(decide-l-or-skip, decide-or-skip) ∈ ?R →
    ⟨{((brk, T), (brk', T')). (T, T') ∈ twl-st-l None ∧ brk = brk' ∧ twl-list-invs T ∧
      clauses-to-update-l T = {#} ∧
       (¬brk ⟶ literals-to-update-l T ≠ {#})∧
       (brk ⟶ literals-to-update-l T = {#}) }⟩ nres-rel›
    unfolding decide-l-or-skip-def decide-or-skip-def
    apply (refine-vcg find-unassigned-lit-l I)
    subgoal unfolding decide-l-or-skip-pre-def by (auto simp: twl-st-l-def)
    subgoal by auto
    subgoal for S S'
      by (cases S)
       (auto simp: decide-lit-l-def propagate-dec-def twl-list-invs-def twl-st-l-def)
    done
  have KK: ‹SPEC (λ(brk, T). cdcl-twl-o** S' T ∧ P brk T) = ⇓ {(S, S'). snd S = S' ∧
     P (fst S) (snd S)} (SPEC (cdcl-twl-o** S'))›
    for S' P
    by (auto simp: conc-fun-def)
  have nf: ‹nofail (SPEC (cdcl-twl-o** S'))› ‹nofail (SPEC (cdcl-twl-o** S'))› for S S'
    by auto
  have set: ‹{((a,b), (a', b')). P a b a' b'} = {(a, b). P (fst a) (snd a) (fst b) (snd b)}› for P
    by auto

  show ?thesis
  proof (intro frefI nres-relI)
    fix S S'
    assume SS': ‹(S, S') ∈ ?R›
    have ‹decide-l-or-skip S
    ≤ ⇓ {((brk, T), brk', T').
         (T, T') ∈ twl-st-l None ∧
         brk = brk' ∧
         twl-list-invs T ∧
         clauses-to-update-l T = {#} ∧
         (¬ brk ⟶ literals-to-update-l T ≠ {#}) ∧ (brk ⟶ literals-to-update-l T = {#})}
       (decide-or-skip S')›
      apply (rule dec[to-⇓, of S S'])
      using SS' by auto
    moreover have ‹decide-or-skip S'
    ≤ ⇓ {(S, S'a).
         snd S = S'a ∧
```

```
        get-conflict (snd S) = None ∧
        (∀ S'. ¬ cdcl-twl-o (snd S) S') ∧
        (fst S ⟶ (∀ S'. ¬ cdcl-twl-stgy (snd S) S')) ∧
        twl-struct-invs (snd S) ∧
        twl-stgy-invs (snd S) ∧
        clauses-to-update (snd S) = {#} ∧
        (¬ fst S ⟶ literals-to-update (snd S) ≠ {#}) ∧
        (¬ (∀ S'a. ¬ cdcl-twl-o S' S'a) ⟶ cdcl-twl-o++ S' (snd S))}
      (SPEC (cdcl-twl-o** S'))›
    by (rule decide-or-skip-spec[of S', unfolded KK]) (use SS' in auto)
  ultimately show ‹decide-l-or-skip S ≤ ⇓ ?S (decide-or-skip S')›
    apply −
    apply unify-Down-invs2+
    apply (simp only: set nf)
    apply (match-⇓)
    subgoal
      apply (rule; rule)
      apply (clarsimp simp: twl-st-l-def)
      done
    subgoal by fast
    done
 qed
qed

lemma refinement-trans-eq:
 ‹A = A' ⟹ B = B' ⟹ R' = R ⟹ A ≤ ⇓ R B ⟹ A' ≤ ⇓ R' B'›
 by (auto simp: pw-ref-iff)

definition cdcl-twl-o-prog-l-pre where
 ‹cdcl-twl-o-prog-l-pre S ⟷
 (∃ S' . (S, S') ∈ twl-st-l None ∧
    twl-struct-invs S' ∧
    twl-stgy-invs S' ∧
    twl-list-invs S)›

definition cdcl-twl-o-prog-l :: ‹'v twl-st-l ⇒ (bool × 'v twl-st-l) nres› where
 ‹cdcl-twl-o-prog-l S =
   do {
     ASSERT(cdcl-twl-o-prog-l-pre S);
     do {
       if get-conflict-l S = None
       then decide-l-or-skip S
       else if count-decided (get-trail-l S) > 0
       then do {
         T ← skip-and-resolve-loop-l S;
         ASSERT(get-conflict-l T ≠ None ∧ get-conflict-l T ≠ Some {#});
         U ← backtrack-l T;
         RETURN (False, U)
       }
       else RETURN (True, S)
     }
   }
 ›


lemma twl-st-lE:
```

‹(⋀M N D NE UE WS Q. T = (M, N, D, NE, UE, WS, Q) ⟹ P (M, N, D, NE, UE, WS, Q)) ⟹ P T›
for T :: ‹'a twl-st-l›
by (*cases T*) *auto*

lemma *weaken-⇓'*: ‹f ≤ ⇓ R' g ⟹ R' ⊆ R ⟹ f ≤ ⇓ R g›
by (*meson pw-ref-iff subset-eq*)

lemma *cdcl-twl-o-prog-l-spec*:
‹(*cdcl-twl-o-prog-l*, *cdcl-twl-o-prog*) ∈
{(S, S'). (S, S') ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} →$_f$
⟨{((*brk*, T), (*brk'*, T')). (T, T') ∈ *twl-st-l None* ∧ *brk* = *brk'* ∧ *twl-list-invs T* ∧
clauses-to-update-l T = {#}}⟩ *nres-rel*›
(**is** ‹ - ∈ *?R* →$_f$ *?I*› **is** ‹ - ∈ *?R* →$_f$ ⟨*?J*⟩*nres-rel*›)
proof −
show *?thesis*
(**is** ‹- ∈ - →$_f$ ⟨*?I'*⟩ *nres-rel*›)
supply [[*goals-limit*=3]]
unfolding *cdcl-twl-o-prog-l-def cdcl-twl-o-prog-def*
find-unassigned-lit-def fref-param1[*symmetric*]
apply (*refine-vcg*
decide-l-or-skip-spec[*THEN fref-to-Down*, *THEN weaken-⇓'*]
skip-and-resolve-loop-l-spec[*THEN fref-to-Down*]
backtrack-l-spec[*THEN fref-to-Down*]; *remove-dummy-vars*)
subgoal for *S S'*
unfolding *cdcl-twl-o-prog-l-pre-def cdcl-twl-o-prog-pre-def* **by** (*rule exI*[*of* - *S'*]) (*force simp*: *twl-st-l*)
subgoal by *auto*
subgoal unfolding *cdcl-twl-o-prog-pre-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-o-prog-pre-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

5.3 Full Strategy

definition *cdcl-twl-stgy-prog-l-inv* :: ‹'v *twl-st-l* ⇒ *bool* × 'v *twl-st-l* ⇒ *bool*› **where**
‹*cdcl-twl-stgy-prog-l-inv* S_0 ≡ λ(*brk*, T). ∃ S_0' T'. (T, T') ∈ *twl-st-l None* ∧
(S_0, S_0') ∈ *twl-st-l None* ∧
twl-struct-invs T' ∧
twl-stgy-invs T' ∧
(*brk* ⟶ *final-twl-state T'*) ∧
*cdcl-twl-stgy*** S_0' T' ∧
clauses-to-update-l T = {#} ∧
(¬*brk* ⟶ *get-conflict-l T* = *None*)›

definition *cdcl-twl-stgy-prog-l* :: ‹'v *twl-st-l* ⇒ 'v *twl-st-l nres*› **where**
‹*cdcl-twl-stgy-prog-l* S_0 =
do {

```
    do {
      (brk, T) ← WHILE_T^cdcl-twl-stgy-prog-l-inv S_0
        (λ(brk, -). ¬brk)
        (λ(brk, S).
        do {
          T ← unit-propagation-outer-loop-l S;
          cdcl-twl-o-prog-l T
        })
        (False, S_0);
      RETURN T
    }
  }
  ›

lemma cdcl-twl-stgy-prog-l-spec:
  ‹(cdcl-twl-stgy-prog-l, cdcl-twl-stgy-prog) ∈
    {(S, S'). (S, S') ∈ twl-st-l None  ∧ twl-list-invs S ∧ clauses-to-update-l S = {#}} →_f
    ⟨{(T, T'). (T, T') ∈ {(T, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T} ∧ True}⟩ nres-rel›
  (is ‹ - ∈ ?R →_f ?I›  is ‹ - ∈ ?R →_f ⟨?J⟩nres-rel›)
proof −
  have R: ‹(a, b) ∈ ?R ⟹
    ((False, a), (False, b)) ∈ {((brk, S), (brk', S')). brk = brk' ∧ (S, S') ∈ ?R}›
    for a b by auto

  show ?thesis
    unfolding cdcl-twl-stgy-prog-l-def cdcl-twl-stgy-prog-def cdcl-twl-o-prog-l-spec
      fref-param1[symmetric] cdcl-twl-stgy-prog-l-inv-def
    apply (refine-rcg R cdcl-twl-o-prog-l-spec[THEN fref-to-Down, THEN weaken-⇓']
        unit-propagation-outer-loop-l-spec[THEN fref-to-Down]; remove-dummy-vars)
    subgoal for S_0 S_0' T T'
      apply (rule exI[of - S_0'])
      apply (rule exI[of - ‹snd T›])
      by (auto simp add: case-prod-beta)
    subgoal by auto
    subgoal by fastforce
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
qed

lemma refine-pair-to-SPEC:
  fixes f :: ‹'s ⇒ 's nres› and g :: ‹'b ⇒ 'b nres›
  assumes ‹(f, g) ∈ {(S, S'). (S, S') ∈ H ∧ R S S'} →_f ⟨{(S, S'). (S, S') ∈ H' ∧ P' S}⟩nres-rel›
    (is ‹- ∈ ?R →_f ?I›)
  assumes ‹R S S'› and [simp]: ‹(S, S') ∈ H›
  shows ‹f S ≤ ⇓ {(S, S'). (S, S') ∈ H' ∧ P' S} (g S')›
proof −
  have ‹(f S, g S') ∈ ?I›
    using assms unfolding fref-def nres-rel-def by auto
  then show ?thesis
    unfolding nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail
    by auto
qed

definition cdcl-twl-stgy-prog-l-pre where
```

‹*cdcl-twl-stgy-prog-l-pre S S'* ⟷
((*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'* ∧ *twl-stgy-invs S'* ∧
clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None* ∧ *twl-list-invs S* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S'*)›

lemma *cdcl-twl-stgy-prog-l-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-l-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-l S* ≤ ⇓ (*twl-st-l None*) (*conclusive-TWL-norestart-run S'*)›
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-l-spec*[*THEN refine-pair-to-SPEC*,
of S S']])
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal
apply (*rule ref-two-step*)
prefer *2*
apply (*rule cdcl-twl-stgy-prog-spec*)
using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** (*auto intro*: *conc-fun-R-mono*)
done

lemma *cdcl-twl-stgy-prog-l-spec-final'*:
assumes
‹*cdcl-twl-stgy-prog-l-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-l S* ≤ ⇓ {(*S*, *T*). (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
twl-struct-invs S' ∧ *twl-stgy-invs S'*} (*conclusive-TWL-norestart-run S'*)›
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-l-spec*[*THEN refine-pair-to-SPEC*,
of S S']])
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal
apply (*rule ref-two-step*)
prefer *2*
apply (*rule cdcl-twl-stgy-prog-spec*)
using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** (*auto intro*: *conc-fun-R-mono*)
done

definition *cdcl-twl-stgy-prog-break-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*cdcl-twl-stgy-prog-break-l S*$_0$ =
do {
b ← *SPEC*(λ-. *True*);
(*b*, *brk*, *T*) ← *WHILE*$_T$λ(*b*, *S*). *cdcl-twl-stgy-prog-l-inv S*$_0$ *S*
(λ(*b*, *brk*, -). *b* ∧ ¬*brk*)
(λ(-, *brk*, *S*). *do* {
T ← *unit-propagation-outer-loop-l S*;
T ← *cdcl-twl-o-prog-l T*;
b ← *SPEC*(λ-. *True*);
RETURN (*b*, *T*)
})
(*b*, *False*, *S*$_0$);
if brk then RETURN T
else cdcl-twl-stgy-prog-l T
}›

lemma *cdcl-twl-stgy-prog-break-l-spec*:

```
  ‹(cdcl-twl-stgy-prog-break-l, cdcl-twl-stgy-prog-break) ∈
    {(S, S'). (S, S') ∈ twl-st-l None  ∧ twl-list-invs S ∧ clauses-to-update-l S = {#}} →f
    ⟨{(T, T'). (T, T') ∈ {(T, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T} ∧ True}⟩ nres-rel›
  (is ‹ - ∈ ?R →f ?I› is ‹ - ∈ ?R →f ⟨?J⟩nres-rel›)
proof −
  have R: ‹(a, b) ∈ ?R ⟹ (bb, bb') ∈ bool-rel ⟹
    ((bb, False, a), (bb', False, b)) ∈ {((b, brk, S), (b', brk', S')). b = b' ∧ brk = brk' ∧
       (S, S') ∈ ?R}›
    for a b bb bb' by auto

  show ?thesis
  supply [[goals-limit=1]]
    unfolding cdcl-twl-stgy-prog-break-l-def cdcl-twl-stgy-prog-break-def cdcl-twl-o-prog-l-spec
      fref-param1[symmetric] cdcl-twl-stgy-prog-l-inv-def
    apply (refine-rcg cdcl-twl-o-prog-l-spec[THEN fref-to-Down]
        unit-propagation-outer-loop-l-spec[THEN fref-to-Down]
        cdcl-twl-stgy-prog-l-spec[THEN fref-to-Down]; remove-dummy-vars)
    apply (rule R)
    subgoal by auto
    subgoal by auto
    subgoal for S0 S0' b b' T T'
      apply (rule exI[of - S0'])
      apply (rule exI[of - ‹snd (snd T)›])
      by (auto simp add: case-prod-beta)
    subgoal
     by auto
    subgoal by fastforce
    subgoal by (auto simp: twl-st-l)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
qed

lemma cdcl-twl-stgy-prog-break-l-spec-final:
  assumes
    ‹cdcl-twl-stgy-prog-l-pre S S'›
  shows
    ‹cdcl-twl-stgy-prog-break-l S ≤ ⇓ (twl-st-l None) (conclusive-TWL-norestart-run S')›
  apply (rule order-trans[OF cdcl-twl-stgy-prog-break-l-spec[THEN refine-pair-to-SPEC,
        of S S']])
  subgoal using assms unfolding cdcl-twl-stgy-prog-l-pre-def by auto
  subgoal using assms unfolding cdcl-twl-stgy-prog-l-pre-def by auto
  subgoal
    apply (rule ref-two-step)
     prefer 2
     apply (rule cdcl-twl-stgy-prog-break-spec)
    using assms unfolding cdcl-twl-stgy-prog-l-pre-def
    by (auto intro: conc-fun-R-mono)
  done


definition cdcl-twl-stgy-prog-early-l :: ‹'v twl-st-l ⇒ (bool × 'v twl-st-l) nres› where
  ‹cdcl-twl-stgy-prog-early-l S0 =
  do {
```

```
    b ← SPEC(λ-. True);
    (b, brk, T) ← WHILE_T λ(b, S). cdcl-twl-stgy-prog-l-inv S_0 S
      (λ(b, brk, -). b ∧ ¬brk)
      (λ(-, brk, S). do {
        T ← unit-propagation-outer-loop-l S;
        T ← cdcl-twl-o-prog-l T;
        b ← SPEC(λ-. True);
        RETURN (b, T)
      })
      (b, False, S_0);
    RETURN (brk, T)
  }›

lemma cdcl-twl-stgy-prog-early-l-spec:
  ‹(cdcl-twl-stgy-prog-early-l, cdcl-twl-stgy-prog-early) ∈
    {(S, S'). (S, S') ∈ twl-st-l None ∧ twl-list-invs S ∧ clauses-to-update-l S = {#}} →_f
    ⟨bool-rel ×_r {(T, T'). (T, T') ∈ {(T, T'). (T, T') ∈ twl-st-l None ∧ twl-list-invs T} ∧ True}⟩
nres-rel›
  (is ‹ - ∈ ?R →_f ?I› is ‹ - ∈ ?R →_f ⟨?J⟩nres-rel›)
proof −
  have R: ‹(a, b) ∈ ?R ⟹ (bb, bb') ∈ bool-rel ⟹
    ((bb, False, a), (bb', False, b)) ∈ {((b, brk, S), (b', brk', S')). b = b' ∧ brk = brk' ∧
      (S, S') ∈ ?R}›
    for a b bb bb' by auto

  show ?thesis
  supply [[goals-limit=1]]
    unfolding cdcl-twl-stgy-prog-early-l-def cdcl-twl-stgy-prog-early-def cdcl-twl-o-prog-l-spec
      fref-param1[symmetric] cdcl-twl-stgy-prog-l-inv-def
    apply (refine-rcg cdcl-twl-o-prog-l-spec[THEN fref-to-Down]
        unit-propagation-outer-loop-l-spec[THEN fref-to-Down]
        cdcl-twl-stgy-prog-l-spec[THEN fref-to-Down]; remove-dummy-vars)
    apply (rule R)
    subgoal by auto
    subgoal by auto
    subgoal for S_0 S_0' b b' T T'
      apply (rule exI[of - S_0'])
      apply (rule exI[of - ‹snd (snd T)›])
      by (auto simp add: case-prod-beta)
    subgoal
     by auto
    subgoal by fastforce
    subgoal by (auto simp: twl-st-l)
    subgoal by auto
    subgoal by auto
    done
qed

lemma refine-pair-to-SPEC2:
  fixes f :: ‹'s ⇒ - nres› and g :: ‹'b ⇒ (-) nres›
  assumes ‹(f, g) ∈ {(S, S'). (S, S') ∈ H ∧ R S S'} →_f ⟨Id ×_r {(S, S'). (S, S') ∈ H' ∧ P' S}⟩nres-rel›
    (is ‹ - ∈ ?R →_f ?I›)
  assumes ‹R S S'› and [simp]: ‹(S, S') ∈ H›
  shows ‹f S ≤ ⇓ (Id ×_r {(S, S'). (S, S') ∈ H' ∧ P' S}) (g S')›
proof −
  have ‹(f S, g S') ∈ ?I›
```

using *assms* **unfolding** *fref-def nres-rel-def* **by** *auto*
then show *?thesis*
unfolding *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
by *auto*
qed

lemma *cdcl-twl-stgy-prog-early-l-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-l-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-early-l S* ≤ ⇓ (*bool-rel* ×$_r$ *twl-st-l None*) (*partial-conclusive-TWL-norestart-run S'*)›
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-early-l-spec*[*THEN refine-pair-to-SPEC2*, *of S S'*]])
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal
apply (*rule ref-two-step*)
prefer *2*
apply (*rule cdcl-twl-stgy-prog-early-spec*)
using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def*
by (*auto intro*: *conc-fun-R-mono*)
done

end
theory *Watched-Literals-Transition-System-Simp*
imports
Watched-Literals-Transition-System-Reduce
Watched-Literals-Transition-System-Restart
begin

context *twl-restart*
begin

theorem *wf-cdcl-twl-stgy-restart*:
‹*wf* {(*T*, *S* :: 'v *twl-st-restart*). *twl-restart-inv S* ∧ *cdcl-twl-stgy-restart S T*}› (**is** ‹*wf ?S*›)
proof −
have ‹*?S* ⊆
{((*S'*, *T'*, *U'*, *m'*, *n'*, *g'*), (*S*, *T*, *U*, *m*, *n*, *g*)).
pcdcl-stgy-restart-inv (*pstate*$_W$ *-of S*, *pstate*$_W$ *-of T*, *pstate*$_W$ *-of U*, *m*, *n*, *g*) ∧
pcdcl-stgy-restart (*pstate*$_W$ *-of S*, *pstate*$_W$ *-of T*, *pstate*$_W$ *-of U*, *m*, *n*, *g*)
(*pstate*$_W$ *-of S'*, *pstate*$_W$ *-of T'*, *pstate*$_W$ *-of U'*, *m'*, *n'*, *g'*)} ∪
{((*S'*, *T'*, *U'*, *m'*, *n'*, *g'*), (*S*, *T*, *U*, *m*, *n*, *g*)).
pcdcl-stgy-restart-inv (*pstate*$_W$ *-of S*, *pstate*$_W$ *-of T*, *pstate*$_W$ *-of U*, *m*, *n*, *g*) ∧
S = *S'* ∧ *T* = *T'* ∧ *m* = *m'* ∧ *n* = *n'* ∧ *g* = *g'* ∧
pstate$_W$ *-of U* = *pstate*$_W$ *-of U'* ∧ (*literals-to-update-measure U'*, *literals-to-update-measure U*)
∈ *lexn less-than 2*}› (**is** ‹- ⊆ *?A* ∪ *?B*›)
by (*auto dest*!: *cdcl-twl-stgy-cdcl*$_W$*-stgy-restart2 simp*: *twl-restart-inv-def*)
moreover have ‹*wf* (*?A* ∪ *?B*)›
apply (*rule wf-union-compatible*)
subgoal
by (*rule wf-subset*[*OF wf-if-measure-f*[*OF wf-pcdcl-twl-stgy-restart*],
of - ‹λ(*S*, *T*, *U*, *m*, *g*). (*pstate*$_W$ *-of S*, *pstate*$_W$ *-of T*, *pstate*$_W$ *-of U*, *m*, *g*)›]) *auto*
subgoal
by (*rule wf-subset*[*OF wf-if-measure-f*[*OF*],

```
      of ‹(lexn less-than 2)›  - ‹λ(S, T, U, m, g). literals-to-update-measure (U)›])
      (auto intro!: wf-lexn)
    subgoal
      by auto
    done
  ultimately show ?thesis
    by (blast intro: wf-subset)
qed

end


context twl-restart-ops
begin

lemma cdcl-twl-stgy-size-get-all-learned:
  ‹cdcl-twl-stgy S T ⟹ size (get-all-learned-clss S) ≤ size (get-all-learned-clss T)›
  by (induction rule: cdcl-twl-stgy.induct)
   (auto simp: cdcl-twl-cp.simps cdcl-twl-o.simps update-clauses.simps
    dest: multi-member-split)

lemma rtranclp-cdcl-twl-stgy-size-get-all-learned:
  ‹cdcl-twl-stgy** S T ⟹ size (get-all-learned-clss S) ≤ size (get-all-learned-clss T)›
  by (induction rule: rtranclp-induct)
    (auto dest!: cdcl-twl-stgy-size-get-all-learned)


lemma (in −) wf-trancl-iff: ‹wf (r⁺) ⟷ wf r›
  by (auto intro!: wf-subset[of ‹r⁺› r] simp: wf-trancl)

lemma (in −) tranclp-inv-tranclp:
  assumes ‹⋀S T. R S T ⟹ P S ⟹ P T›
  shows
    ‹{(T, S). R S T ∧ P S}⁺ = {(T, S). R⁺⁺ S T ∧ P S}›
proof −
  have H: ‹R⁺⁺ S T ⟹ P S ⟹ P T› for S T
    by (induction rule: tranclp-induct)
      (auto dest: assms)
  show ?thesis
    apply (rule; rule; clarify)
    unfolding mem-Collect-eq in-pair-collect-simp
    subgoal for a b
      by (induction rule: trancl-induct) auto
    subgoal for b x
      apply (induction rule: tranclp-induct)
      subgoal for b
        by auto
      subgoal for e f
        by (rule trancl-into-trancl2[of f e])
          (use H[of x e] in auto)
      done
    done
qed

definition partial-conclusive-TWL-run :: ‹'v twl-st ⇒ (bool × 'v twl-st) nres› where
  ‹partial-conclusive-TWL-run S = SPEC(λ(b, T). ∃ R1 R2 m m₀ n₀.
```

*cdcl-twl-stgy-restart*** (*S*, *S*, *S*, m_0, n_0, *True*) (*R1*, *R2*, *T*, *m*) ∧ (¬*b* ⟶ *final-twl-state T*))›

definition *partial-conclusive-TWL-run2*
:: ‹*nat* ⇒ *nat* ⇒ *'v twl-st* ⇒ *'v twl-st* ⇒ *'v twl-st* ⇒ (*bool* × *'v twl-st*) *nres*›
where
‹*partial-conclusive-TWL-run2* m_0 n_0 S_0 T_0 U_0 = *SPEC*(λ(*b*, *T*). ∃ *R1 R2 m*.
*cdcl-twl-stgy-restart*** (S_0, T_0, U_0, m_0, n_0, *True*) (*R1*, *R2*, *T*, *m*) ∧ (¬*b* ⟶ *final-twl-state T*))›

definition *conclusive-TWL-run* :: ‹*'v twl-st* ⇒ *'v twl-st nres*› **where**
‹*conclusive-TWL-run S* = *SPEC*(λ*T*. (∃ *R1 R2 m* m_0 n_0.
*cdcl-twl-stgy-restart*** (*S*, *S*, *S*, m_0, n_0, *True*) (*R1*, *R2*, *T*, *m*) ∧ *final-twl-state T*))›

definition *conclusive-TWL-run2* :: ‹*nat* ⇒ *nat* ⇒ *'v twl-st* ⇒ *'v twl-st* ⇒ *'v twl-st* ⇒ *'v twl-st nres*›
where
‹*conclusive-TWL-run2* m_0 n_0 S_0 T_0 U_0 = *SPEC*(λ*T*. (∃ *R1 R2 m*.
*cdcl-twl-stgy-restart*** (S_0, T_0, U_0, m_0, n_0, *True*) (*R1*, *R2*, *T*, *m*) ∧ *final-twl-state T*))›
end

end
theory *Watched-Literals-Algorithm-Reduce*
imports *Watched-Literals-Algorithm Watched-Literals-Transition-System-Simp*
Watched-Literals-Transition-System-Reduce
Weidenbach-Book-Base.Explorer
begin

context *twl-restart-ops*
begin

We refine in two steps. In the first, we refine the transition system directly. Then we simplify the stat and remove the unnecessary state and replace them by the counts we need.

Restarts are never necessary

definition *GC-required* :: *'v twl-st* ⇒ *nat* ⇒ *nat* ⇒ *bool nres* **where**
‹*GC-required S last-GC-learnt-clss n* = *do* {
ASSERT(*size* (*get-all-learned-clss S*) ≥ *last-GC-learnt-clss*);
SPEC (λ*b*. *b* ⟶ *size* (*get-all-learned-clss S*) − *last-GC-learnt-clss* > *f n*)}›

definition *restart-required* :: *'v twl-st* ⇒ *nat* ⇒ *nat* ⇒ *bool nres* **where**
‹*restart-required S last-Restart-learnt-clss n* = *do* {
ASSERT(*size* (*get-all-learned-clss S*) ≥ *last-Restart-learnt-clss*);
SPEC (λ*b*. *b* ⟶ *size* (*get-all-learned-clss S*) > *last-Restart-learnt-clss*)
}›

definition *inprocessing-required* :: *'v twl-st* ⇒ *bool nres* **where**
‹*inprocessing-required S* = *do* {
SPEC (λ*b*. *True*)
}›

definition (**in** −) *restart-prog-pre-int* :: ‹*'v twl-st* ⇒ *'v twl-st* ⇒ *'v twl-st* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-prog-pre-int last-GC last-Restart S brk* ⟷ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧
(¬*brk* ⟶ *get-conflict S* = *None*) ∧
size (*get-all-learned-clss S*) ≥ *size* (*get-all-learned-clss last-Restart*) ∧
size (*get-all-learned-clss S*) ≥ *size* (*get-all-learned-clss last-GC*) ∧
$cdcl_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)›

definition *restart-prog-int*

:: ‹'v twl-st ⇒ 'v twl-st ⇒ 'v twl-st ⇒ nat ⇒ nat ⇒ bool ⇒ ('v twl-st × 'v twl-st × 'v twl-st × nat × nat) nres›
where
‹restart-prog-int last-GC last-Restart S m n brk = do {
ASSERT(restart-prog-pre-int last-GC last-Restart S brk);
b ← GC-required S (size (get-all-learned-clss last-GC)) n;
b2 ← restart-required S (size (get-all-learned-clss last-Restart)) n;
if b2 ∧ ¬brk then do {
T ← SPEC(λT. cdcl-twl-restart-only S T);
RETURN (last-GC, T, T, Suc m, n)
}
else if b ∧ ¬brk then do {
b ← inprocessing-required S;
if ¬b then do {
T ← SPEC(λT. cdcl-twl-restart S T);
RETURN (T, T, T, m, Suc n)
}
else do {
T ← SPEC(λT. cdcl-twl-inp** S T ∧ count-decided (get-trail T) = 0);
RETURN (T, T, T, m, Suc n)
}
}
else
RETURN (last-GC, last-Restart, S, m, n)
}›

fun *cdcl-twl-stgy-restart-prog-int-inv* **where**
‹cdcl-twl-stgy-restart-prog-int-inv (S₀, T₀, U₀, m₀, n₀) (brk, R, S, T, m, n) ⟷
twl-restart-inv (S₀, T₀, U₀, m₀, n₀, True) ∧
twl-stgy-restart-inv (S₀, T₀, U₀, m₀, n₀, True) ∧
(brk ⟶ final-twl-state T) ∧
cdcl-twl-stgy-restart** (S₀, T₀, U₀, m₀, n₀, True) (R, S, T, m, n, ¬brk) ∧
clauses-to-update T = {#} ∧ (¬brk ⟶ get-conflict T = None)›

lemmas *cdcl-twl-stgy-restart-prog-int-inv-def*[*simp del*] =
cdcl-twl-stgy-restart-prog-int-inv.simps

lemma *cdcl-twl-stgy-restart-prog-int-inv-alt-def*:
‹cdcl-twl-stgy-restart-prog-int-inv (S₀, T₀, U₀, m₀, n₀) (brk, R, S, T, m, n) ⟷
twl-restart-inv (S₀, T₀, U₀, m₀, n₀, True) ∧
twl-stgy-restart-inv (S₀, T₀, U₀, m₀, n₀, True) ∧
twl-stgy-restart-inv (R, S, T, m, n, ¬brk) ∧
twl-restart-inv (R, S, T, m, n, ¬brk) ∧
(brk ⟶ final-twl-state T) ∧
cdcl-twl-stgy-restart** (S₀, T₀, U₀, m₀, n₀, True) (R, S, T, m, n, ¬brk) ∧
clauses-to-update T = {#} ∧ (¬brk ⟶ get-conflict T = None)›
unfolding *cdcl-twl-stgy-restart-prog-int-inv-def*
by (*auto dest*: *rtranclp-cdcl-twl-stgy-restart-twl-restart-inv*
rtranclp-cdcl-twl-stgy-restart-twl-stgy-restart-inv)

In the main loop, and purely to simplify the proof, we remember the state obtained after the last restart in order to relate it to the number of clauses. In a first proof attempt, we try to make do without it by only assuming its existence, but we could no prove that the loop terminates, because the state can change each time.

This state is not needed at all in the execution and will be removed in the next refinement step.

definition *cdcl-twl-stgy-restart-prog-intg*
:: *nat* ⇒ *nat* ⇒ *'v twl-st* ⇒*'v twl-st* ⇒*'v twl-st* ⇒ *'v twl-st nres*
where
‹*cdcl-twl-stgy-restart-prog-intg* m_0 n_0 S_0 T_0 U_0 =
do {
(brk, -, -, T, -, -) ← $WHILE_T$*cdcl-twl-stgy-restart-prog-int-inv* (S_0, T_0, U_0, m_0, n_0)
(λ(brk, -). ¬brk)
(λ(brk, S, S', S'', m, n).
do {
T ← *unit-propagation-outer-loop* S'';
(brk, T) ← *cdcl-twl-o-prog* T;
(S, S', T, m', n') ← *restart-prog-int* S S' T m n brk;
RETURN (brk ∨ *get-conflict* T ≠ *None*, S, S', T, m', n')
})
(*False*, S_0, T_0, U_0, m_0, n_0);
RETURN T
}›

abbreviation *cdcl-twl-stgy-restart-prog-int* **where**
‹*cdcl-twl-stgy-restart-prog-int* S ≡ *cdcl-twl-stgy-restart-prog-intg* 0 0 S S S›

lemmas *cdcl-twl-stgy-restart-prog-int-def* = *cdcl-twl-stgy-restart-prog-intg-def*

abbreviation *cdcl-algo-termination-early-rel*
:: ‹((*bool* × *bool* × *'v twl-st* × *'v twl-st* × *'v twl-st* × *nat* × *nat*) × -) *set*›
where
‹*cdcl-algo-termination-early-rel* ≡
{((*ebrkT* :: *bool*, *brkT* :: *bool*, R' :: *'v twl-st*, S' :: *'v twl-st*, T' :: *'v twl-st*, m' :: *nat*, n' :: *nat*),
(*ebrkS*, *brkS*, R :: *'v twl-st*, S :: *'v twl-st*, T :: *'v twl-st*, m :: *nat*, n :: *nat*)).
twl-restart-inv (R, S, T, m, n, ¬*brkS*) ∧ ¬*brkS* ∧
cdcl-twl-stgy-restart^{++} (R, S, T, m, n, ¬*brkS*) (R', S', T', m', n', ¬*brkT*)} ∪
{((*ebrkT* :: *bool*, *brkT* :: *bool*, R' :: *'v twl-st*, S' :: *'v twl-st*, T' :: *'v twl-st*, m' :: *nat*, n' :: *nat*),
(*ebrkS*, *brkS*, R :: *'v twl-st*, S :: *'v twl-st*, T :: *'v twl-st*, m :: *nat*, n :: *nat*)).
¬*brkS* ∧ *brkT*}›

end

context *twl-restart*
begin

lemma *cdcl-twl-stgy-restart-tranclpI*:
‹*cdcl-twl-stgy*$^{++}$ T U ⟹ *cdcl-twl-stgy-restart*$^{++}$ (R, S, T, m, n, *True*) (R, S, U, m, n, *True*)›
by (*induction rule*: *tranclp-induct*)
(*auto dest*!: *cdcl-twl-stgy-restart.intros*(*1*)[*of* - - R S m n])

lemma *cdcl-twl-stgy-restart-rtranclpI*:
‹*cdcl-twl-stgy*** T U ⟹ *cdcl-twl-stgy-restart*** (R, S, T, m, n, *True*) (R, S, U, m, n, *True*)›
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *cdcl-twl-stgy-restart.intros*(*1*)[*of* - - R S m n])

lemma *wf-cdcl-algo-termination-early-rel*: ‹*wf cdcl-algo-termination-early-rel*› (**is** ‹*wf* (*?C* ∪ *?D*)›)
proof −
have A: ‹{(T, S :: *'v twl-st-restart*). *twl-restart-inv* S ∧ *cdcl-twl-stgy-restart* S T}$^+$ =
{(T, S :: *'v twl-st-restart*). *twl-restart-inv* S ∧ *cdcl-twl-stgy-restart*$^{++}$ S T}› (**is** ‹*?A* = *?B*›)
proof −

```
    have ‹(x, y) ∈ ?A ⟹ (x, y) ∈ ?B› for x y
      by (induction rule: trancl-induct)
        auto
    moreover have ‹cdcl-twl-stgy-restart^++ S T ⟹ twl-restart-inv S ⟹ (T, S) ∈ ?A› for S T
      apply (induction rule: tranclp-induct)
      subgoal by auto
      subgoal for T U
        by (rule trancl-into-trancl2[of - T])
         (use rtranclp-cdcl-twl-stgy-restart-twl-restart-inv[of S T] in ‹simp-all add: tranclp-into-rtranclp›)
      done
    ultimately show ?thesis
      by blast
  qed
  have ‹wf ?D›
    by (rule wf-no-loop) auto
  moreover have ‹wf ?C›
    by (rule wf-if-measure-in-wf[OF wf-trancl[OF wf-cdcl-twl-stgy-restart, unfolded A],
        of - ‹λ(-, brkT, R, S, T, n, n'). (R, S, T, n, n', ¬brkT)›])
      auto
  moreover have ‹?D O ?C ⊆ ?D›
    by auto
  ultimately show ?thesis
    apply (subst Un-commute)
    by (rule wf-union-compatible)
qed

definition (in twl-restart-ops) cdcl-twl-stgy-restart-prog-bounded-intg
    :: nat ⇒ nat ⇒ 'v twl-st ⇒ 'v twl-st ⇒ 'v twl-st ⇒ (bool × 'v twl-st) nres
where
  ‹cdcl-twl-stgy-restart-prog-bounded-intg m n S_0 T_0 U_0 =
  do {
    ebrk ← RES UNIV;
    (ebrk, -, -, -, T, -, -) ← WHILE_T^cdcl-twl-stgy-restart-prog-int-inv (S_0, T_0, U_0, m, n) o snd
      (λ(ebrk, brk, -). ¬brk ∧ ¬ebrk)
      (λ(ebrk, brk, Q, R, S, m, n).
      do {
        T ← unit-propagation-outer-loop S;
        (brk, T) ← cdcl-twl-o-prog T;
        (Q, R, T, m, n) ← restart-prog-int Q R T m n brk;
 ebrk ← RES UNIV;
        RETURN (ebrk, brk ∨ get-conflict T ≠ None, Q, R, T, m, n)
      })
      (ebrk, False, S_0, T_0, U_0, m, n);
    RETURN (ebrk, T)
  }›

abbreviation (in twl-restart-ops) cdcl-twl-stgy-restart-prog-bounded-int where
  ‹cdcl-twl-stgy-restart-prog-bounded-int S ≡ cdcl-twl-stgy-restart-prog-bounded-intg 0 0 S S S›

lemmas cdcl-twl-stgy-restart-prog-bounded-int-def = cdcl-twl-stgy-restart-prog-bounded-intg-def

lemma pcdcl-core-stgy-get-all-learned-clss:
  ‹pcdcl-core-stgy T U ⟹
  size (pget-all-learned-clss T) ≤ size (pget-all-learned-clss U)›
  by (induction rule: pcdcl-core-stgy.induct)
   (cases T; cases U; auto simp: cdcl-conflict.simps cdcl-propagate.simps cdcl-decide.simps
```

cdcl-skip.simps cdcl-resolve.simps cdcl-backtrack.simps
dest!: *arg-cong*[*of* ‹*clauses* -› ‹-› *size*])+

lemma *cdcl-twl-cp-get-all-learned-clss*:
‹*cdcl-twl-cp T U* ⟹
size (*get-all-learned-clss T*) = *size* (*get-all-learned-clss U*)›
by (*induction rule*: *cdcl-twl-cp.induct*)
(*auto simp*: *update-clauses.simps dest*!: *multi-member-split*)

lemma *rtranclp-cdcl-twl-cp-get-all-learned-clss*:
‹*cdcl-twl-cp*** *T U* ⟹
size (*get-all-learned-clss T*) = *size* (*get-all-learned-clss U*)›
by (*induction rule*:*rtranclp-induct*)
(*auto dest*: *cdcl-twl-cp-get-all-learned-clss*)

lemma *cdcl-twl-o-get-all-learned-clss*:
‹*cdcl-twl-o T U* ⟹
size (*get-all-learned-clss T*) ≤ *size* (*get-all-learned-clss U*)›
by (*induction rule*: *cdcl-twl-o.induct*)
(*auto simp*: *update-clauses.simps dest*!: *multi-member-split*)

lemma *rtranclp-cdcl-twl-o-get-all-learned-clss*:
‹*cdcl-twl-o*** *T U* ⟹
size (*get-all-learned-clss T*) ≤ *size* (*get-all-learned-clss U*)›
by (*induction rule*:*rtranclp-induct*)
(*auto dest*: *cdcl-twl-o-get-all-learned-clss*)

lemma *rtranclp-pcdcl-stgy-only-restart-pget-all-learned-clss-mono*:
‹*pcdcl-stgy-only-restart*** *S T* ⟹
size (*pget-all-learned-clss S*) ≤ *size* (*pget-all-learned-clss T*)›
by (*induction rule*:*rtranclp-induct*)
(*auto dest*: *pcdcl-stgy-only-restart-pget-all-learned-clss-mono*)

lemma *cdcl-twl-inp-clauses-to-update*:
‹*cdcl-twl-inp*** *S T* ⟹ *clauses-to-update S* = {#} ⟹ *clauses-to-update T* = {#}›
by (*cases rule*: *rtranclp.cases*, *assumption*)
(*auto simp*: *cdcl-twl-inp.simps cdcl-twl-subsumed.simps cdcl-twl-subresolution.simps*
cdcl-twl-restart.simps cdcl-twl-unitres.simps cdcl-twl-unitres-true.simps
cdcl-twl-pure-remove.simps)

lemma *restart-prog-bounded-spec*:
assumes
‹*iebrk* ∈ *UNIV*› **and**
inv: ‹(*cdcl-twl-stgy-restart-prog-int-inv* (*S*$_0$, *T*$_0$, *U*$_0$, *m*$_0$, *n*$_0$) ∘ *snd*) (*ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*)› **and**
cond: ‹*case* (*ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*) *of*
(*ebrk*, *brk*, *uu*-) ⇒ ¬ *brk* ∧ ¬ *ebrk*› **and**
other-inv: ‹*cdcl-twl-o-prog-spec V* (*brkW*, *W*)› **and**
‹*twl-struct-invs V*› **and**
‹*cdcl-twl-cp*** *U V*› **and**
‹*literals-to-update V* = {#}› **and**
‹∀ *S*′. ¬ *cdcl-twl-cp V S*′› **and**
‹*twl-stgy-invs V*›
shows ‹*restart-prog-int S T W m n brkW*
≤ *SPEC*
(λ*x*. (*case x of* (*Q*, *R*, *T*, *m*, *n*) ⇒ *do* {
ebrk ← *RES UNIV*;

```
                    RETURN (ebrk, brkW ∨ get-conflict T ≠ None, Q, R, T, m, n)
                  })
          ≤ SPEC
            (λs'. (cdcl-twl-stgy-restart-prog-int-inv (S₀, T₀, U₀, m₀, n₀) ∘ snd) s' ∧
                (s', ebrk, brk, S, T, U, m, n)
                ∈ cdcl-algo-termination-early-rel))›
  (is ‹- ≤ SPEC (λx. - ≤ SPEC(λs. ?I s ∧ - ∈ ?term))›)
proof −

  have [simp]: ‹¬brk› ‹¬ebrk›
    using cond by auto
  have struct-invs': ‹cdcl-twl-restart W T' ⟹ cdcl-twl-stgy** T' V' ⟹ twl-struct-invs V'› for T' V'
    using assms(3) cdcl-twl-restart-twl-struct-invs
      by (metis (no-types, lifting) assms(4) case-prodD rtranclp-cdcl-twl-stgy-twl-struct-invs)
  have stgy-invs: ‹cdcl-twl-restart W V' ⟹ cdcl-twl-stgy** V' W' ⟹ twl-stgy-invs W'› for V' W'
    using assms(3) cdcl-twl-restart-twl-stgy-invs
    by (metis (no-types, lifting) assms(4) case-prodD cdcl-twl-restart-twl-struct-invs
      rtranclp-cdcl-twl-stgy-twl-stgy-invs)

  have res-no-clss-to-upd: ‹cdcl-twl-restart U V ⟹ clauses-to-update V = {#}› for V
    by (auto simp: cdcl-twl-restart.simps)
  have res-no-confl: ‹cdcl-twl-restart U V ⟹ get-conflict V = None› for V
    by (auto simp: cdcl-twl-restart.simps)

  have
    struct-invs-S0: ‹twl-struct-invs S₀› ‹twl-struct-invs T₀› ‹twl-struct-invs U₀› and
    struct-invs-S: ‹twl-struct-invs S› and
    struct-invs-T: ‹twl-struct-invs T› and
    struct-invs-U: ‹twl-struct-invs U› and
    twl-stgy-invs-S0: ‹twl-stgy-invs S₀›‹twl-stgy-invs T₀›‹twl-stgy-invs U₀› and
    twl-stgy-invs-S: ‹twl-stgy-invs S› and
    twl-stgy-invs-T: ‹twl-stgy-invs T› and
    twl-stgy-invs-U: ‹twl-stgy-invs U› and
    ‹brk ⟶ final-twl-state U› and
    twl-res: ‹cdcl-twl-stgy-restart** (S₀, T₀, U₀, m₀, n₀, True) (S, T, U, m, n, ¬ brk)› and
    clss-T: ‹clauses-to-update U = {#}› and
    confl: ‹¬ brk ⟶ get-conflict U = None› and
    STU-inv: ‹pcdcl-stgy-restart-inv (pstate_W-of S, pstate_W-of T, pstate_W-of U, m, n, ¬ brk)› and
    [simp]: ‹twl-restart-inv (S₀, T₀, U₀, m₀, n₀, True)›
    ‹twl-stgy-restart-inv (S₀, T₀, U₀, m₀, n₀, True)› and
    init: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S)›
      ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of T)›
      ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of U)›
    using inv unfolding cdcl-twl-stgy-restart-prog-int-inv-alt-def prod.case comp-def snd-conv
      twl-restart-inv-def twl-stgy-restart-inv-def by fast+

  have
    cdcl-o: ‹cdcl-twl-o** V W› and
    conflict-W: ‹get-conflict W ≠ None ⟹ count-decided (get-trail W) = 0› and
    brk'-no-step: ‹brkW ⟹ final-twl-state W› and
    struct-invs-W: ‹twl-struct-invs W› and
    stgy-invs-W: ‹twl-stgy-invs W› and
    clss-to-upd-W: ‹clauses-to-update W = {#}› and
    lits-to-upd-W: ‹¬ brkW ⟶ literals-to-update W ≠ {#}› and
    confl-W: ‹¬ brkW ⟶ get-conflict W = None› and
    ent: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of V)›
```

using *other-inv* **unfolding** *final-twl-state-def* **by** *fast+*
have *ent-W*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of W*)›
using *assms*(5) *rtranclp-cdcl-twl-o-stgyD*[*OF cdcl-o*] *ent*
rtranclp-cdcl-twl-stgy-entailed-by-init **by** *blast*
have ‹*cdcl-twl-stgy*** *V W›
by (*meson cdcl-o assms*(5) *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD*
rtranclp-trans)

have ‹*size* (*get-all-learned-clss T*) ≤ *size* (*get-all-learned-clss W*)›
‹*size* (*get-all-learned-clss S*) ≤ *size* (*get-all-learned-clss W*)›
using *STU-inv assms*(6) *cdcl-o* **unfolding** *pcdcl-stgy-restart-inv-def*
by (*auto dest*!: *rtranclp-pcdcl-tcore-stgy-pget-all-learned-clss-mono*
rtranclp-cdcl-twl-cp-get-all-learned-clss rtranclp-cdcl-twl-o-get-all-learned-clss
rtranclp-pcdcl-stgy-only-restart-pget-all-learned-clss-mono)
then have *restart-W*: ‹*restart-prog-pre-int S T W brkW*›
using *struct-invs-W stgy-invs-W confl-W ent-W* **unfolding** *restart-prog-pre-int-def* **by** *auto*

have *UW*: ‹*cdcl-twl-stgy-restart*** (*S*, *T*, *U*, *m*, *n*, *True*) (*S*, *T*, *W*, *m*, *n*, *True*)›
apply (*rule cdcl-twl-stgy-restart-rtranclpI*)
by (*meson* ‹*cdcl-twl-stgy*** *V W*› *assms*(6) *rtranclp-cdcl-twl-cp-stgyD rtranclp-trans*)
have *restart-only*: ‹*?I* (*ebrk'*, *False* ∨ *get-conflict X* ≠ *None*, *S*, *X*, *X*, *Suc m*, *n*)› (**is** *?A*) **and**
restart-only-term: ‹((*ebrk'*, *False* ∨ *get-conflict X* ≠ *None*, *S*, *X*, *X*, *Suc m*, *n*), *ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*) ∈ *?term*› (**is** *?B*)
if
‹*restart-prog-pre-int S T W False*› **and**
less: ‹*True* ⟶
size (*get-all-learned-clss T*) < *size* (*get-all-learned-clss W*)› **and**
WX: ‹*cdcl-twl-restart-only W X*› **and**
‹*ebrk'* ∈ *UNIV*› **and**
y **and**
x
for *x y X ebrk'*
proof −
have *WX'*: ‹*cdcl-twl-stgy-restart* (*S*, *T*, *W*, *m*, *n*, *True*) (*S*, *X*, *X*, *Suc m*, *n*, *True*)›
by (*rule cdcl-twl-stgy-restart.intros*)
(*use less WX* **in** *auto*)
show *?A*
using *UW WX twl-res WX' clss-to-upd-W*
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def cdcl-twl-restart-only.simps*)
show *?B*
using *STU-inv UW WX' init*
by (*auto simp*: *twl-restart-inv-def struct-invs-S struct-invs-T struct-invs-U*)
qed
have *GC*: ‹*?I* (*ebrk'*, *False* ∨ *get-conflict Y* ≠ *None*, *Y*, *Y*, *Y*, *m*, *Suc n*)› (**is** *?A*) **and**
GC-term: ‹((*ebrk'*, *False* ∨ *get-conflict Y* ≠ *None*, *Y*, *Y*, *Y*, *m*, *Suc n*), *ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*) ∈ *?term*› (**is** *?B*)
if
‹*restart-prog-pre-int S T W False*› **and**
less: ‹*True* ⟶ *f n* < *size* (*get-all-learned-clss W*) − *size* (*get-all-learned-clss S*)› **and**
WX: ‹*cdcl-twl-inp*** *W Y*›
‹*count-decided* (*get-trail Y*) = *0*› **and**
‹*ebrk'* ∈ *UNIV*› **and**
‹¬ *brkW*›
for *x X Y ebrk' W'*
proof −
have *struct-X*: ‹*twl-struct-invs Y*›

using *rtranclp-cdcl-twl-inp-invs*(*1*)[*OF WX*(*1*) *struct-invs-W ent-W*] **.**

have *clss*: ‹*clauses-to-update Y* = {#}›
using *cdcl-twl-inp-clauses-to-update*[*OF WX*(*1*) *clss-to-upd-W*] **.**
have *WX'*: ‹*cdcl-twl-stgy-restart* (*S*, *T*, *W*, *m*, *n*, *True*) (*Y*, *Y*, *Y*, *m*, *Suc n*, *True*)›
by (*rule cdcl-twl-stgy-restart.intros*(*2*))
(*use less WX clss* **in** ‹*auto dest*!: *cdcl-twl-inp.intros*›)
have ‹*count-decided* (*get-trail Y*) = *0* ⟹ *get-conflict Y* ≠ *None* ⟹ *final-twl-state Y*›
by (*auto simp*: *final-twl-state-def*)
moreover have ‹*count-decided* (*get-trail Y*) = *0* ⟹ *get-conflict Y* ≠ *None* ⟹
cdcl-twl-stgy-restart (*Y*, *Y*, *Y*, *m*, *Suc n*, *True*) (*Y*, *Y*, *Y*, *m*, *Suc n*, *False*)›
by (*rule cdcl-twl-stgy-restart.intros*)
(*auto simp*: *pcdcl-twl-final-state-def*)
ultimately show *?A*
using *UW WX twl-res WX' clss*
by (*cases* ‹*get-conflict Y* = *None*›) (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def*)
show *?B*
using *STU-inv UW WX' init*
by (*auto simp*: *twl-restart-inv-def struct-invs-S struct-invs-T struct-invs-U*)
qed
have *continue*: ‹*?I* (*ebrk'*, *brkW'* ∨ ¬*unsat*, *S*, *T*, *W*, *m*, *n*)› (**is** *?A*) **and**
continue-term: ‹((*ebrk'*, *brkW'* ∨ ¬*unsat*, *S*, *T*, *W*, *m*, *n*), *ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*) ∈ *?term*› (**is** *?B*)
if ‹*unsat* ⟷ *get-conflict W* = *None*› ‹*brkW'* ⟷ *brkW*›
for *ebrk' unsat brkW'*
proof −
show *?A*
using *brk'-no-step confl-W clss-to-upd-W UW twl-res*
cdcl-twl-stgy-restart.intros(*4*)[*of W S T m n*]
unfolding *that*
apply (*cases* ‹*get-conflict W* = *None*›; *cases brkW*)
apply (*auto simp add*: *cdcl-twl-stgy-restart-prog-int-inv-def*)
apply (*metis* (*no-types*, *lifting*) *final-twl-state-def pcdcl-twl-final-state-def rtranclp.simps rtranclp-trans*)+
done
have ‹*cdcl-twl-stgy*$^{++}$ *V W*› **if** ‹¬*brkW*›
using ‹*cdcl-twl-stgy*** *V W*› *lits-to-upd-W that assms*(*7*) **unfolding** *rtranclp-unfold*
by *auto*
then have [*simp*]: ‹*cdcl-twl-stgy-restart*$^{++}$ (*S*, *T*, *U*, *m*, *n*, *True*) (*S*, *T*, *W*, *m*, *n*, *True*)› **if** ‹¬*brkW*›
by (*meson assms*(*6*) *cdcl-twl-stgy-restart-tranclpI*
rtranclp-cdcl-twl-cp-stgyD rtranclp-tranclp-tranclp that)

show *?B*
using *STU-inv brk'-no-step init*
apply (*cases* ‹*get-conflict W* = *None*›; *cases brkW*)
by (*auto simp*: *twl-restart-inv-def struct-invs-S struct-invs-T struct-invs-U that*)
qed
have *noinp-continue*: ‹*?I* (*xd*, *False* ∨ *get-conflict ab* ≠ *None*, *ab*, *ab*, *ab*, *m*, *Suc n*)› (**is** *?A*) **and**
noinp-term: ‹((*xd*, *False* ∨ *get-conflict ab* ≠ *None*, *ab*, *ab*, *ab*, *m*, *Suc n*), *ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*) ∈ *?term*›
(**is** *?B*)
if
‹*restart-prog-pre-int S T W False*› **and**
‹*size* (*get-all-learned-clss S*) ≤ *size* (*get-all-learned-clss W*)› **and**
less: ‹*True* ⟶
f n < *size* (*get-all-learned-clss W*) − *size* (*get-all-learned-clss S*)› **and**

```
    ‹size (get-all-learned-clss T) ≤ size (get-all-learned-clss W)› and
    ‹xa ⟶ size (get-all-learned-clss T) < size (get-all-learned-clss W)› and
    ‹¬ (xa ∧ ¬ False)› and
    WX: ‹cdcl-twl-restart W ab› and
    ‹xd ∈ UNIV› and
    [simp]: x ‹¬ brkW› ‹¬ xb›
    for x xa xb ab xd
  proof −
    have [simp]: ‹get-conflict W = None› ‹get-conflict ab = None› ‹clauses-to-update ab = {#}›
      using that(1) WX
      by (auto simp: restart-prog-pre-int-def cdcl-twl-restart.simps)
    have WX': ‹cdcl-twl-stgy-restart (S, T, W, m, n, True) (ab, ab, ab, m, Suc n, True)›
      by (rule cdcl-twl-stgy-restart.intros(2)[of - - - ab])
        (use less WX in ‹auto dest!: cdcl-twl-inp.intros›)
    then show ?A
      using UW twl-res by (auto simp add: cdcl-twl-stgy-restart-prog-int-inv-def)
    show ?B
      using UW twl-res WX' ‹twl-restart-inv (S₀, T₀, U₀, m₀, n₀, True)› by (auto
        dest: rtranclp-cdcl-twl-stgy-restart-twl-restart-inv)
  qed

  show ?thesis
    unfolding restart-prog-int-def restart-required-def GC-required-def inprocessing-required-def
    apply (refine-vcg lhs-step-If; remove-dummy-vars)
    subgoal by (rule restart-W)
    subgoal by (auto simp: restart-prog-pre-int-def)
    subgoal by (auto simp: restart-prog-pre-int-def)
    subgoal by (rule restart-only)
    subgoal by (rule restart-only-term)
    subgoal by (rule noinp-continue)
    subgoal by (rule noinp-term)
    subgoal by (rule GC)
    subgoal by (rule GC-term)
    subgoal by (rule continue) auto
    subgoal by (rule continue-term) auto
    done
qed


lemma (in twl-restart) cdcl-twl-stgy-prog-bounded-int-spec:
  fixes n :: nat
  assumes ‹twl-restart-inv (S, T, U, m, n, False)› and
    ‹twl-stgy-restart-inv (S, T, U, m, n, False)› and
    ‹clauses-to-update U = {#}› and
    ‹get-conflict U = None›
  shows
    ‹cdcl-twl-stgy-restart-prog-bounded-intg m n S T U ≤ partial-conclusive-TWL-run2 m n S T U›
  supply RETURN-as-SPEC-refine[refine2 del]
  unfolding cdcl-twl-stgy-restart-prog-bounded-intg-def full-def partial-conclusive-TWL-run2-def
  apply (refine-vcg
    WHILEIT-rule[where
    R = ‹cdcl-algo-termination-early-rel›];
    remove-dummy-vars)
  subgoal
    by (rule wf-cdcl-algo-termination-early-rel)
  subgoal
```

using *assms* **by** (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def twl-restart-inv-def*
twl-struct-invs-def pcdcl-stgy-restart-inv-def twl-stgy-restart-inv-def)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def*
twl-restart-inv-def
dest!: *rtranclp-cdcl-twl-stgy-restart-twl-restart-inv*)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def*)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def*)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def*
twl-restart-inv-def
dest: *rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs*)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def*
twl-restart-inv-def no-step-cdcl-twl-cp-no-step-cdcl$_W$-cp
dest: *rtranclp-cdcl-twl-stgy-restart-twl-restart-inv*)
subgoal by *fast*
subgoal for *x a aa ab ac ad ae be xa*
using *assms*
using
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[*of* ‹(*S*, *T*, *U*, *m*, *n*, *True*)›
‹(*ab*, *ac*, *ad*, *ae*, *be*, *True*)›]
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def*
twl-restart-inv-def
intro: *rtranclp-cdcl-twl-stgy-entailed-by-init*
dest!: *rtranclp-cdcl-twl-cp-stgyD*
simp flip: *state$_W$-of-def*)
subgoal
by (*rule restart-prog-bounded-spec*)
subgoal for *x brk T U m n ebrk V*
apply (*rule-tac x*=*T* **in** *exI*)
apply (*rule-tac x*=*U* **in** *exI*)
apply (*rule-tac x*= ‹(*m*, *n*, ¬*brk*)› **in** *exI*)
by (*auto simp add*: *cdcl-twl-stgy-restart-prog-int-inv-def*)
done

lemma *cdcl-twl-stgy-restart-prog-int-spec*:
fixes S_0 :: ‹'*v twl-st*›
assumes ‹*twl-restart-inv* (S_0, T_0, U_0, m_0, n_0, *False*)› **and**
‹*twl-stgy-restart-inv* (S_0, T_0, U_0, m_0, n_0, *False*)› **and**
‹*clauses-to-update* U_0 = {#}› **and**
‹*get-conflict* U_0 = *None*›
shows
‹*cdcl-twl-stgy-restart-prog-intg* m_0 n_0 S_0 T_0 U_0 ≤ *conclusive-TWL-run2* m_0 n_0 S_0 T_0 U_0›
proof −
define *RETURN-FALSE* **where** ‹*RETURN-FALSE* = *RETURN False*›
have *cdcl-twl-stgy-restart-prog-alt-def*:
‹*cdcl-twl-stgy-restart-prog-intg* m_0 n_0 S_0 T_0 U_0 =
do {
- ← *RETURN False*;
(*brk*, -, -, *T*, -, -) ← *WHILE*$_T$*cdcl-twl-stgy-restart-prog-int-inv* (S_0, T_0, U_0, m_0, n_0)
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*, *S'*, *S''*, *m*, *n*).

do {
T ← *unit-propagation-outer-loop S''*;
(*brk*, *T*) ← *cdcl-twl-o-prog T*;
(*S*, *S'*, *T*, *m'*, *n'*) ← *restart-prog-int S S' T m n brk*;
- ← *RETURN-FALSE*;
RETURN (*brk* ∨ *get-conflict T* ≠ *None*, *S*, *S'*, *T*, *m'*, *n'*)
})
(*False*, S_0, T_0, U_0, m_0, n_0);
RETURN T
}›
unfolding *cdcl-twl-stgy-restart-prog-int-def RETURN-FALSE-def*
by *auto*

have [*refine*]: ‹*RETURN False* ≤ ⇓ {(*b*, *b'*). (*b* = *b'*) ∧ ¬*b*} (*RES UNIV*)›
‹*RETURN-FALSE* ≤ ⇓ {(*b*, *b'*). (*b* = *b'*) ∧ ¬*b*} (*RES UNIV*)›
by (*auto intro*!: *RETURN-RES-refine simp*: *RETURN-FALSE-def*)
have [*refine*]: ‹((*False*, S_0, T_0, U_0, (m_0::*nat*), (n_0 :: *nat*)), *ebrk*, *False*, S_0, T_0, U_0, m_0, n_0) ∈ {(*T*, (*b*, *S*)). *S* = *T* ∧ ¬*b*}›
if ‹(*u*, *ebrk*) ∈ {(*b*, *b'*). (*b* = *b'*) ∧ ¬*b*}› **for** *ebrk u*
using *that*
by *auto*
have *this-is-the-identity*: ‹*x* ≤ ⇓*Id* (*x'*)› **if** ‹*x* = *x'*›
for *x x'*
using *that* **by** *auto*

have *ref-early*: ‹*cdcl-twl-stgy-restart-prog-intg* m_0 n_0 S_0 T_0 U_0 ≤ ⇓{((*S*), (*ebrk*, *T*)). *S* = *T* ∧ ¬*ebrk*}
(*cdcl-twl-stgy-restart-prog-bounded-intg* m_0 n_0 S_0 T_0 U_0)›
unfolding *cdcl-twl-stgy-restart-prog-alt-def cdcl-twl-stgy-restart-prog-bounded-intg-def*
apply *refine-rcg*
apply *assumption*
subgoal by *auto*
subgoal by *auto*
apply (*rule this-is-the-identity*)
subgoal by *auto*
apply (*rule this-is-the-identity*)
subgoal by *auto*
apply (*rule this-is-the-identity*)
subgoal by *simp*
subgoal by *auto*
subgoal by *auto*
done

show *?thesis*
apply (*rule order-trans*[*OF ref-early*])
apply (*rule order-trans*[*OF ref-two-step'*])
apply (*rule cdcl-twl-stgy-prog-bounded-int-spec*[*OF assms*])
unfolding *conc-fun-RES partial-conclusive-TWL-run2-def conclusive-TWL-run2-def* **by** *fastforce*
qed

end

context *twl-restart-ops*
begin

definition (**in** −) *restart-prog-pre* :: ‹'*v twl-st* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-prog-pre S last-GC last-Restart brk* ⟷ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧

```
    (¬brk ⟶ get-conflict S = None) ∧
    size (get-all-learned-clss S) ≥ last-Restart ∧
    size (get-all-learned-clss S) ≥ last-GC ∧
    cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S)›

definition restart-prog
  :: ‹'v twl-st ⇒ nat ⇒ nat ⇒ nat ⇒ bool ⇒ ('v twl-st × nat × nat × nat) nres›
where
  ‹restart-prog S last-GC last-Restart n brk = do {
     ASSERT(restart-prog-pre S last-GC last-Restart brk);
     b ← GC-required S last-GC  n;
     b2 ← restart-required S last-Restart  n;
     if b2 ∧ ¬brk then do {
       T ← SPEC(λT. cdcl-twl-restart-only S T);
       RETURN (T, last-GC, (size (get-all-learned-clss T)), n)
     }
     else
     if b ∧ ¬brk then do {
         b ← inprocessing-required S;
         if ¬b then  do {
           V ← SPEC(λU. cdcl-twl-restart S U);
           RETURN (V, (size (get-all-learned-clss V)), (size (get-all-learned-clss V)), Suc n)
       } else do {
           T ← SPEC(λT. cdcl-twl-inp** S T ∧ count-decided (get-trail T) = 0);
           RETURN (T, (size (get-all-learned-clss T)), (size (get-all-learned-clss T)), Suc n)
        }
     }
     else
       RETURN (S, last-GC, last-Restart, n)
   }›

lemma restart-prog-spec:
  ‹(uncurry4 restart-prog, uncurry5 restart-prog-int) ∈
  {(((((S, last-GC), last-Restart), n), brk),
  (((((last-GC', last-Restart'), S'), m'), n'), brk')).
  S = S' ∧ last-GC = size (get-all-learned-clss last-GC') ∧
  last-Restart = size (get-all-learned-clss last-Restart') ∧
  n = n' ∧ brk = brk'} →_f
  ⟨{((S, last-GC, last-Restart, n),
   (last-GC', last-Restart', S', m', n')).
  S = S' ∧ last-GC = size (get-all-learned-clss last-GC') ∧
  last-Restart = size (get-all-learned-clss last-Restart') ∧
  n = n'}⟩nres-rel›
proof −
  have this-is-the-identity: ‹x ≤ ⇓Id (x')› if ‹x = x'› for x x'
    using that by auto
  show ?thesis
    unfolding restart-prog-def restart-prog-int-def uncurry-def
    apply (intro frefI nres-relI)
    apply refine-rcg
    subgoal
      by (auto simp: restart-prog-pre-def restart-prog-pre-int-def)
    apply (rule this-is-the-identity)
    subgoal
      by auto
    apply (rule this-is-the-identity)
```

```
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    apply (rule this-is-the-identity)
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    done
qed

fun cdcl-twl-stgy-restart-prog-inv:: 'v twl-st × nat ⇒ (bool × 'v twl-st × nat × nat ×nat) ⇒ bool where
 [simp del]: ‹cdcl-twl-stgy-restart-prog-inv (S₀, n₀)(brk, T, last-GC, last-Restart, n) ⟷
    (∃ last-GC' last-Restart' m m₀ T₀ U₀. last-GC = size (get-all-learned-clss last-GC') ∧
      last-Restart = size (get-all-learned-clss last-Restart') ∧
     cdcl-twl-stgy-restart-prog-int-inv (T₀, U₀, S₀, m₀, n₀) (brk, last-GC', last-Restart', T, m, n))›

lemma cdcl-twl-stgy-restart-prog-inv-def:
  ‹cdcl-twl-stgy-restart-prog-inv= (λ(S₀, n₀) (brk, T, last-GC, last-Restart, n).
    (∃ last-GC' last-Restart' m m₀ T₀ U₀. last-GC = size (get-all-learned-clss last-GC') ∧
      last-Restart = size (get-all-learned-clss last-Restart') ∧
  cdcl-twl-stgy-restart-prog-int-inv (T₀, U₀, S₀, m₀, n₀) (brk, last-GC', last-Restart', T, m, n)))›
  by (force intro!: ext simp: cdcl-twl-stgy-restart-prog-inv.simps)

definition cdcl-twl-stgy-restart-progg
  :: nat ⇒ nat ⇒ nat ⇒ 'v twl-st ⇒ 'v twl-st nres
where
  ‹cdcl-twl-stgy-restart-progg n₀ last-GC last-Restart S₀ =
  do {
    (brk, T, -, -, -) ← WHILE_T^cdcl-twl-stgy-restart-prog-inv (S₀, n₀)
      (λ(brk, -). ¬brk)
      (λ(brk, S'', S, S', n).
      do {
        T ← unit-propagation-outer-loop S'';
        (brk, T) ← cdcl-twl-o-prog T;
        (T, S, S', n') ← restart-prog T S S' n brk;
        RETURN (brk ∨ get-conflict T ≠ None, T, S, S', n')
      })
      (False, S₀, last-GC, last-Restart, n₀);
```

```
    RETURN T
  }›

abbreviation cdcl-twl-stgy-restart-prog where
  ‹cdcl-twl-stgy-restart-prog S ≡
    cdcl-twl-stgy-restart-progg 0 (size (get-all-learned-clss S)) (size (get-all-learned-clss S)) S›

lemmas cdcl-twl-stgy-restart-prog-def = cdcl-twl-stgy-restart-progg-def

lemma (in −) fref-to-Down-curry4-5:
  ‹(uncurry4 f, uncurry5 g) ∈ [P]_f A → ⟨B⟩nres-rel ⟹
    (⋀x x' y y' z z' a a' b b' c'. P (((((x', y'), z'), a'), b'), c') ⟹
      (((((x, y), z), a), b), (((((x', y'), z'), a'), b'), c')) ∈ A ⟹
       f x y z a b ≤ ⇓ B (g x' y' z' a' b' c'))›
  unfolding fref-def uncurry-def nres-rel-def
  by auto

lemma cdcl-twl-stgy-restart-progg-cdcl-twl-stgy-restart-prog-intg:
  ‹cdcl-twl-stgy-restart-progg n_0 (size (get-all-learned-clss T)) (size (get-all-learned-clss U)) S ≤ ⇓Id
(cdcl-twl-stgy-restart-prog-intg m_0 n_0 T U S)›
proof −
  have this-is-the-identity: ‹x ≤ ⇓Id (x')› if ‹x = x'› for x x'
    using that by auto
  show ?thesis
    unfolding cdcl-twl-stgy-restart-prog-def cdcl-twl-stgy-restart-prog-int-def uncurry-def
    apply (refine-rcg
      WHILEIT-refine[where R = ‹{((brk :: bool,  S :: 'v twl-st, last-GC, last-Restart, n),
     (brk', last-GC', last-Restart', S', m', n')).
     S = S' ∧ last-GC = size (get-all-learned-clss last-GC') ∧
     last-Restart = size (get-all-learned-clss last-Restart') ∧
       n = n' ∧ brk = brk'}›]
       restart-prog-spec[THEN fref-to-Down-curry4-5])
    subgoal
      by auto
    subgoal
      unfolding cdcl-twl-stgy-restart-prog-inv-def by blast
    subgoal
      by auto
    apply (rule this-is-the-identity)
    subgoal
      by auto
    apply (rule this-is-the-identity)
    subgoal
      by auto
    subgoal
      by simp
    subgoal
      by simp
    subgoal
      by simp
    subgoal
      by simp
    done
qed

lemma cdcl-twl-stgy-restart-prog-cdcl-twl-stgy-restart-prog-int:
```

‹(cdcl-twl-stgy-restart-prog, cdcl-twl-stgy-restart-prog-int) ∈ Id →$_f$ ⟨Id⟩nres-rel›
proof −
show *?thesis*
unfolding *uncurry-def Down-id-eq*
apply (*intro frefI nres-relI*)
apply (*rule order-trans*[*OF cdcl-twl-stgy-restart-progg-cdcl-twl-stgy-restart-prog-intg*])
apply *auto*
done
qed

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-specg*:
fixes S_0 :: ‹'v twl-st›
assumes ‹twl-restart-inv (T_0, U_0, S_0, m_0, n_0, *False*)› **and**
‹twl-stgy-restart-inv (T_0, U_0, S_0, m_0, n_0, *False*)› **and**
‹clauses-to-update S_0 = {#}› **and**
‹get-conflict S_0 = None›
shows
‹cdcl-twl-stgy-restart-progg n_0 (size (get-all-learned-clss T_0)) (size (get-all-learned-clss U_0)) S_0 ≤ conclusive-TWL-run2 m_0 n_0 T_0 U_0 S_0›
apply (*rule order-trans*)
apply (*rule cdcl-twl-stgy-restart-progg-cdcl-twl-stgy-restart-prog-intg*[*of - - - - m_0*])
apply *simp*
apply (*rule cdcl-twl-stgy-restart-prog-int-spec*[*OF assms*])
done

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-spec*:
fixes *S* :: ‹'v twl-st›
assumes
‹twl-struct-invs S› **and** ‹twl-stgy-invs S› **and** ‹clauses-to-update S = {#}› **and**
‹get-conflict S = None› ‹cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (state$_W$-of S)›
shows
‹cdcl-twl-stgy-restart-prog S ≤ conclusive-TWL-run S›
by (*rule order-trans*[*OF cdcl-twl-stgy-restart-prog-specg*[*of - - - 0*]])
(*use assms* **in** ‹*force simp*: *twl-restart-inv-def pcdcl-stgy-restart-inv-def*
twl-struct-invs-def twl-stgy-restart-inv-def conclusive-TWL-run2-def
conclusive-TWL-run-def›)+

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-prog-boundedg*
:: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'v twl-st* ⇒ (*bool* × *'v twl-st*) *nres*
where
‹cdcl-twl-stgy-restart-prog-boundedg n_0 last-GC last-Restart S_0=
do {
ebrk ← *RES UNIV*;
(*ebrk*, -, *T*, -, -, -) ← *WHILE*$_T$^cdcl-twl-stgy-restart-prog-inv (S_0, n_0) *o snd*
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, *S*, *last-GC*, *last-Restart*, *n*).
do {
T ← *unit-propagation-outer-loop S*;
(*brk*, *T*) ← *cdcl-twl-o-prog T*;
(*T*, *last-GC*, *last-Restart*, *n*) ← *restart-prog T last-GC last-Restart n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk* ∨ *get-conflict T* ≠ *None*, *T*, *last-GC*, *last-Restart*, *n*)
})
(*ebrk*, *False*, S_0, *last-GC*, *last-Restart*, n_0);

```
    RETURN (ebrk, T)
  }›

abbreviation cdcl-twl-stgy-restart-prog-bounded where
 ‹cdcl-twl-stgy-restart-prog-bounded S ≡ cdcl-twl-stgy-restart-prog-boundedg 0 (size (get-all-learned-clss
S))
      (size (get-all-learned-clss S)) S›

lemmas cdcl-twl-stgy-restart-prog-bounded-def =
  cdcl-twl-stgy-restart-prog-boundedg-def

lemma cdcl-twl-stgy-restart-prog-boundedg-cdcl-twl-stgy-restart-prog-bounded-intg:
  ‹cdcl-twl-stgy-restart-prog-boundedg n₀ (size (get-all-learned-clss T₀))
      (size (get-all-learned-clss U₀)) S ≤ ⇓Id (cdcl-twl-stgy-restart-prog-bounded-intg m₀ n₀ T₀ U₀ S)›
proof −
  have this-is-the-identity: ‹x ≤ ⇓Id (x')› if ‹x = x'› for x x'
    using that by auto
  show ?thesis
    unfolding cdcl-twl-stgy-restart-prog-bounded-def cdcl-twl-stgy-restart-prog-bounded-intg-def
      uncurry-def
    apply (refine-rcg
      WHILEIT-refine[where R = ‹{((ebrk :: bool, brk :: bool,  S :: 'v twl-st, last-GC, last-Restart, n),
     (ebrk', brk', last-GC', last-Restart', S', m', n')).
    S = S' ∧ last-GC = size (get-all-learned-clss last-GC') ∧
    last-Restart = size (get-all-learned-clss last-Restart') ∧
      n = n' ∧ brk = brk' ∧ ebrk = ebrk'}›]
      restart-prog-spec[THEN fref-to-Down-curry4-5])
    subgoal
      by auto
    subgoal for ebrk ebrka x x'
      unfolding cdcl-twl-stgy-restart-prog-inv-def prod.case case-prod-beta comp-def
      apply (rule exI[of - ‹fst (snd (snd ((x'))))›])
      apply (rule exI[of - ‹fst (snd (snd (snd ((x')))))›])
      apply (rule-tac exI[of - ‹fst (snd (snd (snd (snd (snd (x'))))))›])
      apply (rule-tac exI[of - m₀])
      apply (rule-tac exI[of - T₀])
      apply (rule-tac exI[of - U₀])
      by simp
    subgoal
      by auto
    apply (rule this-is-the-identity)
    subgoal
      by auto
    apply (rule this-is-the-identity)
    subgoal
      by auto
    subgoal
      by simp
    subgoal
      by simp
    subgoal
      by simp
    subgoal
      by simp
    done
qed
```

lemma *cdcl-twl-stgy-restart-prog-bounded-cdcl-twl-stgy-restart-prog-bounded-int*:
‹(*cdcl-twl-stgy-restart-prog-bounded*, *cdcl-twl-stgy-restart-prog-bounded-int*) ∈ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
proof −
have *this-is-the-identity*: ‹*x* ≤ ⇓*Id* (*x'*)› **if** ‹*x* = *x'*› **for** *x x'*
using *that* **by** *auto*
show *?thesis*
unfolding *uncurry-def*
apply (*intro frefI nres-relI*)
apply (*rule order-trans*[*OF cdcl-twl-stgy-restart-prog-boundedg-cdcl-twl-stgy-restart-prog-bounded-intg*])
apply (*auto simp add*:)
done
qed
thm *twl-restart-inv-def*
lemma (**in** *twl-restart*) *cdcl-twl-stgy-prog-bounded-spec*:
assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and**
‹*get-conflict S* = *None*› ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)›
shows
‹*cdcl-twl-stgy-restart-prog-bounded S* ≤ *partial-conclusive-TWL-run S*›
apply (*rule order-trans*)
apply (*rule cdcl-twl-stgy-restart-prog-bounded-cdcl-twl-stgy-restart-prog-bounded-int*[*THEN fref-to-Down*,
of - S])
apply *simp*
apply *simp*
apply (*rule order-trans*[*OF ref-two-step'*])
apply (*rule cdcl-twl-stgy-prog-bounded-int-spec*)
apply ((*use assms* **in** ‹*auto simp*: *twl-restart-inv-def pcdcl-stgy-restart-inv-def*
twl-struct-invs-def twl-stgy-restart-inv-def partial-conclusive-TWL-run2-def
partial-conclusive-TWL-run-def›)+)[4]
unfolding *partial-conclusive-TWL-run2-def partial-conclusive-TWL-run-def*
by *fastforce*

definition *cdcl-twl-stgy-restart-prog-early-intg*
:: *'v twl-st* ⇒ *'v twl-st* ⇒ *'v twl-st* ⇒ *'v twl-st nres*
where
‹*cdcl-twl-stgy-restart-prog-early-intg S*$_0$ *T*$_0$ *U*$_0$=
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, *last-GC*, *last-Restart*, *T*,*m*, *n*) ← *WHILE*$_T$*cdcl-twl-stgy-restart-prog-int-inv* (*S*$_0$, *T*$_0$, *U*$_0$, *0*, *0*) *o snd*
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, *last-GC*, *last-Restart*, *S*, *m*, *n*).
do {
T ← *unit-propagation-outer-loop S*;
(*brk*, *T*) ← *cdcl-twl-o-prog T*;
(*last-GC*, *last-Restart*, *T*, *m*, *n*) ← *restart-prog-int last-GC last-Restart T m n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk* ∨ *get-conflict T* ≠ *None*, *last-GC*, *last-Restart*, *T*, *m*, *n*)
})
(*ebrk*, *False*, *S*$_0$, *T*$_0$, *U*$_0$, *0*, *0*);
if ¬*brk then do* {
(*brk*, *last-GC*, *last-Restart*, *T*, -, -) ← *WHILE*$_T$*cdcl-twl-stgy-restart-prog-int-inv* (*last-GC*, *last-Restart*, *T*, *m*, *n*)
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *last-GC*, *last-Restart*, *S*, *m*, *n*).
do {

```
   T ← unit-propagation-outer-loop S;
   (brk, T) ← cdcl-twl-o-prog T;
   (last-GC, last-Restart, T, m, n) ← restart-prog-int last-GC last-Restart T m n brk;
   RETURN (brk ∨ get-conflict T ≠ None, last-GC, last-Restart, T, m, n)
 })
 (False, last-GC, last-Restart, T, m, n);
     RETURN T
    }
    else RETURN T
 }›

lemmas cdcl-twl-stgy-restart-prog-early-int-def =
  cdcl-twl-stgy-restart-prog-early-intg-def

lemma cdcl-twl-stgy-restart-prog-early-intg-alt-def:
  ‹cdcl-twl-stgy-restart-prog-early-intg S_0  T_0 U_0 =
  do {
    ebrk ← RES UNIV;
   (ebrk, brk, last-GC, last-Restart, T,m, n) ← WHILE_T^cdcl-twl-stgy-restart-prog-int-inv (S_0, T_0, U_0, 0, 0) o snd
      (λ(ebrk, brk, -). ¬brk ∧ ¬ebrk)
      (λ(ebrk, brk, last-GC, last-Restart, S, m, n).
      do {
        T ← unit-propagation-outer-loop S;
        (brk, T) ← cdcl-twl-o-prog T;
        (last-GC, last-Restart, T, m, n) ← restart-prog-int last-GC last-Restart T m n brk;
 ebrk ← RES UNIV;
        RETURN (ebrk, brk ∨ get-conflict T ≠ None, last-GC, last-Restart, T, m, n)
      })
      (ebrk, False, S_0, T_0, U_0, 0, 0);
    if ¬brk then do {
      cdcl-twl-stgy-restart-prog-intg m n last-GC last-Restart T
    } else RETURN T
  }›
   unfolding cdcl-twl-stgy-restart-prog-intg-def cdcl-twl-stgy-restart-prog-early-intg-def
   by (auto intro!: bind-cong[OF refl])

definition cdcl-twl-stgy-restart-prog-early :: 'v twl-st ⇒ 'v twl-st nres where
  ‹cdcl-twl-stgy-restart-prog-early S_0 =
  do {
    ebrk ← RES UNIV;
   (ebrk, brk, T, last-GC, last-Restart, n) ← WHILE_T^cdcl-twl-stgy-restart-prog-inv (S_0, 0) o snd
      (λ(ebrk, brk, -). ¬brk ∧ ¬ebrk)
      (λ(ebrk, brk, S, last-GC, last-Restart, n).
      do {
        T ← unit-propagation-outer-loop S;
        (brk, T) ← cdcl-twl-o-prog T;
        (T, last-GC, last-Restart, n) ← restart-prog T last-GC last-Restart n brk;
 ebrk ← RES UNIV;
        RETURN (ebrk, brk ∨ get-conflict T ≠ None, T, last-GC, last-Restart, n)
      })
      (ebrk, False, S_0, size (get-all-learned-clss S_0), size (get-all-learned-clss S_0), 0);
    if ¬brk then do {
      (brk, T, last-GC, last-Restart, -) ← WHILE_T^cdcl-twl-stgy-restart-prog-inv (T, n)
 (λ(brk, -). ¬brk)
 (λ(brk, S, last-GC, last-Restart, n).
```

```
 do {
   T ← unit-propagation-outer-loop S;
   (brk, T) ← cdcl-twl-o-prog T;
   (T, last-GC, last-Restart, n) ← restart-prog T last-GC last-Restart n brk;
   RETURN (brk ∨ get-conflict T ≠ None, T, last-GC, last-Restart, n)
 })
 (False, T, last-GC, last-Restart, n);
     RETURN T
    }
    else RETURN T
  }›

abbreviation cdcl-twl-stgy-restart-prog-early-int where
  ‹cdcl-twl-stgy-restart-prog-early-int S ≡ cdcl-twl-stgy-restart-prog-early-intg S S S›


lemma (in twl-restart) cdcl-twl-stgy-prog-early-int-spec:
  assumes ‹twl-struct-invs S› and ‹twl-stgy-invs S› and ‹clauses-to-update S = {#}› and
    ‹get-conflict S = None› ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S)›
  shows
    ‹cdcl-twl-stgy-restart-prog-early-int S ≤ conclusive-TWL-run S›
proof −
  show ?thesis
    unfolding cdcl-twl-stgy-restart-prog-early-intg-alt-def conclusive-TWL-run-def
    apply (refine-vcg
      cdcl-twl-stgy-restart-prog-int-spec[THEN order-trans]
      WHILEIT-rule[where
      R = ‹cdcl-algo-termination-early-rel›];
      remove-dummy-vars)
    subgoal
      by (rule wf-cdcl-algo-termination-early-rel)
    subgoal
      using assms by (auto simp: cdcl-twl-stgy-restart-prog-int-inv-def twl-restart-inv-def
        twl-struct-invs-def pcdcl-stgy-restart-inv-def twl-stgy-restart-inv-def)
    subgoal
      by (auto simp: cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def
        twl-restart-inv-def
        dest!: rtranclp-cdcl-twl-stgy-restart-twl-restart-inv)
    subgoal
      by (auto simp: cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def)
    subgoal
      by (auto simp: cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def)
    subgoal
      by (auto simp: cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def
        twl-restart-inv-def
        dest: rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs)
    subgoal
      by (auto simp: cdcl-twl-stgy-restart-prog-int-inv-def twl-stgy-restart-inv-def
        twl-restart-inv-def no-step-cdcl-twl-cp-no-step-cdcl_W-cp
        dest: rtranclp-cdcl-twl-stgy-restart-twl-restart-inv)
    subgoal by fast
    subgoal for x a aa ab ac ad ae be xa
      using assms
      using
        rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[of ‹(S, S, S, 0, 0, True)›
        ‹(ab, ac, ad, ae, be, True)›]
```

by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-def*
twl-restart-inv-def
intro: *rtranclp-cdcl-twl-stgy-entailed-by-init*
dest!: *rtranclp-cdcl-twl-cp-stgyD*
simp flip: $state_W$*-of-def*)
subgoal
by (*rule restart-prog-bounded-spec*)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-alt-def twl-restart-inv-def*
twl-stgy-restart-inv-def pcdcl-stgy-restart-inv-def)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-alt-def twl-restart-inv-def*
twl-stgy-restart-inv-def pcdcl-stgy-restart-inv-def)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-alt-def twl-restart-inv-def*
twl-stgy-restart-inv-def pcdcl-stgy-restart-inv-def)
subgoal
by (*auto simp*: *cdcl-twl-stgy-restart-prog-int-inv-alt-def twl-restart-inv-def*
twl-stgy-restart-inv-def pcdcl-stgy-restart-inv-def)
subgoal for *x a aa ab ac ad ae be*
unfolding *conclusive-TWL-run2-def cdcl-twl-stgy-restart-prog-int-inv-def*
cdcl-twl-stgy-restart-prog-int-inv-alt-def comp-def snd-conv
apply (*rule SPEC-rule*)
apply *normalize-goal+*
apply (*rule-tac x=xb* **in** *exI*)
apply (*rule-tac x=xc* **in** *exI*)
apply (*rule-tac x=xd* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply *auto*
done
subgoal for *x a aa ab ac ad ae be*
unfolding *conclusive-TWL-run2-def cdcl-twl-stgy-restart-prog-int-inv-def*
cdcl-twl-stgy-restart-prog-int-inv-alt-def comp-def snd-conv
apply *normalize-goal+*
apply (*rule-tac x=ab* **in** *exI*)
apply (*rule-tac x=ac* **in** *exI*)
apply (*rule-tac x=* ‹(*ae*, *be*, ¬*aa*)› **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply *auto*
done
done
qed

lemma *cdcl-twl-stgy-restart-prog-early-cdcl-twl-stgy-restart-prog-early*:
‹*cdcl-twl-stgy-restart-prog-early S* ≤ ⇓*Id* (*cdcl-twl-stgy-restart-prog-early-int S*)›
proof −
have *this-is-the-identity*: ‹*x* ≤ ⇓*Id* (*x'*)› **if** ‹*x* = *x'*› **for** *x x'*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-early-def cdcl-twl-stgy-restart-prog-early-int-def*
uncurry-def
apply (*refine-rcg*
WHILEIT-refine[**where** *R* = ‹{((*ebrk* :: *bool*, *brk* :: *bool*, *S* :: *'v twl-st*, *last-GC*, *last-Restart*, *n*),
(*ebrk'*, *brk'*, *last-GC'*, *last-Restart'*, *S'*, *m'*, *n'*)).

S = *S'* ∧ *last-GC* = *size* (*get-all-learned-clss last-GC'*) ∧
last-Restart = *size* (*get-all-learned-clss last-Restart'*) ∧
n = *n'* ∧ *brk* = *brk'* ∧ *ebrk* = *ebrk'*}⟩]
WHILEIT-refine[**where** *R* = ⟨{((*brk* :: *bool*, *S* :: '*v twl-st*, *last-GC*, *last-Restart*, *n*),
(*brk'*, *last-GC'*, *last-Restart'*, *S'*, *m'*, *n'*)).
S = *S'* ∧ *last-GC* = *size* (*get-all-learned-clss last-GC'*) ∧
last-Restart = *size* (*get-all-learned-clss last-Restart'*) ∧
n = *n'* ∧ *brk* = *brk'*}⟩]
restart-prog-spec[*THEN fref-to-Down-curry4-5*])
subgoal
by *auto*
subgoal for *ebrk ebrka x x'*
unfolding *cdcl-twl-stgy-restart-prog-inv-def prod.case case-prod-beta comp-def*
apply (*rule exI*[*of* - ⟨*fst* (*snd* (*snd* ((*x'*))))⟩])
apply (*rule exI*[*of* - ⟨*fst* (*snd* (*snd* (*snd* ((*x'*)))))⟩])
apply (*rule-tac exI*[*of* - ⟨*fst* (*snd* (*snd* (*snd* (*snd* (*snd* (*x'*))))))⟩])
apply (*rule-tac exI*[*of* - *0*])
apply (*rule-tac exI*[*of* - *S*])
apply (*rule-tac exI*[*of* - *S*])
by *simp*
subgoal
by *auto*
apply (*rule this-is-the-identity*)
subgoal
by *auto*
apply (*rule this-is-the-identity*)
subgoal
by *auto*
subgoal
by *simp*
subgoal
by *simp*
subgoal
by *simp*
subgoal
by *simp*
subgoal
by *auto*
subgoal for *ebrk ebrka x x' x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h x1i x2i x1j x2j xa x'a*
unfolding *cdcl-twl-stgy-restart-prog-inv-def prod.case case-prod-beta comp-def*
apply (*rule exI*[*of* - ⟨*fst* (*snd* (((*x'a*))))⟩])
apply (*rule exI*[*of* - ⟨*fst* (*snd* (*snd* (((*x'a*)))))⟩])
apply (*rule-tac exI*[*of* - ⟨*fst* ((*snd* (*snd* (*snd* (*snd* (*x'a*))))))⟩])
apply (*rule-tac exI*[*of* - *x1e*])
apply (*rule-tac exI*[*of* - *x1b*])
apply (*rule-tac exI*[*of* - *x1c*])
apply (*cases x'a*)
by *simp*
subgoal
by *auto*
apply (*rule this-is-the-identity*)
subgoal
by *auto*
apply (*rule this-is-the-identity*)
subgoal

by *auto*
subgoal
by *simp*
subgoal
by *simp*
subgoal
by *simp*
subgoal
by *simp*
subgoal
by *simp*
done
qed

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-early-spec*:
assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and**
‹*get-conflict S* = *None*› ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)›
shows
‹*cdcl-twl-stgy-restart-prog-early S* ≤ *conclusive-TWL-run S*›
apply (*rule order-trans*[*OF cdcl-twl-stgy-restart-prog-early-cdcl-twl-stgy-restart-prog-early*])
apply (*subst Down-id-eq*)
apply (*rule cdcl-twl-stgy-prog-early-int-spec*[*OF assms*])
done

end

end
theory *Watched-Literals-List-Inprocessing*
imports *Watched-Literals-List Watched-Literals-Algorithm-Reduce*
begin

definition *all-learned-lits-of-l* :: ‹'*v twl-st-l* ⇒ '*v clause*› **where**
‹*all-learned-lits-of-l S'* ≡ *all-lits-of-mm* (*mset* ‘# *learned-clss-lf* (*get-clauses-l S'*) + *get-unit-learned-clss-l S'* +
get-subsumed-learned-clauses-l S' + *get-learned-clauses0-l S'*)›
definition *all-init-lits-of-l* :: ‹'*v twl-st-l* ⇒ '*v clause*› **where**
‹*all-init-lits-of-l S'* ≡ *all-lits-of-mm* (*mset* ‘# *get-init-clss-l S'* + *get-unit-init-clauses-l S'* +
get-subsumed-init-clauses-l S' + *get-init-clauses0-l S'*)›

inductive *cdcl-twl-subsumed-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
subsumed-II:
‹*cdcl-twl-subsumed-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop C' N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *C'*)) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*mset* (*N* ∝ *C*) ⊆# *mset* (*N* ∝ *C'*)› ‹*C* ∈# *dom-m N*› ‹*C'* ∈# *dom-m N*›
‹*irred N C'*› ‹*irred N C*› ‹*C* ≠ *C'*› ‹*C'* ∉ *set* (*get-all-mark-of-propagated M*)›|
subsumed-RR:
‹*cdcl-twl-subsumed-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop C' N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *C'*)) *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*mset* (*N* ∝ *C*) ⊆# *mset* (*N* ∝ *C'*)› ‹*C* ∈# *dom-m N*› ‹*C'* ∈# *dom-m N*›
‹¬*irred N C'*› ‹¬*irred N C*›‹*C* ≠ *C'*› ‹*C'* ∉ *set* (*get-all-mark-of-propagated M*)›|
subsumed-IR:
‹*cdcl-twl-subsumed-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop C' N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *C'*)) *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*mset* (*N* ∝ *C*) ⊆# *mset* (*N* ∝ *C'*)› ‹*C* ∈# *dom-m N*› ‹*C'* ∈# *dom-m N*›
‹*irred N C*› ‹¬*irred N C'*› ‹*C'* ∉ *set* (*get-all-mark-of-propagated M*)› |
subsumed-RI:

⟨*cdcl-twl-subsumed-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmupd C* (*N* ∝ *C*, *True*) (*fmdrop C′ N*), *D*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *C′*))
NS, *US*, *N0*, *U0*, {#}, *Q*)⟩
if ⟨*mset* (*N* ∝ *C*) ⊆# *mset* (*N* ∝ *C′*)⟩ ⟨*C* ∈# *dom-m N*⟩ ⟨*C′* ∈# *dom-m N*⟩
⟨¬*irred N C*⟩ ⟨*irred N C′*⟩ ⟨*C′* ∉ *set* (*get-all-mark-of-propagated M*)⟩
⟨¬*tautology* (*mset* (*N* ∝ *C*))⟩
⟨*distinct-mset* (*mset* (*N* ∝ *C*))⟩

lemma *convert-lits-l-drop*:
⟨*C* ∉ *set* (*get-all-mark-of-propagated M*) ⟹
(*M*, *M′*) ∈ *convert-lits-l* (*fmdrop C N*) *E* ⟷ (*M*, *M′*) ∈ *convert-lits-l N E*⟩
unfolding *convert-lits-l-def list-rel-def in-pair-collect-simp*
apply (*rule iffI*; (*rule list.rel-mono-strong*, *assumption*))
apply (*auto simp*: *convert-lits-l-def list-rel-def p2rel-def convert-lit.simps*
cdcl$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*)
done

lemma *convert-lits-l-update-sel*:
⟨*C* ∈# *dom-m N* ⟹ *C′* = *N* ∝ *C* ⟹
(*M*, *M′*) ∈ *convert-lits-l* (*fmupd C* (*C′*, *b*) *N*) *E* ⟷ (*M*, *M′*) ∈ *convert-lits-l N E*⟩
unfolding *convert-lits-l-def list-rel-def in-pair-collect-simp*
apply (*rule iffI*; (*rule list.rel-mono-strong*, *assumption*))
apply (*auto simp*: *convert-lits-l-def list-rel-def p2rel-def convert-lit.simps*
cdcl$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*)
done

lemma *learned-clss-l-mapsto-upd-in-irrelev*: ⟨*C* ∈# *dom-m N* ⟹ ¬*irred N C* ⟹
learned-clss-l (*fmupd C* (*C′*, *True*) *N*) = *remove1-mset* (*N* ∝ *C*, *irred N C*) (*learned-clss-l N*)⟩
by (*auto simp*: *ran-m-mapsto-upd-notin ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd-irrelev*:
⟨*C* ∈# *dom-m N* ⟹ ¬*irred N C* ⟹
init-clss-l (*fmupd C* (*C′*, *True*) *N*) = *add-mset* (*C′*, *True*) ((*init-clss-l N*))⟩
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *cdcl-twl-subsumed-l-cdcl-twl-subsumed*:
assumes ⟨*cdcl-twl-subsumed-l S T*⟩ **and**
SS′: ⟨(*S*, *S′*) ∈ *twl-st-l None*⟩
shows ⟨∃ *T′*. (*T*, *T′*) ∈ *twl-st-l None* ∧ *cdcl-twl-subsumed S′ T′*⟩
proof −
obtain *M′ N′ U′ D′ NE′ UE′ NS′ US′ WS′ N0′ U0′ Q′* **where**
S′: ⟨*S′* = (*M′*, *N′*, *U′*, *D′*, *NE′*, *UE′*,*NS′*, *US′*, *N0′*, *U0′*, *WS′*, *Q′*)⟩
by (*cases S′*)

show *?thesis*
using *assms*(*1*)
proof (*cases rule*: *cdcl-twl-subsumed-l.cases*)
case (*subsumed-II N C C′ M D NE UE NEk UEk NS US N0 U0 Q*)
define *N′′* **where** ⟨*N′′* = (*N′* − {#*twl-clause-of* (*N* ∝ *C*), *twl-clause-of* (*N* ∝ *C′*)#})⟩
let *?E* = ⟨*twl-clause-of* (*N* ∝ *C*)⟩
let *?E′* = ⟨*twl-clause-of* (*N* ∝ *C′*)⟩
have ⟨(*N* ∝ *C*, *irred N C*) ∈# *init-clss-l N*⟩ **and**
H: ⟨(*N* ∝ *C′*, *irred N C′*) ∈# *remove1-mset* (*N* ∝ *C*, *irred N C*)(*init-clss-l N*)⟩
using *subsumed-II*
apply (*auto dest*!: *multi-member-split simp*: *ran-m-def add-mset-eq-add-mset remove1-mset-add-mset-If*
split: *if-splits*)

```
    apply (metis prod.collapse)+
    done
  from multi-member-split[OF this(1)] multi-member-split[OF this(2)]
  have ‹N' = N'' + {#twl-clause-of (N ∝ C), twl-clause-of (N ∝ C')#}›
    using subsumed-II SS' unfolding N''-def
    by (auto simp: ran-m-def S' twl-st-l-def add-mset-eq-add-mset dest!: multi-member-split)
  then show ?thesis
    using SS' subsumed-II H unfolding S'
    apply (auto simp: add-mset-eq-add-mset image-mset-Diff conj-disj-distribL
      conj-disj-distribR eq-commute[of ‹twl-clause-of -›] eq-commute[of ‹remove1-mset - -›]
    S' convert-lits-l-drop learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if
      eq-commute[of ‹add-mset - (add-mset - -)› ‹image-mset - -›]
      twl-st-l-def mset-take-mset-drop-mset' learned-clss-l-l-fmdrop-irrelev
      simp del: twl-clause-of.simps
      simp flip: twl-clause-of.simps
      dest!: intro!: exI[of - ‹get-trail S'›])
    apply (auto simp: init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if
      eq-commute[of ‹add-mset - (add-mset - -)› ‹image-mset - -›]
      twl-st-l-def mset-take-mset-drop-mset' learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop
      intro!: cdcl-twl-subsumed-II-simp[where C = ?E and C' = ?E'])
    done
next
  case (subsumed-RR N C C' M D NE UE NEk UEk NS US N0 U0 Q)
  define U'' where ‹U'' = (U' - {#twl-clause-of (N ∝ C), twl-clause-of (N ∝ C')#})›
  let ?E = ‹twl-clause-of (N ∝ C)›
  let ?E' = ‹twl-clause-of (N ∝ C')›
  have ‹(N ∝ C, irred N C) ∈# learned-clss-l N› and
    H: ‹(N ∝ C', irred N C') ∈# remove1-mset (N ∝ C, irred N C)(learned-clss-l N)›
    using subsumed-RR
  apply (auto dest!: multi-member-split simp: ran-m-def add-mset-eq-add-mset remove1-mset-add-mset-If
      split: if-splits)
    apply (metis prod.collapse)+
    done
  from multi-member-split[OF this(1)] multi-member-split[OF this(2)]
  have ‹U' = U'' + {#twl-clause-of (N ∝ C), twl-clause-of (N ∝ C')#}›
    using subsumed-RR SS' unfolding U''-def
    by (auto simp: ran-m-def S' twl-st-l-def add-mset-eq-add-mset dest!: multi-member-split)
  then show ?thesis
    using SS' subsumed-RR H unfolding S'
    apply (auto simp: add-mset-eq-add-mset image-mset-Diff conj-disj-distribL
      conj-disj-distribR eq-commute[of ‹twl-clause-of -›] eq-commute[of ‹remove1-mset - -›]
    S' convert-lits-l-drop learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if
      eq-commute[of ‹add-mset - (add-mset - -)› ‹image-mset - -›]
      twl-st-l-def mset-take-mset-drop-mset'
      simp del: twl-clause-of.simps
      simp flip: twl-clause-of.simps
      dest!: intro!: exI[of - ‹get-trail S'›])
    apply (auto simp: init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if
      eq-commute[of ‹add-mset - (add-mset - -)› ‹image-mset - -›]
      twl-st-l-def mset-take-mset-drop-mset' learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop
      learned-clss-l-l-fmdrop
      cdcl-twl-subsumed-RR-simp[where C = ?E and C' = ?E'])
    done
next
  case (subsumed-IR N C C' M D NE UE NEk UEk NS US N0 U0 Q)
  define U'' where ‹U'' = (U' - {#twl-clause-of (N ∝ C')#})›
```

define N'' **where** ‹$N'' = (N' - \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C)\#\})$›
let $?E$ = ‹$\mathit{twl\text{-}clause\text{-}of}\ (N \propto C)$›
let $?E'$ = ‹$\mathit{twl\text{-}clause\text{-}of}\ (N \propto C')$›
have ‹$(N \propto C, \mathit{irred}\ N\ C) \in\# \mathit{init\text{-}clss\text{-}l}\ N$› **and**
H: ‹$(N \propto C', \mathit{irred}\ N\ C') \in\# (\mathit{learned\text{-}clss\text{-}l}\ N)$›
using *subsumed-IR*
apply (*auto dest*!: *multi-member-split simp*: *ran-m-def add-mset-eq-add-mset remove1-mset-add-mset-If split*: *if-splits*)
apply (*metis prod.collapse*)+
done
from *multi-member-split*[*OF this*(*1*)] *multi-member-split*[*OF this*(*2*)]
have ‹$U' = U'' + \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C')\#\}$› ‹$N' = N'' + \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C)\#\}$›
using *subsumed-IR SS'* **unfolding** *U''-def N''-def*
by (*auto simp*: *ran-m-def S' twl-st-l-def add-mset-eq-add-mset dest*!: *multi-member-split*)
then show *?thesis*
using *SS' subsumed-IR H* **unfolding** *S'*
apply (*auto simp*: *add-mset-eq-add-mset image-mset-Diff conj-disj-distribL conj-disj-distribR eq-commute*[*of* ‹*twl-clause-of* -›] *eq-commute*[*of* ‹*remove1-mset* - -›]
S' convert-lits-l-drop learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if
eq-commute[*of* ‹*add-mset* - (*add-mset* - -)› ‹*image-mset* - -›]
twl-st-l-def mset-take-mset-drop-mset'
simp del: *twl-clause-of.simps*
simp flip: *twl-clause-of.simps*
dest!: *intro*!: *exI*[*of* - ‹*get-trail S'*›])
apply (*auto simp*: *init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if*
eq-commute[*of* ‹*add-mset* - (*add-mset* - -)› ‹*image-mset* - -›]
twl-st-l-def mset-take-mset-drop-mset' learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop
learned-clss-l-l-fmdrop
cdcl-twl-subsumed-IR-simp[**where** C = *?E* **and** C' = *?E'*])
done
next
case (*subsumed-RI N C C' M D NE UE NEk UEk NS US N0 U0 Q*)
define U'' **where** ‹$U'' = (U' - \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C)\#\})$›
define N'' **where** ‹$N'' = (N' - \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C')\#\})$›
let $?E$ = ‹$\mathit{twl\text{-}clause\text{-}of}\ (N \propto C)$›
let $?E'$ = ‹$\mathit{twl\text{-}clause\text{-}of}\ (N \propto C')$›
have ‹$(N \propto C', \mathit{irred}\ N\ C') \in\# \mathit{init\text{-}clss\text{-}l}\ N$› **and**
H: ‹$(N \propto C, \mathit{irred}\ N\ C) \in\# (\mathit{learned\text{-}clss\text{-}l}\ N)$› **and**
[*simp*,*iff*]: ‹$C \in\# \mathit{remove1\text{-}mset}\ C'\ (\mathit{dom\text{-}m}\ N)$› ‹$\neg\ \mathit{irred}\ (\mathit{fmdrop}\ C'\ N)\ C$› ‹$C \neq C'$›
‹$(N \propto C, \mathit{False}) \in\# \mathit{learned\text{-}clss\text{-}l}\ N$›
using *subsumed-RI*
apply (*auto dest*!: *multi-member-split simp*: *ran-m-def add-mset-eq-add-mset remove1-mset-add-mset-If split*: *if-splits*)
apply (*metis prod.collapse*)+
done
from *multi-member-split*[*OF this*(*1*)] *multi-member-split*[*OF this*(*2*)]
have ‹$U' = U'' + \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C)\#\}$› ‹$N' = N'' + \{\#\mathit{twl\text{-}clause\text{-}of}\ (N \propto C')\#\}$›
using *subsumed-RI SS'* **unfolding** *U''-def N''-def*
by (*auto simp*: *ran-m-def S' twl-st-l-def add-mset-eq-add-mset dest*!: *multi-member-split*)
then show *?thesis*
using *SS' subsumed-RI H* **unfolding** *S'*
apply (*auto simp*: *add-mset-eq-add-mset image-mset-Diff conj-disj-distribL conj-disj-distribR eq-commute*[*of* ‹*twl-clause-of* -›] *eq-commute*[*of* ‹*remove1-mset* - -›]
S' convert-lits-l-drop learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if
eq-commute[*of* ‹*add-mset* - (*add-mset* - -)› ‹*image-mset* - -›] *convert-lits-l-update-sel*
twl-st-l-def mset-take-mset-drop-mset'

simp del: *twl-clause-of.simps*
simp flip: *twl-clause-of.simps*
dest!: *intro*!: *exI*[*of* - ‹*get-trail S'*›])
apply (*auto simp*: *init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if*
eq-commute[*of* ‹*add-mset* - (*add-mset* - -)› ‹*image-mset* - -›]
twl-st-l-def mset-take-mset-drop-mset' learned-clss-l-l-fmdrop-irrelev init-clss-l-fmdrop
learned-clss-l-l-fmdrop convert-lits-l-drop learned-clss-l-mapsto-upd-in-irrelev
init-clss-l-mapsto-upd init-clss-l-mapsto-upd-irrelev
intro!: *cdcl-twl-subsumed-RI-simp*[**where** *C* = *?E* **and** *C'* = *?E'*])
done
qed
qed

lemma *cdcl-twl-subsumed-l-list-invs*:
‹*cdcl-twl-subsumed-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
apply (*induction rule*: *cdcl-twl-subsumed-l.induct*)
apply (*auto simp*: *twl-list-invs-def cdcl$_W$-restart-mset.in-get-all-mark-of-propagated-in-trail*
ran-m-mapsto-upd distinct-mset-remove1-All distinct-mset-dom
ran-m-lf-fmdrop
dest: *in-diffD*)
apply (*subst* (*asm*) *ran-m-mapsto-upd*)
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remove1-All distinct-mset-dom*
ran-m-lf-fmdrop)
done

inductive *cdcl-twl-subresolution-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
twl-subresolution-II-nonunit:
‹*cdcl-twl-subresolution-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmupd C'* (*E*, *True*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *C'*)) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if
‹*C* ∈# *dom-m N*›
‹*C'* ∈# *dom-m N*›
‹*mset* (*N* ∝ *C*) = *add-mset L D*›
‹*mset* (*N* ∝ *C'*) = *add-mset* (−*L*) *D'*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D'*›
‹*mset E* = *remdups-mset D'*› ‹*length E* ≥ *2*› ‹*distinct E*› ‹∀ *L* ∈# *mset E*. *undefined-lit M L*›
‹*C'* ∉ *set* (*get-all-mark-of-propagated M*)› ‹*irred N C*› ‹*irred N C'*› |
twl-subresolution-II-unit:
‹*cdcl-twl-subresolution-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K 0* # *M*, *fmdrop C' N*, *None*, *NE*, *UE*, *add-mset* {#*K*#} *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *C'*)) *NS*, *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
if
‹*C* ∈# *dom-m N*›
‹*C'* ∈# *dom-m N*›
‹*mset* (*N* ∝ *C*) = *add-mset L D*›
‹*mset* (*N* ∝ *C'*) = *add-mset* (−*L*) *D'*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D'*›
‹*remdups-mset D'* = {#*K*#}›
‹*undefined-lit M K*›
‹*C'* ∉ *set* (*get-all-mark-of-propagated M*)› ‹*irred N C*› ‹*irred N C'*› |
twl-subresolution-LL-nonunit:
‹*cdcl-twl-subresolution-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmupd C'* (*E*, *False*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *C'*)) *US*, *N0*, *U0*, {#}, *Q*)›
if

‹$C \in\# \mathit{dom\text{-}m}\ N$›
‹$C' \in\# \mathit{dom\text{-}m}\ N$›
‹$\mathit{mset}\ (N \propto C) = \mathit{add\text{-}mset}\ L\ D$›
‹$\mathit{mset}\ (N \propto C') = \mathit{add\text{-}mset}\ (-L)\ D'$›
‹$\mathit{count\text{-}decided}\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg \mathit{tautology}\ (D + D')$›
‹$\mathit{mset}\ E = \mathit{remdups\text{-}mset}\ D'$› ‹$\mathit{length}\ E \geq 2$› ‹$\mathit{distinct}\ E$› ‹$\forall L \in\# \mathit{mset}\ E.\ \mathit{undefined\text{-}lit}\ M\ L$›
‹$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M)$› ‹$\neg \mathit{irred}\ N\ C$› ‹$\neg \mathit{irred}\ N\ C'$› |
twl-subresolution-LL-unit:
‹$\mathit{cdcl\text{-}twl\text{-}subresolution\text{-}l}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
($\mathit{Propagated}\ K\ 0\ \#\ M,\ \mathit{fmdrop}\ C'\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{add\text{-}mset}\ \{\#K\#\}\ \mathit{UEk},\ \mathit{NS},\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ \mathit{add\text{-}mset}\ (-K)\ Q$)›
if
‹$C \in\# \mathit{dom\text{-}m}\ N$›
‹$C' \in\# \mathit{dom\text{-}m}\ N$›
‹$\mathit{mset}\ (N \propto C) = \mathit{add\text{-}mset}\ L\ D$›
‹$\mathit{mset}\ (N \propto C') = \mathit{add\text{-}mset}\ (-L)\ D'$›
‹$\mathit{count\text{-}decided}\ M = 0$› ‹$D \subseteq\# D'$›
‹$\mathit{remdups\text{-}mset}\ D' = \{\#K\#\}$›
‹$\mathit{undefined\text{-}lit}\ M\ K$›
‹$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M)$› ‹$\neg \mathit{irred}\ N\ C$› ‹$\neg \mathit{irred}\ N\ C'$› |
twl-subresolution-LI-nonunit:
‹$\mathit{cdcl\text{-}twl\text{-}subresolution\text{-}l}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
($M,\ \mathit{fmupd}\ C'\ (E,\ \mathit{False})\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q$)›
if
‹$C \in\# \mathit{dom\text{-}m}\ N$›
‹$C' \in\# \mathit{dom\text{-}m}\ N$›
‹$\mathit{mset}\ (N \propto C) = \mathit{add\text{-}mset}\ L\ D$›
‹$\mathit{mset}\ (N \propto C') = \mathit{add\text{-}mset}\ (-L)\ D'$›
‹$\mathit{count\text{-}decided}\ M = 0$› ‹$D \subseteq\# D'$›
‹$\mathit{mset}\ E = \mathit{remdups\text{-}mset}\ D'$› ‹$\mathit{length}\ E \geq 2$› ‹$\mathit{distinct}\ E$› ‹$\forall L \in\# \mathit{mset}\ E.\ \mathit{undefined\text{-}lit}\ M\ L$›
‹$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M)$› ‹$\mathit{irred}\ N\ C$› ‹$\neg \mathit{irred}\ N\ C'$› |
twl-subresolution-LI-unit:
‹$\mathit{cdcl\text{-}twl\text{-}subresolution\text{-}l}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
($\mathit{Propagated}\ K\ 0\ \#\ M,\ \mathit{fmdrop}\ C'\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{add\text{-}mset}\ \{\#K\#\}\ \mathit{UEk},\ \mathit{NS},\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ \mathit{add\text{-}mset}\ (-K)\ Q$)›
if
‹$C \in\# \mathit{dom\text{-}m}\ N$›
‹$C' \in\# \mathit{dom\text{-}m}\ N$›
‹$\mathit{mset}\ (N \propto C) = \mathit{add\text{-}mset}\ L\ D$›
‹$\mathit{mset}\ (N \propto C') = \mathit{add\text{-}mset}\ (-L)\ D'$›
‹$\mathit{count\text{-}decided}\ M = 0$› ‹$D \subseteq\# D'$›
‹$\mathit{remdups\text{-}mset}\ D' = \{\#K\#\}$›
‹$\mathit{undefined\text{-}lit}\ M\ K$›
‹$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M)$› ‹$\mathit{irred}\ N\ C$› ‹$\neg \mathit{irred}\ N\ C'$› |
twl-subresolution-IL-nonunit:
‹$\mathit{cdcl\text{-}twl\text{-}subresolution\text{-}l}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
($M,\ \mathit{fmupd}\ C'\ (E,\ \mathit{True})\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q$)›
if
‹$C \in\# \mathit{dom\text{-}m}\ N$›
‹$C' \in\# \mathit{dom\text{-}m}\ N$›
‹$\mathit{mset}\ (N \propto C) = \mathit{add\text{-}mset}\ L\ D$›
‹$\mathit{mset}\ (N \propto C') = \mathit{add\text{-}mset}\ (-L)\ D'$›
‹$\mathit{count\text{-}decided}\ M = 0$› ‹$D \subseteq\# D'$›
‹$\mathit{mset}\ E = \mathit{remdups\text{-}mset}\ D'$› ‹$\mathit{length}\ E \geq 2$› ‹$\mathit{distinct}\ E$› ‹$\forall L \in\# \mathit{mset}\ E.\ \mathit{undefined\text{-}lit}\ M\ L$›

‹$C' \notin$ *set* (*get-all-mark-of-propagated* M)› ‹¬*irred* N C› ‹*irred* N C'› |
twl-subresolution-IL-unit:
‹*cdcl-twl-subresolution-l* (M, N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, Q)
(*Propagated* K 0 # M, *fmdrop* C' N, *None*, *NE*, *UE*, *add-mset* {#K#} *NEk*, *UEk*, *add-mset* (*mset* ($N \propto C'$)) *NS*, *US*, *N0*, *U0*, {#}, *add-mset* ($-K$) Q)›
if
‹$C \in\#$ *dom-m* N›
‹$C' \in\#$ *dom-m* N›
‹*mset* ($N \propto C$) = *add-mset* L D›
‹*mset* ($N \propto C'$) = *add-mset* ($-L$) D'›
‹*count-decided* M = 0› ‹$D \subseteq\#$ D'›
‹*remdups-mset* D' = {#K#}›
‹*undefined-lit* M K›
‹$C' \notin$ *set* (*get-all-mark-of-propagated* M)› ‹¬*irred* N C› ‹*irred* N C'›

lemma *convert-lits-l-update-sel2*:
‹$C \in\#$ *dom-m* $N \Longrightarrow C \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
(M, M') $\in$ *convert-lits-l* (*fmupd* C (C', b) N) $E \longleftrightarrow$ (M, M') $\in$ *convert-lits-l* N E›
unfolding *convert-lits-l-def list-rel-def in-pair-collect-simp*
apply (*rule iffI*; (*rule list.rel-mono-strong*, *assumption*))
apply (*auto simp*: *convert-lits-l-def list-rel-def p2rel-def convert-lit.simps*
cdcl$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*)
done

lemma *twl-subresolution-II-nonunit-simps*:
‹*cdcl-twl-subresolution* S T›
if
‹S = (M, N, U, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, Q)›
‹T = (M, *add-mset* E (*remove1-mset* C' N), U, *None*, *NE*, *UE*, *add-mset* (*clause* C') *NS*, *US*, *N0*, *U0*, {#}, Q)›
‹*clause* C = *add-mset* L D›
‹*clause* C' = *add-mset* ($-L$) D'›
‹*count-decided* M = 0› ‹$D \subseteq\#$ D'› ‹¬*tautology* (D + D')›
‹*clause* E = *remdups-mset* D'› ‹*size* (*watched* E) = 2› ‹*struct-wf-twl-cls* E›
‹$\forall$ $L \in\#$ *clause* E. *undefined-lit* M L›
‹$C \in\#$ N›
‹$C' \in\#$ N›
using *cdcl-twl-subresolution.twl-subresolution-II-nonunit*[*of* C L D C' D' M E ‹N − {#C, C'#}› U] *that*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-II-unit-simps*:
‹*cdcl-twl-subresolution* S T›
if
‹S = (M, N, U, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, Q)›
‹T = (*Propagated* K {#K#} # M, (*remove1-mset* C' N), U, *None*, *add-mset* {#K#} *NE*, *UE*, *add-mset* (*clause* C') *NS*, *US*, *N0*, *U0*, {#}, *add-mset* ($-K$) Q)›
‹*clause* C = *add-mset* L D›
‹*clause* C' = *add-mset* ($-L$) D'›
‹*count-decided* M = 0› ‹$D \subseteq\#$ D'› ‹¬*tautology* (D + D')›
‹*remdups-mset* D' = {#K#}›
‹*undefined-lit* M K›
‹$C \in\#$ N›
‹$C' \in\#$ N›
using *cdcl-twl-subresolution.twl-subresolution-II-unit*[*of* C L D C' D' M K ‹N − {#C, C'#}› U] *that*

by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-LL-nonunit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*T* = (*M*, *N*, *add-mset E* (*remove1-mset C′ U*), *None*, *NE*, *UE*, *NS*, *add-mset* (*clause C′*) *US*, *N0*, *U0*, {#}, *Q*)›
‹*clause C* = *add-mset L D*›
‹*clause C′* = *add-mset* (−*L*) *D′*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D′*› ‹¬*tautology* (*D* + *D′*)›
‹*clause E* = *remdups-mset D′*› ‹*size* (*watched E*) = *2*› ‹*struct-wf-twl-cls E*›
‹∀ *L* ∈# *clause E*. *undefined-lit M L*›
‹*C* ∈# *U*›
‹*C′* ∈# *U*›
using *cdcl-twl-subresolution.twl-subresolution-LL-nonunit*[*of C L D C′ D′ M E N* ‹*U* − {#*C*, *C′*#}›]
that
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-LL-unit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*T* = (*Propagated K* {#*K*#} # *M*, (*N*), *remove1-mset C′ U*, *None*, *NE*, *add-mset* {#*K*#} *UE*, *NS*, *add-mset* (*clause C′*) *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
‹*clause C* = *add-mset L D*›
‹*clause C′* = *add-mset* (−*L*) *D′*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D′*› ‹¬*tautology* (*D* + *D′*)›
‹*remdups-mset D′* = {#*K*#}›
‹*undefined-lit M K*›
‹*C* ∈# *U*›
‹*C′* ∈# *U*›
using *cdcl-twl-subresolution.twl-subresolution-LL-unit*[*of C L D C′ D′ M K N* ‹*U* − {#*C*, *C′*#}›]
that
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-LI-nonunit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*T* = (*M*, *N*, *add-mset E* (*remove1-mset C′ U*), *None*, *NE*, *UE*, *NS*, *add-mset* (*clause C′*) *US*, *N0*, *U0*, {#}, *Q*)›
‹*clause C* = *add-mset L D*›
‹*clause C′* = *add-mset* (−*L*) *D′*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D′*› ‹¬*tautology* (*D* + *D′*)›
‹*clause E* = *remdups-mset D′*› ‹*size* (*watched E*) = *2*› ‹*struct-wf-twl-cls E*›
‹∀ *L* ∈# *clause E*. *undefined-lit M L*›
‹*C* ∈# *N*›
‹*C′* ∈# *U*›
using *cdcl-twl-subresolution.twl-subresolution-LI-nonunit*[*of C L D C′ D′ M E* ‹*N* − {#*C*#}› ‹*U* − {#*C′*#}›] *that*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-LI-unit-simps*:
‹*cdcl-twl-subresolution S T*›
if

‹$S = (M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$T = (Propagated\ K\ \{\#K\#\}\ \#\ M,\ (N),\ remove1\text{-}mset\ C'\ U,\ None,\ NE,\ add\text{-}mset\ \{\#K\#\}\ UE,$
$NS,\ add\text{-}mset\ (clause\ C')\ US,\ N0,\ U0,\ \{\#\},\ add\text{-}mset\ (-K)\ Q)$›
‹$clause\ C = add\text{-}mset\ L\ D$›
‹$clause\ C' = add\text{-}mset\ (-L)\ D'$›
‹$count\text{-}decided\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg tautology\ (D + D')$›
‹$remdups\text{-}mset\ D' = \{\#K\#\}$›
‹$undefined\text{-}lit\ M\ K$›
‹$C \in\# N$›
‹$C' \in\# U$›
using *cdcl-twl-subresolution.twl-subresolution-LI-unit*[*of C L D C' D' M K* ‹$N - \{\#C\#\}$› ‹$U - \{\#C'\#\}$›] *that*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-IL-nonunit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹$S = (M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$T = (M,\ add\text{-}mset\ E\ (remove1\text{-}mset\ C'\ N),\ U,\ None,\ NE,\ UE,\ add\text{-}mset\ (clause\ C')\ NS,\ US,\ N0,$
$U0,\ \{\#\},\ Q)$›
‹$clause\ C = add\text{-}mset\ L\ D$›
‹$clause\ C' = add\text{-}mset\ (-L)\ D'$›
‹$count\text{-}decided\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg tautology\ (D + D')$›
‹$clause\ E = remdups\text{-}mset\ D'$› ‹$size\ (watched\ E) = 2$› ‹*struct-wf-twl-cls E*›
‹$\forall L \in\# clause\ E.\ undefined\text{-}lit\ M\ L$›
‹$C' \in\# N$›
‹$C \in\# U$›
using *cdcl-twl-subresolution.twl-subresolution-IL-nonunit*[*of C L D C' D' M E* ‹$N - \{\#C'\#\}$› ‹$U - \{\#C\#\}$›] *that*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *twl-subresolution-IL-unit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹$S = (M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$T = (Propagated\ K\ \{\#K\#\}\ \#\ M,\ remove1\text{-}mset\ C'\ N,\ \ U,\ None,\ add\text{-}mset\ \{\#K\#\}\ NE,\ UE,$
$add\text{-}mset\ (clause\ C')\ NS,\ US,\ N0,\ U0,\ \{\#\},\ add\text{-}mset\ (-K)\ Q)$›
‹$clause\ C = add\text{-}mset\ L\ D$›
‹$clause\ C' = add\text{-}mset\ (-L)\ D'$›
‹$count\text{-}decided\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg tautology\ (D + D')$›
‹$remdups\text{-}mset\ D' = \{\#K\#\}$›
‹$undefined\text{-}lit\ M\ K$›
‹$C' \in\# N$›
‹$C \in\# U$›
using *cdcl-twl-subresolution.twl-subresolution-IL-unit*[*of C L D C' D' M K* ‹$N - \{\#C'\#\}$› ‹$U - \{\#C\#\}$›] *that*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset add-mset-commute*)

lemma *cdcl-twl-subresolution-l-cdcl-twl-subresolution*:
assumes ‹*cdcl-twl-subresolution-l S T*› **and**
SS': ‹$(S,\ S') \in twl\text{-}st\text{-}l\ None$› **and**
list-invs: ‹*twl-list-invs S*›
shows ‹$\exists T'.\ (T,\ T') \in twl\text{-}st\text{-}l\ None \wedge cdcl\text{-}twl\text{-}subresolution\ S'\ T'$›
proof −
have *tauto*: ‹$\forall C \in \#dom\text{-}m\ (get\text{-}clauses\text{-}l\ S).\ \neg tautology\ (mset\ (get\text{-}clauses\text{-}l\ S \propto C))$›
using *list-invs* **unfolding** *twl-list-invs-def*

```
    by (auto simp: dest!: multi-member-split)
  have H1: ‹C ∈# dom-m N ⟹
    C' ∈# dom-m N ⟹
    mset (N ∝ C) = add-mset L D ⟹
    mset (N ∝ C') = add-mset (- L) D' ⟹
    D ⊆# D' ⟹
    ∀ C∈#dom-m N. ¬ tautology (mset (N ∝ C)) ⟹ tautology (D + D') ⟹ False› for C C' L D N
D'
   by (metis mset-subset-eqD tautology-add-mset tautology-minus tautology-union)
  from assms tauto show ?thesis
    apply -
    supply [simp] = convert-lits-l-update-sel2
    apply (induction rule: cdcl-twl-subresolution-l.induct)
    subgoal for C N C' L D D' M E NE UE NEk UEk NS US N0 U0 Q
      using H1[of C N C' L D D']
      apply (auto simp: twl-st-l-def)
      apply (rule-tac x = x in exI)
      apply auto
      apply (rule twl-subresolution-II-nonunit-simps[where
          C' = ‹(TWL-Clause (mset (watched-l (N ∝ C'))) (mset (unwatched-l (N ∝ C'))))› and
        C = ‹(TWL-Clause (mset (watched-l (N ∝ C))) (mset (unwatched-l (N ∝ C))))›])
      apply (auto simp: init-clss-l-mapsto-upd image-mset-remove1-mset-if
        learned-clss-l-mapsto-upd-irrel
        mset-take-mset-drop-mset'
        )
      done
    subgoal for C N C' L D D' M K NE UE NS US N0 U0 Q
      using H1[of C N C' L D D']
      apply (auto simp: twl-st-l-def)
      apply (rule-tac x = ‹Propagated K {#K#} # x› in exI)
      apply (auto simp: convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset)
      apply (rule twl-subresolution-II-unit-simps[where
          C' = ‹(TWL-Clause (mset (watched-l (N ∝ C'))) (mset (unwatched-l (N ∝ C'))))› and
          C = ‹(TWL-Clause (mset (watched-l (N ∝ C))) (mset (unwatched-l (N ∝ C))))›])
      apply (auto simp: init-clss-l-mapsto-upd image-mset-remove1-mset-if
        learned-clss-l-mapsto-upd-irrel
        mset-take-mset-drop-mset'
        init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev; fail
        )+
      done
    subgoal for C N C' L D D' M E NE UE NS US N0 U0 Q
      apply (auto simp: twl-st-l-def)
      apply (rule-tac x = ‹x› in exI)
      apply (auto simp: convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset
        intro!: twl-subresolution-LL-nonunit-simps[where
          C' = ‹(TWL-Clause (mset (watched-l (N ∝ C'))) (mset (unwatched-l (N ∝ C'))))› and
        C = ‹(TWL-Clause (mset (watched-l (N ∝ C))) (mset (unwatched-l (N ∝ C))))›]
        simp:  init-clss-l-mapsto-upd image-mset-remove1-mset-if
        learned-clss-l-mapsto-upd-irrel
        mset-take-mset-drop-mset'
        init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
        init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
        learned-clss-l-mapsto-upd-irrel
        mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
          init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
          init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
```

learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev; *fail*
)+
done
subgoal for *C N C′ L D D′ M K NE UE NS US N0 U0 Q*
using *H1*[*of C N C′ L D D′*]
apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x* = ‹*Propagated K* {#*K*#} # *x*› **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *twl-subresolution-LL-unit-simps*[**where**
C′ = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C′*))) (*mset* (*unwatched-l* (*N* ∝ *C′*))))› **and**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
)
done
subgoal for *C N C′ L D D′ M E NE UE NS US N0 U0 Q*
using *H1*[*of C N C′ L D D′*]
apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x* = ‹*get-trail S′*› **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *twl-subresolution-LI-nonunit-simps*[**where**
C′ = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C′*))) (*mset* (*unwatched-l* (*N* ∝ *C′*))))› **and**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev; *fail*
)+
done
subgoal for *C N C′ L D D′ M K NE UE NS US N0 U0 Q*
using *H1*[*of C N C′ L D D′*]
apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x* = ‹*Propagated K* {#*K*#} # *x*› **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *twl-subresolution-LI-unit-simps*[**where**
C′ = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C′*))) (*mset* (*unwatched-l* (*N* ∝ *C′*))))› **and**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel

mset-take-mset-drop-mset'
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
)
done
subgoal for *C N C' L D D' M E NE UE NS US N0 U0 Q*
using *H1*[*of C N C' L D D'*]
apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x* = ‹*get-trail S'*› **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *twl-subresolution-IL-nonunit-simps*[**where**
C' = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C'*))) (*mset* (*unwatched-l* (*N* ∝ *C'*))))› **and**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset'
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev; *fail*
)+
done
subgoal for *C N C' L D D' M K NE UE NS US N0 U0 Q*
using *H1*[*of C N C' L D D'*]
apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x* = ‹*Propagated K* {#*K*#} # *x*› **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *twl-subresolution-IL-unit-simps*[**where**
C' = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C'*))) (*mset* (*unwatched-l* (*N* ∝ *C'*))))› **and**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset'
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
)
done
done

qed

inductive *cdcl-twl-unitres-true-l* :: ‹'v *twl-st-l* ⇒ 'v *twl-st-l* ⇒ *bool*› **where**
‹*cdcl-twl-unitres-true-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop C N*, *None*, *add-mset* (*mset* (*N* ∝ *C*)) *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*L* ∈# *mset* (*N* ∝ *C*)› ‹*get-level M L* = *0*› ‹*L* ∈ *lits-of-l M*›
‹*C* ∈# *dom-m N*› ‹*irred N C*›
‹*C* ∉ *set* (*get-all-mark-of-propagated M*)› |
‹*cdcl-twl-unitres-true-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop C N*, *None*, *NE*, *add-mset* (*mset* (*N* ∝ *C*)) *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if ‹*L* ∈# *mset* (*N* ∝ *C*)› ‹*get-level M L* = *0*› ‹*L* ∈ *lits-of-l M*›
‹*C* ∈# *dom-m N*› ‹¬*irred N C*›
‹*C* ∉ *set* (*get-all-mark-of-propagated M*)›

lemma *cdcl-twl-unitres-true-intro1*:
‹*cdcl-twl-unitres-true S T*›
if ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*T* = (*M*, *remove1-mset C N*, *U*, *None*, *add-mset* (*clause C*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*L* ∈# *clause C*› ‹*get-level M L* = *0*› ‹*L* ∈ *lits-of-l M*› ‹*C* ∈# *N*›
using *cdcl-twl-unitres-true.intros*(*1*)[*of L C M* ‹*N* − {#*C*#}› *U*] *that*
by *auto*

lemma *cdcl-twl-unitres-true-intro2*:
‹*cdcl-twl-unitres-true S T*›
if ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*T* = (*M*, *N*, *remove1-mset C U*, *None*, *NE*, *add-mset* (*clause C*) *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*L* ∈# *clause C*› ‹*get-level M L* = *0*› ‹*L* ∈ *lits-of-l M*› ‹*C* ∈# *U*›
using *cdcl-twl-unitres-true.intros*(*2*)[*of L C M N* ‹*U* − {#*C*#}›] *that*
by *auto*

lemma *cdcl-twl-unitres-true-l-cdcl-twl-unitres-true*:
assumes ‹*cdcl-twl-unitres-true-l S T*› **and**
SS': ‹(*S*, *S'*) ∈ *twl-st-l None*›
shows ‹∃ *T'*. (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-unitres-true S' T'*›
using *assms*
apply (*induction rule*: *cdcl-twl-unitres-true-l.induct*)
subgoal for *L N C M NE UE NS US Q*
apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *cdcl-twl-unitres-true-intro1*[**where**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset'
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel convert-lits-l-drop
mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
)
done
subgoal for *L N C M NE UE NS US N0 U0 Q*

apply (*auto simp*: *twl-st-l-def*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*auto simp*: *convert-lit.simps convert-lits-l-drop convert-lits-l-add-mset*
intro!: *cdcl-twl-unitres-true-intro2*[**where**
C = ‹(*TWL-Clause* (*mset* (*watched-l* (*N* ∝ *C*))) (*mset* (*unwatched-l* (*N* ∝ *C*))))›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
learned-clss-l-mapsto-upd-irrel convert-lits-l-drop
mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev
)
done
done

lemma *cdcl-twl-subresolution-l-list-invs*:
‹*cdcl-twl-subresolution-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
using *distinct-mset-dom*[*of* ‹*get-clauses-l S*›] **apply** −
by (*induction rule*: *cdcl-twl-subresolution-l.induct*)
(*auto simp*: *twl-list-invs-def cdcl*$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*
ran-m-mapsto-upd ran-m-def add-mset-eq-add-mset tautology-add-mset
dest: *in-diffD dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-True-l-list-invs*:
‹*cdcl-twl-unitres-true-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
by (*induction rule*: *cdcl-twl-unitres-true-l.induct*)
(*auto simp*: *twl-list-invs-def cdcl*$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*
dest: *in-diffD*)

inductive *cdcl-twl-unitres-l* :: ‹*′v twl-st-l* ⇒ *′v twl-st-l* ⇒ *bool*› **where**
‹*cdcl-twl-unitres-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmupd D* (*E*, *irred N D*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *D*)) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
if
‹*D* ∈# *dom-m N*›
‹*count-decided M* = *0*› **and**
‹*mset* (*N* ∝ *D*) = *C* +*C′*›
‹(*mset* ‘# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C′*)›
‹¬*tautology C*› ‹*distinct-mset C*›
‹*struct-wf-twl-cls* (*twl-clause-of E*)›
‹*Multiset.Ball* (*mset E*) (*undefined-lit M*)›
‹*mset E* = *C*›
‹*D* ∉ *set* (*get-all-mark-of-propagated M*)› ‹*irred N D*› |
‹*cdcl-twl-unitres-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K 0* # *M*, *fmdrop D N*, *None*, *NE*, *UE*, *add-mset* {#*K*#} *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *D*)) *NS*, *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
if
‹*D* ∈# *dom-m N*›
‹*count-decided M* = *0*› **and**
‹*mset* (*N* ∝ *D*) = {#*K*#} +*C′*›
‹(*mset* ‘# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C′*)›

‹$D \notin$ *set* (*get-all-mark-of-propagated M*)› ‹*irred N D*›
‹*undefined-lit M K*› |
‹*cdcl-twl-unitres-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmupd D* (*E*, *irred N D*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *D*)) *US*, *N0*, *U0*, {#}, *Q*)›
if
‹*D* ∈# *dom-m N*›
‹*count-decided M* = *0*› **and**
‹*mset* (*N* ∝ *D*) = *C* +*C'*›
‹(*mset* '# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C'*)›
‹¬*tautology C*› ‹*distinct-mset C*›
‹*struct-wf-twl-cls* (*twl-clause-of E*)›
‹*Multiset.Ball* (*mset E*) (*undefined-lit M*)›
‹*mset E* = *C*›
‹*D* ∉ *set* (*get-all-mark-of-propagated M*)› ‹¬*irred N D*›
‹*atms-of* (*mset E*) ⊆ *atms-of-mm* (*mset* '# *init-clss-lf N*) ∪ *atms-of-mm NE* ∪ *atms-of-mm NEk* ∪ *atms-of-mm NS* ∪ *atms-of-mm N0*› |
‹*cdcl-twl-unitres-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K 0* # *M*, *fmdrop D N*, *None*, *NE*, *UE*, *NEk*, *add-mset* {#*K*#} *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *D*)) *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
if
‹*D* ∈# *dom-m N*›
‹*count-decided M* = *0*› **and**
‹*mset* (*N* ∝ *D*) = {#*K*#} +*C'*›
‹(*mset* '# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*) ⊨*psm mset-set* (*CNot C'*)›
‹*D* ∉ *set* (*get-all-mark-of-propagated M*)› ‹¬*irred N D*›
‹*undefined-lit M K*›
‹*atm-of K* ∈ *atms-of-mm* (*mset* '# *init-clss-lf N*) ∪ *atms-of-mm NE* ∪ *atms-of-mm NEk* ∪ *atms-of-mm NS* ∪ *atms-of-mm N0*› |
‹*cdcl-twl-unitres-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop D N*, *Some* {#}, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *D*)) *US*, *N0*, *add-mset* {#} *U0*, {#}, {#})›
if
‹*D* ∈# *dom-m N*›
‹(*mset* '# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*) ⊨*psm mset-set* (*CNot* (*mset* (*N* ∝ *D*)))›
‹¬*irred N D*›
‹*count-decided M* = *0*›
‹*D* ∉ *set* (*get-all-mark-of-propagated M*)› |
‹*cdcl-twl-unitres-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop D N*, *Some* {#}, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *D*)) *NS*, *US*, *add-mset* {#} *N0*, *U0*, {#}, {#})›
if
‹*D* ∈# *dom-m N*›
‹(*mset* '# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*) ⊨*psm mset-set* (*CNot* (*mset* (*N* ∝ *D*)))›
‹*irred N D*›
‹*count-decided M* = *0*›
‹*D* ∉ *set* (*get-all-mark-of-propagated M*)›

lemma *cdcl-twl-unitres-I1*:
‹*cdcl-twl-unitres S T*›
if ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*T* = (*M*, *add-mset E* (*remove1-mset D N*), *U*, *None*, *NE*, *UE*, *add-mset* (*clause D*) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
‹*count-decided M* = *0*› **and**
‹*D* ∈# *N*›
‹*clause D* = *C*+*C'*›

‹(clauses N + NE + NS + N0) ⊨psm mset-set (CNot C')›
‹¬tautology C› ‹distinct-mset C›
‹struct-wf-twl-cls E›
‹Multiset.Ball (clause E) (undefined-lit M)›
‹clause E = C›
using *that cdcl-twl-unitres.intros(1)[of M D C C' ‹N − {#D#}› NE NS N0 E U UE US U0 Q]*
by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-I2*:
‹cdcl-twl-unitres S T›
if ‹S = (M, N, U, None, NE, UE, NS, US, N0, U0, {#}, Q)›
‹T = (Propagated K {#K#} # M, (remove1-mset D N), U, None, add-mset {#K#} NE, UE, add-mset (clause D) NS, US, N0, U0, {#}, add-mset (−K) Q)›
‹count-decided M = 0› **and**
‹D ∈# N›
‹clause D = {#K#}+C'›
‹(clauses N + NE + NS + N0) ⊨psm mset-set (CNot C')›
‹undefined-lit M K›
using *that cdcl-twl-unitres.intros(2)[of M D ‹{#K#}› C' ‹N − {#D#}› NE NS N0 K U UE US U0 Q]*
by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-I3*:
‹cdcl-twl-unitres S T›
if ‹S = (M, N, U, None, NE, UE, NS, US, N0, U0, {#}, Q)›
‹T = (M, N, add-mset E (remove1-mset D U), None, NE, UE, NS, add-mset (clause D) US, N0, U0, {#}, Q)›
‹count-decided M = 0› **and**
‹D ∈# U›
‹clause D = C+C'›
‹(clauses N + NE + NS + N0) ⊨psm mset-set (CNot C')›
‹¬tautology C› ‹distinct-mset C›
‹struct-wf-twl-cls E›
‹Multiset.Ball (clause E) (undefined-lit M)›
‹clause E = C›
‹atms-of C ⊆ atms-of-ms (clause ‘ set-mset N) ∪ atms-of-mm NE ∪ atms-of-mm NS ∪ atms-of-mm N0›
using *that cdcl-twl-unitres.intros(3)[of M D C C' N NE NS N0 E ‹U − {#D#}› UE US U0 Q]*
by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-I4*:
‹cdcl-twl-unitres S T›
if ‹S = (M, N, U, None, NE, UE, NS, US, N0, U0, {#}, Q)›
‹T = (Propagated K {#K#} # M, N, remove1-mset D U, None, NE, add-mset {#K#} UE, NS, add-mset (clause D) US, N0, U0, {#}, add-mset (−K) Q)›
‹count-decided M = 0› **and**
‹D ∈# U›
‹clause D = {#K#}+C'›
‹(clauses N + NE + NS + N0) ⊨psm mset-set (CNot C')›
‹undefined-lit M K›
‹atm-of K ∈ atms-of-ms (clause ‘ set-mset N) ∪ atms-of-mm NE ∪ atms-of-mm NS ∪ atms-of-mm N0›
using *that cdcl-twl-unitres.intros(4)[of M D ‹{#K#}› C' N NE NS N0 K ‹U − {#D#}› UE US U0 Q]*
by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-I5*:
  ‹*cdcl-twl-unitres S T*›
  if ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
    ‹*T* = (*M*, *N*, *remove1-mset D U*, *Some* {#}, *NE*, *UE*, *NS*, *add-mset* (*clause D*) *US*, *N0*, *add-mset* {#} *U0*, {#}, {#})›
    ‹*count-decided M* = *0*› **and**
    ‹*D* ∈# *U*›
    ‹(*clauses N* + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot* (*clause D*))›
  using *that cdcl-twl-unitres.intros*(*5*)[*of M N NE NS N0 D* ‹*remove1-mset D U*› *UE US U0 Q*]
  by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-I6*:
  ‹*cdcl-twl-unitres S T*›
  if ‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
    ‹*T* = (*M*, *remove1-mset D N*, *U*, *Some* {#}, *NE*, *UE*, *add-mset* (*clause D*) *NS*, *US*, *add-mset* {#} *N0*, *U0*, {#}, {#})›
    ‹*count-decided M* = *0*› **and**
    ‹*D* ∈# *N*›
    ‹(*clauses* (*add-mset D N*) + *NE* + *NS* + *N0*) ⊨*psm mset-set* (*CNot* (*clause D*))›
  using *that cdcl-twl-unitres.intros*(*6*)[*of M D* ‹*remove1-mset D N*› *NE NS N0 U UE US U0 Q*]
  by (*auto dest*!: *multi-member-split*)

lemma *cdcl-twl-unitres-l-cdcl-twl-unitres*:
  assumes ‹*cdcl-twl-unitres-l S T*› **and**
    SS': ‹(*S*, *S'*) ∈ *twl-st-l None*›
  shows ‹∃ *T'*. (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-unitres S' T'*›
  using *assms*
  apply (*induction rule*: *cdcl-twl-unitres-l.induct*)
  subgoal for *D N M C C' NE NEk NS N0 E UE UEk US U0 Q*
    apply (*auto simp*: *twl-st-l-def*)[]
    apply (*rule-tac x*=*x* **in** *exI*)
    apply (*auto simp*: *twl-st-l-def*
      intro!: *cdcl-twl-unitres-I1*[**where** *E*= ‹*twl-clause-of E*› **and** *D* = ‹*twl-clause-of* (*N* ∝ *D*)›]
      simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
      learned-clss-l-mapsto-upd-irrel
      mset-take-mset-drop-mset'
      init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
      init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev Un-assoc
      learned-clss-l-mapsto-upd-irrel convert-lits-l-drop
      mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
      init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop convert-lits-l-update-sel2
      init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
      learned-clss-l-fmupd-if learned-clss-l-mapsto-upd
      init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev)[]
    done
  subgoal for *D N M K C' NE NS N0 UE US U0 Q*
    apply (*auto simp*: *twl-st-l-def*)[]
    apply (*rule-tac x*= ‹*Propagated K* {#*K*#} # *x*› **in** *exI*)
    apply (*auto simp*: *twl-st-l-def*
      intro!: *cdcl-twl-unitres-I2*[**where** *K*=*K* **and** *D* = ‹*twl-clause-of* (*N* ∝ *D*)›]
      simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
      learned-clss-l-mapsto-upd-irrel
      mset-take-mset-drop-mset'
      init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
      init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev Un-assoc
      learned-clss-l-mapsto-upd-irrel convert-lits-l-drop

mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop convert-lits-l-update-sel2
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel convert-lits-l-add-mset
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd convert-lit.intros(*3*)
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev)[]
done
subgoal for *D N M C C′ NE NEk NS N0 E UE UEk US U0 Q*
apply (*auto simp*: *twl-st-l-def*)[]
apply (*rule-tac x*=*x* **in** *exI*)
apply (*auto 5 3 simp*: *twl-st-l-def*
intro!: *cdcl-twl-unitres-I3*[**where** *E*= ‹*twl-clause-of E*› **and** *D* = ‹*twl-clause-of* (*N* ∝ *D*)›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev Un-assoc
learned-clss-l-mapsto-upd-irrel convert-lits-l-drop
mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop convert-lits-l-update-sel2
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd image-image
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev)[]
done
subgoal for *D N M K C′ NE NS N0 UE US U0 Q*
apply (*auto simp*: *twl-st-l-def*; *rule-tac x*= ‹*Propagated K* {#*K*#} # *x*› **in** *exI*)
apply (*auto simp*: *twl-st-l-def*
intro!: *cdcl-twl-unitres-I4*[**where** *K*=*K* **and** *D* = ‹*twl-clause-of* (*N* ∝ *D*)›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev image-image
learned-clss-l-mapsto-upd-irrel convert-lits-l-drop Un-assoc
mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop convert-lits-l-update-sel2
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel convert-lits-l-add-mset
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd convert-lit.intros(*3*)
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev)
done
subgoal for *D N NE NS N0 M UE US U0 Q*
by (*auto simp*: *twl-st-l-def*; *rule-tac x*= ‹*x*› **in** *exI*)
(*auto simp*: *twl-st-l-def*
intro!: *cdcl-twl-unitres-I5*[**where** *D* = ‹*twl-clause-of* (*N* ∝ *D*)›]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset′
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev image-image
learned-clss-l-mapsto-upd-irrel convert-lits-l-drop Un-assoc
mset-take-mset-drop-mset′ init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop convert-lits-l-update-sel2
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel convert-lits-l-add-mset
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd convert-lit.intros(*3*)
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev)
subgoal for *D N NE NS N0 M UE US U0 Q*
by (*auto simp*: *twl-st-l-def*; *rule-tac x*= ‹*x*› **in** *exI*)

(*auto simp*: *twl-st-l-def*
intro!: *cdcl-twl-unitres-I6*[**where** $D = \langle$*twl-clause-of* $(N \propto D)\rangle$]
simp: *init-clss-l-mapsto-upd image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel
mset-take-mset-drop-mset'
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev image-image
learned-clss-l-mapsto-upd-irrel convert-lits-l-drop Un-assoc
mset-take-mset-drop-mset' init-clss-l-mapsto-upd-irrelev
init-clss-l-fmdrop-irrelev learned-clss-l-l-fmdrop convert-lits-l-update-sel2
init-clss-l-mapsto-upd-irrel-notin init-clss-l-mapsto-upd-irrel convert-lits-l-add-mset
learned-clss-l-fmupd-if learned-clss-l-mapsto-upd convert-lit.intros(*3*)
init-clss-l-fmdrop learned-clss-l-l-fmdrop-irrelev)
done

definition *simplify-clause-with-unit* :: ‹*nat* ⇒ (*'v*, *nat*) *ann-lits* ⇒ *'v clauses-l* ⇒ (*bool* × *bool* × *'v clauses-l*) *nres*› **where**
‹*simplify-clause-with-unit* = (λ*C M N*. *do* {
SPEC(λ(*unc*, *b*, *N'*). *fmdrop C N* = *fmdrop C N'* ∧ *mset* (*N'* ∝ *C*) ⊆# *mset* (*N* ∝ *C*) ∧ *C* ∈# *dom-m N'* ∧
(¬*b* ⟶ (∀ *L* ∈# *mset* (*N'* ∝ *C*). *undefined-lit M L*)) ∧
(∀ *L* ∈# *mset* (*N* ∝ *C*) − *mset* (*N'* ∝ *C*). *defined-lit M L*) ∧
(*irred N C* = *irred N' C*) ∧
(*b* ⟷ (∃ *L*. *L* ∈# *mset* (*N* ∝ *C*) ∧ *L* ∈ *lits-of-l M*)) ∧
(*unc* ⟶ (*N* = *N'* ∧ ¬*b*)))
})›

definition *simplify-clause-with-unit-st-pre* :: ‹*nat* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*simplify-clause-with-unit-st-pre* = (λ*C S*.
C ∈# *dom-m* (*get-clauses-l S*) ∧ *count-decided* (*get-trail-l S*) = *0* ∧ *get-conflict-l S* = *None* ∧ *clauses-to-update-l S* = {#} ∧
twl-list-invs S ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
(∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*)))
)›

definition *simplify-clause-with-unit-st* :: ‹*nat* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*simplify-clause-with-unit-st* = (λ*C* (*M*, *N*$_0$, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*simplify-clause-with-unit-st-pre C* (*M*, *N*$_0$, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
ASSERT (*C* ∈# *dom-m N*$_0$ ∧ *count-decided M* = *0* ∧ *D* = *None* ∧ *WS* = {#} ∧ *no-dup M* ∧ *C* ≠ *0*);
let S = (*M*, *N*$_0$, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*);
if False
then RETURN (*M*, *N*$_0$, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
else do {
let E = *mset* (*N*$_0$ ∝ *C*);
let irr = *irred N*$_0$ *C*;
(*unc*, *b*, *N*) ← *simplify-clause-with-unit C M N*$_0$;
ASSERT(*fmdrop C N* = *fmdrop C N*$_0$ ∧ *irred N C* = *irred N*$_0$ *C* ∧ *mset* (*N* ∝ *C*) ⊆# *mset* (*N*$_0$ ∝ *C*) ∧
C ∈# *dom-m N*);
if unc then do {
ASSERT (*N* = *N*$_0$);
RETURN (*M*, *N*$_0$, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)

```
    }
    else if b then do {
      let T = (M, fmdrop C N, D, (if irr then add-mset E else id) NE, (if ¬irr then add-mset E else
id) UE, NEk, UEk, NS, US, N0, U0, WS, Q);
        ASSERT (set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S));
        ASSERT (set-mset (all-learned-lits-of-l T) = set-mset (all-learned-lits-of-l S));
      RETURN T
    }
    else if size (N ∝ C) = 1
    then do {
      let L = ((N ∝ C) ! 0);
      let T = (Propagated L 0 # M, fmdrop C N, D, NE, UE, (if irr then add-mset {#L#} else id)
NEk,
          (if ¬irr then add-mset {#L#} else id)UEk, (if irr then add-mset E else id) NS,
          (if ¬irr then add-mset E else id)US, N0, U0, WS, add-mset (−L) Q);
        ASSERT (set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S));
        ASSERT (set-mset (all-learned-lits-of-l T) = set-mset (all-learned-lits-of-l S));
        ASSERT (undefined-lit M L ∧ L ∈# all-init-lits-of-l S);
      RETURN T}
    else if size (N ∝ C) = 0
    then do {
        let T = (M, fmdrop C N, Some {#}, NE, UE, NEk, UEk, (if irr then add-mset E else id)
NS, (if ¬irr then add-mset E else id) US, (if irr then add-mset {#} else id) N0, (if ¬irr then add-mset
{#} else id)U0, WS, {#});
        ASSERT (set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S));
        ASSERT (set-mset (all-learned-lits-of-l T) = set-mset (all-learned-lits-of-l S));
        RETURN T
    }
    else do {
        let T = (M, N, D, NE, UE, NEk, UEk, (if irr then add-mset E else id) NS, (if ¬irr then
add-mset E else id) US, N0, U0, WS, Q);
        ASSERT (set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S));
        ASSERT (set-mset (all-learned-lits-of-l T) = set-mset (all-learned-lits-of-l S));
        RETURN T
    }
  }
 })›

lemma true-clss-clss-def-more-atms:
 ‹N ⊨ps N' ⟷ (∀ I. total-over-m I (N ∪ N' ∪ N'') ⟶ consistent-interp I ⟶ I ⊨s N ⟶ I ⊨s
N')›
 (is ‹?A ⟷ ?B›)
proof (rule iffI)
 assume ?A
 then show ?B
  by (simp add: true-clss-clss-def)
next
 assume ?B
 show ?A
  unfolding true-clss-clss-def
 proof (intro allI impI)
  fix I
  assume tot: ‹total-over-m I (N ∪ N')› and
    cons: ‹consistent-interp I› and
    IN: ‹I ⊨s N›
```

let *?I* = ‹*I* ∪ *Pos* ‘ {*A*. *A* ∈ *atms-of-ms N''* ∧ *A* ∉ *atm-of* ‘ *I*}›
have ‹*total-over-m ?I* (*N* ∪ *N'* ∪ *N''*)›
using *tot atms-of-s-def* **by** (*fastforce simp*: *total-over-m-def total-over-set-def*)
moreover have ‹*consistent-interp ?I*›
using *cons* **unfolding** *consistent-interp-def*
by (*force simp*: *uminus-lit-swap*)
moreover have ‹*?I* ⊨*s N*›
using *IN* **by** *auto*
ultimately have ‹*?I* ⊨*s N'*›
using ‹*?B*› **by** *blast*
then show ‹*I* ⊨*s N'*›
by (*smt atms-of-ms-mono atms-of-s-def imageE literal.sel*(1) *mem-Collect-eq*
notin-vars-union-true-clss-true-clss subsetD sup-ge2 tot total-over-m-alt-def)
qed
qed

lemma *true-clss-clss-def-iff-negation-in-model*:
‹*A* ⊨*ps CNot C'* ⟷ (∀ *L* ∈# *C'*. *A* ⊨*ps* {{#−*L*#}})›
apply (*rule iffI*)
subgoal
by (*simp add*: *true-clss-clss-in-imp-true-clss-cls*)
subgoal
apply (*subst* (*asm*)*true-clss-clss-def-more-atms*[*of* - - ‹{*C'*}›])
apply (*auto simp*: *true-clss-clss-def true-clss-def-iff-negation-in-model*
dest!: *multi-member-split*)
done
done

lemma
fixes *M* :: ‹('*v*, *nat*) *ann-lit list*› **and** *N NE UE NS US N0 U0 Q NEk UEk*
defines ‹*S* ≡ (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
assumes
‹*D* ∈# *dom-m N*›
‹*count-decided M* = *0*› **and**
‹*D* ∉ *set* (*get-all-mark-of-propagated M*)› **and**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
st-invs: ‹*twl-struct-invs T*› **and**
dec: ‹*count-decided* (*get-trail-l S*) = *0*› **and**
false: ‹∀ *L* ∈# *C'*. ¬*undefined-lit* (*get-trail-l S*) *L*›
‹∀ *L* ∈# *C'*. *L* ∉ *lits-of-l* (*get-trail-l S*)› **and**
ent: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*)›
shows *cdcl-twl-unitres-l-intros2'*:
‹*irred N D* ⟹ *undefined-lit M K* ⟹ *mset* (*N* ∝ *D*) = {#*K*#} +*C'* ⟹
cdcl-twl-unitres-l (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K 0* # *M*, *fmdrop D N*, *None*, *NE*, *UE*, *add-mset* {#*K*#} *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *D*)) *NS*, *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
(**is** ‹*?A* ⟹ - ⟹ - ⟹ *?B*›) **and**
cdcl-twl-unitres-l-intros4':
‹¬*irred N D* ⟹ *undefined-lit M K* ⟹ *mset* (*N* ∝ *D*) = {#*K*#} +*C'* ⟹
cdcl-twl-unitres-l (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated K 0* # *M*, *fmdrop D N*, *None*, *NE*, *UE*, *NEk*, *add-mset* {#*K*#} *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *D*)) *US*, *N0*, *U0*, {#}, *add-mset* (−*K*) *Q*)›
(**is** ‹*?A'* ⟹ - ⟹ - ⟹ *?B'*›) **and**
cdcl-twl-unitres-false-entailed:
‹(*mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses-l S* + *get-subsumed-init-clauses-l S*

+ *get-init-clauses0-l S*) ⊨*psm mset-set* (*CNot C′*)› **and**
cdcl-twl-unitres-l-intros5′:
‹¬*irred N D* ⟹ *mset* (*N* ∝ *D*) = *C′* ⟹
cdcl-twl-unitres-l (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop D N*, *Some* {#}, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *D*)) *US*, *N0*,
add-mset {#} *U0*, {#}, {#})›
(**is** ‹*?E* ⟹ - ⟹ *?F*›) **and**
cdcl-twl-unitres-l-intros6′:
‹*irred N D* ⟹ *mset* (*N* ∝ *D*) = *C′* ⟹
cdcl-twl-unitres-l (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmdrop D N*, *Some* {#}, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *D*)) *NS*, *US*, *add-mset* {#} *N0*,
U0, {#}, {#})›
(**is** ‹*?E′* ⟹ - ⟹ *?F′*›) **and**
cdcl-twl-unitres-l-intros1′:
‹*irred N D* ⟹ *mset* (*N* ∝ *D*) = *mset C* +*C′* ⟹ *length C* ≥ *2* ⟹ ¬*tautology* (*mset* (*N* ∝ *D*)) ⟹
∀ *L*∈*set C*. *undefined-lit M L* ⟹ *N′* = *fmupd D* (*C*, *irred N D*) *N* ⟹
cdcl-twl-unitres-l (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N′*, *None*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *D*)) *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
(**is** ‹*?G* ⟹ - ⟹ - ⟹ - ⟹ - ⟹ - ⟹ *?H*›) **and**
cdcl-twl-unitres-l-intros3′:
‹¬*irred N D* ⟹ *mset* (*N* ∝ *D*) = *mset C* +*C′* ⟹ *length C* ≥ *2* ⟹ ¬*tautology* (*mset* (*N* ∝ *D*)) ⟹
∀ *L*∈*set C*. *undefined-lit M L* ⟹ *N′* = *fmupd D* (*C*, *irred N D*) *N* ⟹
cdcl-twl-unitres-l (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N′*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *D*)) *US*, *N0*, *U0*, {#}, *Q*)›
(**is** ‹*?G′* ⟹ - ⟹ - ⟹ - ⟹ - ⟹ - ⟹ *?H′*›)
proof −
have ‹*all-decomposition-implies-m* (*cdcl*$_W$*-restart-mset.clauses* ((*state*$_W$*-of T*)))
(*get-all-ann-decomposition* (*trail* ((*state*$_W$*-of T*)))›) **and**
alien: ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of T*)› **and**
dist: ‹*cdcl*$_W$*-restart-mset.distinct-cdcl*$_W$*-state* (*state*$_W$*-of T*)› **and**
tauto: ‹∀ *s*∈#*learned-clss* (*state*$_W$*-of T*). ¬ *tautology s*› **and**
nd: ‹*no-dup* (*trail* (*state-of* (*pstate*$_W$*-of T*)))›
using *st-invs* **unfolding** *twl-struct-invs-def* **unfolding** *pcdcl-all-struct-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def state*$_W$*-of-def pcdcl-all-struct-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by *fast*+
moreover have *H*: ‹(λ*x*. *mset* (*fst x*)) ‘ {*a*. *a* ∈# *ran-m b* ∧ *snd a*} ∪ (*set-mset d* ∪ *set-mset d′*) ∪ *f* ∪ *h* ∪ *U*⊨*ps*
unmark-l aa ⟹
(*a*, *aa*) ∈ *convert-lits-l b* (*d′* + *e*) ⟹
(λ*x*. *mset* (*fst x*)) ‘ {*a*. *a* ∈# *ran-m b* ∧ *snd a*} ∪ (*set-mset d* ∪ *set-mset d′*) ∪ *f* ∪ *h* ⊨*ps U* ⟹
− *L* ∈ *lits-of-l a* ⟹
(λ*x*. *mset* (*fst x*)) ‘ {*a*. *a* ∈# *ran-m b* ∧ *snd a*} ∪ (*set-mset d* ∪ *set-mset d′*) ∪ *f* ∪ *h* ⊨*p* {#− *L*#}›
for *aa* :: ‹('*v*, '*v clause*) *ann-lits*› **and** *a* :: ‹('*v*, *nat*) *ann-lit list*› **and** *b d e f L U h d′*
by (*smt in-unmark-l-in-lits-of-l-iff list-of-l-convert-lits-l*
true-clss-clss-in-imp-true-clss-cls true-clss-clss-left-right
true-clss-clss-union-and)
moreover have *false*: ‹∀ *L* ∈# *C′*. −*L* ∈ *lits-of-l* (*get-trail-l S*)›
using *false nd* **by** (*auto dest*!: *multi-member-split*
simp: *Decided-Propagated-in-iff-in-lits-of-l*)
ultimately show *ent2*: ‹(*mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses-l S* + *get-subsumed-init-clauses-l S* + *get-init-clauses0-l S*) ⊨*psm mset-set* (*CNot C′*)›
using *dec ST false ent*

by (*cases S*; *cases T*)
(*auto simp*: *twl-st-l-def all-decomposition-implies-def clauses-def*
get-all-ann-decomposition-count-decided0 image-image mset-take-mset-drop-mset′
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init-def true-clss-clss-def-iff-negation-in-model
dest!: *multi-member-split*)
moreover {
have ‹*N* ∝ *D* ∈# *init-clss-lf N* ∨ *N* ∝ *D* ∈# *learned-clss-lf N*›
using *assms*(*1*,*2*) **by** (*auto simp*: *ran-m-def*)
then have ‹*atms-of* (*mset* (*N* ∝ *D*)) ⊆ *atms-of-mm* (*mset* ‘# *init-clss-lf N*) ∪ *atms-of-mm NE* ∪ *atms-of-mm NEk* ∪ *atms-of-mm NS* ∪ *atms-of-mm N0*›
using *alien assms*(*1*) *ST*
by (*fastforce simp*: *twl-st-l-def clauses-def mset-take-mset-drop-mset′ image-image*
cdcl$_W$-restart-mset.no-strange-atm-def conj-disj-distribR Collect-disj-eq Collect-conv-if
dest!: *multi-member-split*[*of* - ‹*ran-m N*›])
} note *H* = *this*
ultimately show ‹*?A* ⟹ *?B*› ‹*?A′* ⟹ *undefined-lit M K* ⟹ *?B′*›
if ‹*mset* (*N* ∝ *D*) = {#*K*#} + *C′*› ‹*undefined-lit M K*›
using *assms cdcl-twl-unitres-l.intros*(*2*)[*of D N M K C′ NE NEk NS N0 UE UEk US U0 Q*]
cdcl-twl-unitres-l.intros(*4*)[*of D N M K C′ NE NEk NS N0 UE UEk US U0 Q*] *that*
by (*auto simp*: *Un-assoc*)

show *?F* **if** ‹*?E*› ‹*mset* (*N* ∝ *D*) = *C′*›
proof −
have ‹*mset* ‘# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0* ⊨*psm mset-set* (*CNot* (*mset* (*N* ∝ *D*)))›
using *ent2 that assms* **by** (*auto simp*: *Un-assoc*)
then show *?thesis*
using *cdcl-twl-unitres-l.intros*(*5*)[*of D N NE NEk NS N0 M UE UEk US U0 Q*] *assms that*
by (*auto simp*: *Un-assoc*)
qed

show *?F′* **if** ‹*?E′*› ‹*mset* (*N* ∝ *D*) = *C′*›
proof −
have ‹*mset* ‘# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0* ⊨*psm mset-set* (*CNot* (*mset* (*N* ∝ *D*)))›
using *ent2 that assms* **by** (*auto simp*: *Un-assoc*)
then show *?thesis*
using *cdcl-twl-unitres-l.intros*(*6*)[*of D N NE NEk NS N0 M UE UEk US U0 Q*] *assms that*
by *auto*
qed
show *?H* **if** *?G* ‹*mset* (*N* ∝ *D*) = *mset C* + *C′*› ‹*length C* ≥ *2*› ‹∀ *L*∈*set C*. *undefined-lit M L*›
‹¬*tautology* (*mset* (*N* ∝ *D*))› ‹*N′* = *fmupd D* (*C*, *irred N D*) *N*›
proof −
have *dist-C*: ‹*distinct-mset* (*mset* (*N* ∝ *D*))›
using *that dist ST assms*(*2*)
by (*auto simp*: *twl-st-l-def cdcl$_W$-restart-mset.distinct-cdcl$_W$-state-def ran-m-def*
S-def mset-take-mset-drop-mset′
dest!: *multi-member-split*)
moreover have ‹*struct-wf-twl-cls* (*twl-clause-of C*)› ‹*distinct C*›
using *that distinct-mset-mono*[*of* ‹*mset C*› ‹*mset* (*N* ∝ *D*)›] *dist-C*
by (*auto simp*: *twl-st-l-def cdcl$_W$-restart-mset.distinct-cdcl$_W$-state-def*
mset-take-mset-drop-mset′)
moreover have ‹¬*tautology* (*mset C*)›
using *not-tautology-mono*[*of* ‹*mset C*› ‹*mset C* + *C′*›] *that assms* **by** *auto*
ultimately show *?thesis*
using *cdcl-twl-unitres-l.intros*(*1*)[*of D N M* ‹*mset C*› *C′ NE NEk NS N0 C UE UEk US U0*] *assms that ent2*
by (*auto simp*: *Un-assoc*)

```
  qed
  show ?H' if ?G' ‹mset (N ∝ D) = mset C + C'› ‹length C ≥ 2› ‹∀ L∈set C. undefined-lit M L›
    ‹¬tautology (mset (N ∝ D))› ‹N' = fmupd D (C, irred N D) N›
  proof −
    have dist-C: ‹distinct-mset (mset (N ∝ D))›
      using that dist ST assms(2)
      by (auto simp: twl-st-l-def cdcl_W-restart-mset.distinct-cdcl_W-state-def ran-m-def
          S-def mset-take-mset-drop-mset'
        dest!: multi-member-split)
    moreover have ‹struct-wf-twl-cls (twl-clause-of C)› ‹distinct C›
      using that distinct-mset-mono[of ‹mset C› ‹mset (N ∝ D)›] dist-C
      by (auto simp: twl-st-l-def cdcl_W-restart-mset.distinct-cdcl_W-state-def
        mset-take-mset-drop-mset')
    moreover have ‹¬tautology (mset C)›
      using not-tautology-mono[of ‹mset C› ‹mset C + C'›] that assms by auto
    ultimately show ?thesis
     using cdcl-twl-unitres-l.intros(3)[of D N M ‹mset C› C' NE NEk NS N0 C UE UEk US U0] assms
that ent2 H
      by (auto simp: Un-assoc)
  qed
qed


lemma fmdrop-eq-update-eq: ‹fmdrop C aa = fmdrop C bh ⟹ C ∈# dom-m bh ⟹
  bh = fmupd C (bh ∝ C, irred bh C) aa›
  apply (subst (asm) fmlookup-inject[symmetric])
  apply (subst fmlookup-inject[symmetric])
  apply (intro ext)
  apply auto[]
  by (metis fmlookup-drop)

lemma fmdrop-eq-update-eq2: ‹fmdrop C aa = fmdrop C bh ⟹ C ∈# dom-m bh ⟹ b = irred bh C
⟹
  bh = fmupd C (bh ∝ C, b) aa›
  using fmdrop-eq-update-eq by fast

lemma twl-st-l-struct-invs-distinct:
  assumes
    ST: ‹(S, T) ∈ twl-st-l b› and
    C: ‹C ∈# dom-m (get-clauses-l S)› and
    invs: ‹twl-struct-invs T›
  shows ‹distinct (get-clauses-l S ∝ C)›
proof −
  have ‹(∀ C ∈# get-clauses T. struct-wf-twl-cls C)›
    using invs unfolding twl-struct-invs-def by (cases T) (auto simp: twl-st-inv.simps)
  moreover have ‹twl-clause-of (get-clauses-l S ∝ C) ∈# (get-clauses T)›
    using ST C by (cases S; cases T) (auto simp: twl-st-l-def)
  ultimately show ?thesis
    by (auto dest!: multi-member-split simp: mset-take-mset-drop-mset')
qed

lemma list-length-2-isabelle-come-on:
  ‹length C ≠ Suc 0 ⟹ C ≠ [] ⟹ length C ≥ 2›
  by (cases C; cases ‹tl C›) auto

lemma in-all-lits-of-mm-init-clss-l-single-out:
```

‹xa ∈# all-lits-of-m (mset (aa ∝ C)) ⟹
C ∈# dom-m aa ⟹ irred aa C ⟹ xa ∈# all-lits-of-mm {#mset (fst x). x ∈# init-clss-l aa#}›
and
in-all-lits-of-mm-learned-clss-l-single-out:
‹xa ∈# all-lits-of-m (mset (aa ∝ C)) ⟹
C ∈# dom-m aa ⟹ ¬irred aa C ⟹ xa ∈# all-lits-of-mm {#mset (fst x). x ∈# learned-clss-l aa#}›
by (auto simp: ran-m-def all-lits-of-mm-add-mset dest!: multi-member-split)

lemma pget-all-learned-clss-get-all-learned-clss-l:
‹(S, x) ∈ twl-st-l b ⟹
pget-all-learned-clss (pstate$_W$-of x) = get-all-learned-clss-l S›
by (cases x; cases S)
(auto simp: get-learned-clss-l-def twl-st-l-def mset-take-mset-drop-mset′)
lemma pget-all-init-clss-get-all-init-clss-l:
‹(S, x) ∈ twl-st-l b ⟹
pget-all-init-clss (pstate$_W$-of x) = get-all-init-clss-l S›
by (cases x; cases S)
(auto simp: get-init-clss-l-def twl-st-l-def mset-take-mset-drop-mset′)

lemma simplify-clause-with-unit-st-spec:
assumes ‹simplify-clause-with-unit-st-pre C S›
shows ‹simplify-clause-with-unit-st C S ≤ ⇓Id (SPEC(λT.
(S = T ∨ cdcl-twl-unitres-l S T ∨ cdcl-twl-unitres-true-l S T) ∧
(set (get-all-mark-of-propagated (get-trail-l T)) ⊆
set (get-all-mark-of-propagated (get-trail-l S)) ∪ {0}) ∧
(dom-m (get-clauses-l T) = dom-m (get-clauses-l S) ∨
dom-m (get-clauses-l T) = remove1-mset C (dom-m (get-clauses-l S))) ∧
(∀C′ ∈# dom-m (get-clauses-l T). C ≠ C′ ⟶ fmlookup (get-clauses-l S) C′ = fmlookup (get-clauses-l T) C′) ∧
(C ∈# dom-m (get-clauses-l T) ⟶ (∀L∈set (get-clauses-l T ∝ C). undefined-lit (get-trail-l T) L))))›
proof −
obtain T **where**
C: ‹C ∈# dom-m (get-clauses-l S)› ‹count-decided (get-trail-l S) = 0› **and**
confl: ‹get-conflict-l S = None› **and**
clss: ‹clauses-to-update-l S = {#}› **and**
ST: ‹(S, T) ∈ twl-st-l None› **and**
st-invs: ‹twl-struct-invs T› **and**
list-invs: ‹twl-list-invs S› **and**
ent: ‹cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init ((state$_W$-of T))› **and**
annot: ‹set (get-all-mark-of-propagated (get-trail-l S)) ⊆ {0}›
using assms **unfolding** simplify-clause-with-unit-st-pre-def
by auto
have C0: ‹C ≠ 0› ‹C ∉ set (get-all-mark-of-propagated (get-trail-l S))›
using list-invs C annot
by (auto simp: twl-list-invs-def)
have add-new: ‹L ∈# all-init-lits-of-l S ⟷ L ∈# all-init-lits-of-l S + all-learned-lits-of-l S›
if ‹simplify-clause-with-unit-st-pre C S› **for** L S
using that **unfolding** simplify-clause-with-unit-st-pre-def twl-struct-invs-def
pcdcl-all-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def
cdcl$_W$-restart-mset.no-strange-atm-def state$_W$-of-def[symmetric] **apply** −
by normalize-goal+
(auto simp add: all-init-lits-of-l-def all-learned-lits-of-l-def
in-all-lits-of-mm-ain-atms-of-iff pget-all-learned-clss-get-all-learned-clss-l
pget-all-init-clss-get-all-init-clss-l image-Un get-learned-clss-l-def)

have [*dest*]: ‹*fmdrop C aa = fmdrop C bj* ⟹
remove1-mset C (*dom-m bj*) = *remove1-mset C* (*dom-m aa*)› **for** *C ab bj aa*
by (*metis dom-m-fmdrop*)
have *n-d*:‹*no-dup* (*get-trail-l S*)›
using *ST st-invs* **unfolding** *twl-struct-invs-def pcdcl-all-struct-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by *simp*
have *in-lits*:
‹*C* ∈# *dom-m aa* ∧ *count-decided a* = *0* ∧ *ab* = *None* ∧ *ci* = {#} ∧ *no-dup a* ∧ *C* ≠ *0*⟹
case x of
(*unc*, *b*, *N'*) ⇒
fmdrop C aa = *fmdrop C N'* ∧
mset (*N'* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧
C ∈# *dom-m N'* ∧
(¬ *b* ⟶ (∀ *L*∈#*mset* (*N'* ∝ *C*). *undefined-lit a L*)) ∧
Multiset.Ball (*mset* (*aa* ∝ *C*) − *mset* (*N'* ∝ *C*)) (*defined-lit a*) ∧
irred aa C = *irred N' C* ∧ *b* = (∃ *L*. *L* ∈# *mset* (*aa* ∝ *C*) ∧ *L* ∈ *lits-of-l a*) ∧
(*unc* ⟶ (*aa* = *N'* ∧ ¬*b*)) ⟹
fmdrop C bj = *fmdrop C aa* ∧ *irred bj C* = *irred aa C* ∧ *mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧ *C*
∈# *dom-m bj* ⟹
x = (*unc*, *bj'*) ⟹
bj' = (*aj*, *bj*) ⟹
¬*unc* ⟹ *aj* ⟹
set-mset
(*all-init-lits-of-l*
(*a*, *fmdrop C bj*, *ab*, (*if irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *ac*,
(*if* ¬ *irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *ad*, *ae*, *af*, *ag*, *ah*, *ai*, *bi*, *ci*, *di*)) =
set-mset (*all-init-lits-of-l* (*a*, *aa*, *ab*, *ac*, *ad*, *ae*, *af*, *ag*, *ah*, *ai*, *bi*, *ci*, *di*))
›
‹*C* ∈# *dom-m aa* ∧ *count-decided a* = *0* ∧ *ab* = *None* ∧ *ci* = {#} ∧ *no-dup a* ∧ *C* ≠ *0* ⟹
case x of
(*unc*, *b*, *N'*) ⇒
fmdrop C aa = *fmdrop C N'* ∧
mset (*N'* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧
C ∈# *dom-m N'* ∧
(¬ *b* ⟶ (∀ *L*∈#*mset* (*N'* ∝ *C*). *undefined-lit a L*)) ∧
Multiset.Ball (*mset* (*aa* ∝ *C*) − *mset* (*N'* ∝ *C*)) (*defined-lit a*) ∧
irred aa C = *irred N' C* ∧ *b* = (∃ *L*. *L* ∈# *mset* (*aa* ∝ *C*) ∧ *L* ∈ *lits-of-l a*) ∧
(*unc* ⟶ (*aa* = *N'* ∧ ¬*b*)) ⟹
fmdrop C bj = *fmdrop C aa* ∧ *irred bj C* = *irred aa C* ∧ *mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧ *C*
∈# *dom-m bj* ⟹
x = (*unc*, *bj'*) ⟹
bj' = (*aj*, *bj*) ⟹
¬*unc* ⟹ *aj* ⟹
set-mset
(*all-learned-lits-of-l*
(*a*, *fmdrop C bj*, *ab*, (*if irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *ac*,
(*if* ¬ *irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *ad*, *ae*, *af*, *ag*, *ah*, *ai*, *bi*, *ci*, *di*)) =
set-mset (*all-learned-lits-of-l* (*a*, *aa*, *ab*, *ac*, *ad*, *ae*, *af*, *ag*, *ah*, *ai*, *bi*, *ci*, *di*)) ›
for *a b aa ba ab bb ac bc ad bd ae be af bf ag bg ah bh ai bi x aj bj unc bj' ci di*
using *assms distinct-mset-dom*[*of bj*] *distinct-mset-dom*[*of aa*]
apply (*auto simp*: *all-init-lits-of-l-def get-init-clss-l-def ran-m-def*
all-lits-of-mm-add-mset all-lits-of-mm-union
all-learned-lits-of-l-def get-learned-clss-l-def
dest!: *multi-member-split*[*of* - ‹*dom-m* -›]

cong:)
apply (*smt* (*verit*, *best*) *image-mset-cong2*)+
done
have *in-lits-prop*:
‹*C* ∈# *dom-m aa* ∧ *count-decided a* = 0 ∧ *ab* = *None* ∧ *di* = {#} ∧ *no-dup a* ∧ *C* ≠ 0 ⟹
case x of
(*unc*, *b*, *N'*) ⇒
fmdrop C aa = *fmdrop C N'* ∧
mset (*N'* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧
C ∈# *dom-m N'* ∧
(¬ *b* ⟶ (∀ *L*∈#*mset* (*N'* ∝ *C*). *undefined-lit a L*)) ∧
Multiset.Ball (*mset* (*aa* ∝ *C*) − *mset* (*N'* ∝ *C*)) (*defined-lit a*) ∧
irred aa C = *irred N' C* ∧ *b* = (∃ *L*. *L* ∈# *mset* (*aa* ∝ *C*) ∧ *L* ∈ *lits-of-l a*) ∧
(*unc* ⟶ (*aa* = *N'*∧ ¬*b*)) ⟹
x = (*unc*, *bj'*) ⟹
bj' = (*aj*, *bj*) ⟹
¬*unc* ⟹ ¬*aj* ⟹
fmdrop C bj = *fmdrop C aa* ∧ *irred bj C* = *irred aa C* ∧ *mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧ *C* ∈# *dom-m bj* ⟹
length (*bj* ∝ *C*) = 1 ⟹
set-mset
(*all-init-lits-of-l*
(*Propagated* (*bj* ∝ *C* ! 0) 0 # *a*, *fmdrop C bj*, *ab*, *ah*, *ai*, (*if irred aa C then add-mset* {#*bj* ∝ *C* ! 0#} *else id*) *ac*,
(*if* ¬ *irred aa C then add-mset* {#*bj* ∝ *C* ! 0#} *else id*) *ad*, (*if irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *ae*,
(*if* ¬ *irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *af*, *ag*, *ci*, *di*, *add-mset* (− *bj* ∝ *C* ! 0) *bi*)) =
set-mset (*all-init-lits-of-l* (*a*, *aa*, *ab*, *ah*, *ai*, *ac*, *ad*, *ae*, *af*, *ag*, *ci*, *di*, *bi*))
›
‹*C* ∈# *dom-m aa* ∧ *count-decided a* = 0 ∧ *ab* = *None* ∧ *di* = {#} ∧ *no-dup a* ∧ *C* ≠ 0 ⟹
case x of
(*unc*, *b*, *N'*) ⇒
fmdrop C aa = *fmdrop C N'* ∧
mset (*N'* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧
C ∈# *dom-m N'* ∧
(¬ *b* ⟶ (∀ *L*∈#*mset* (*N'* ∝ *C*). *undefined-lit a L*)) ∧
Multiset.Ball (*mset* (*aa* ∝ *C*) − *mset* (*N'* ∝ *C*)) (*defined-lit a*) ∧
irred aa C = *irred N' C* ∧ *b* = (∃ *L*. *L* ∈# *mset* (*aa* ∝ *C*) ∧ *L* ∈ *lits-of-l a*) ∧
(*unc* ⟶ (*aa* = *N'*∧ ¬*b*)) ⟹
x = (*unc*, *bj'*) ⟹
bj' = (*aj*, *bj*) ⟹
¬*unc* ⟹ ¬*aj* ⟹
fmdrop C bj = *fmdrop C aa* ∧ *irred bj C* = *irred aa C* ∧ *mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ∧ *C* ∈# *dom-m bj* ⟹
length (*bj* ∝ *C*) = 1 ⟹
set-mset
(*all-learned-lits-of-l*
(*Propagated* (*bj* ∝ *C* ! 0) 0 # *a*, *fmdrop C bj*, *ab*, *ah*, *ai*, (*if irred aa C then add-mset* {#*bj* ∝ *C* ! 0#} *else id*) *ac*,
(*if* ¬ *irred aa C then add-mset* {#*bj* ∝ *C* ! 0#} *else id*) *ad*, (*if irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *ae*,
(*if* ¬ *irred aa C then add-mset* (*mset* (*aa* ∝ *C*)) *else id*) *af*, *ag*, *ci*, *di*, *add-mset* (− *bj* ∝ *C* ! 0) *bi*)) =
set-mset (*all-learned-lits-of-l* (*a*, *aa*, *ab*, *ah*, *ai*, *ac*, *ad*, *ae*, *af*, *ag*, *ci*, *di*, *bi*))
›

for *a b aa ba ab bb ac bc ad bd ae be af bf ag bg ah bh ai bi x aj bj bj' unc ci di*
subgoal
using *assms distinct-mset-dom*[*of bj*] *distinct-mset-dom*[*of aa*]
image-mset-cong2[*of* ‹*dom-m* (*fmdrop C aa*)›
‹λ*x*. *the* (*fmlookup* (*fmdrop C aa*) *x*)› ‹λ*x*. *the* (*fmlookup aa x*)› ‹*dom-m* (*fmdrop C aa*)›,
simplified]
apply (*cases* ‹*bj* ∝ *C*›)
apply (*auto simp*: *all-init-lits-of-l-def get-init-clss-l-def ran-m-def*
all-lits-of-mm-add-mset all-lits-of-mm-union all-lits-of-m-add-mset
all-learned-lits-of-l-def get-learned-clss-l-def
dest!: *multi-member-split*[*of* - ‹*dom-m* -›]
cong:)
apply (*metis in-clause-in-all-lits-of-m set-mset-mset*)
apply (*metis all-lits-of-m-add-mset member-add-mset multi-member-split set-mset-mset*)
apply (*smt* (*verit*, *best*) *image-mset-cong2*)+
done
subgoal
using *assms distinct-mset-dom*[*of bj*] *distinct-mset-dom*[*of aa*]
image-mset-cong2[*of* ‹*dom-m* (*fmdrop C aa*)›
‹λ*x*. *the* (*fmlookup* (*fmdrop C aa*) *x*)› ‹λ*x*. *the* (*fmlookup aa x*)› ‹*dom-m* (*fmdrop C aa*)›,
simplified]
apply (*cases* ‹*bj* ∝ *C*›)
apply (*auto simp*: *all-init-lits-of-l-def get-init-clss-l-def ran-m-def*
all-lits-of-mm-add-mset all-lits-of-mm-union all-lits-of-m-add-mset
all-learned-lits-of-l-def get-learned-clss-l-def
dest!: *multi-member-split*[*of* - ‹*dom-m* -›]
cong:)
apply (*smt* (*verit*, *best*) *image-mset-cong2*)+
apply (*metis in-clause-in-all-lits-of-m set-mset-mset*)
apply (*metis all-lits-of-m-add-mset member-add-mset multi-member-split set-mset-mset*)
apply (*smt* (*verit*, *best*) *image-mset-cong2*)+
done
done

have *in-lits2*: ‹*mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ⟹
C ∈# *dom-m bj* ⟹ *C* ∈# *dom-m aa* ⟹ *length* (*bj* ∝ *C*) > *0* ⟹
irred bj C ⟹ *irred aa C* ⟹ *bj* ∝ *C* ! *0* ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *init-clss-l aa*#}›
‹*mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ⟹
C ∈# *dom-m bj* ⟹ *C* ∈# *dom-m aa* ⟹ *length* (*bj* ∝ *C*) > *0* ⟹
irred bj C ⟹ *irred aa C* ⟹ −*bj* ∝ *C* ! *0* ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *init-clss-l aa*#}›
‹*mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ⟹
C ∈# *dom-m bj* ⟹ *C* ∈# *dom-m aa* ⟹ *length* (*bj* ∝ *C*) > *0* ⟹
¬*irred bj C* ⟹ ¬*irred aa C* ⟹ *bj* ∝ *C* ! *0* ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *learned-clss-l aa*#}› ‹*mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ⟹
C ∈# *dom-m bj* ⟹ *C* ∈# *dom-m aa* ⟹ *length* (*bj* ∝ *C*) > *0* ⟹
¬*irred bj C* ⟹ ¬*irred aa C* ⟹ −*bj* ∝ *C* ! *0* ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *learned-clss-l aa*#}›
‹*mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ⟹
fmdrop C bj = *fmdrop C aa* ⟹
C ∈# *dom-m bj* ⟹ *irred bj C* ⟹ *irred aa C* ⟹
C ∈# *dom-m aa* ⟹
xa ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *init-clss-l bj*#} ⟹ *xa* ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *init-clss-l aa*#}
›
‹*mset* (*bj* ∝ *C*) ⊆# *mset* (*aa* ∝ *C*) ⟹
fmdrop C bj = *fmdrop C aa* ⟹

C ∈# *dom-m bj* ⟹ ¬*irred bj C* ⟹ ¬*irred aa C* ⟹
C ∈# *dom-m aa* ⟹
xa ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *learned-clss-l bj*#} ⟹ *xa* ∈# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *learned-clss-l aa*#}
›
for *bj C aa xa*
apply (*auto 5 3 dest*!: *multi-member-split*[*of* - ‹*dom-m* -›] *simp*: *ran-m-def all-lits-of-mm-add-mset in-all-lits-of-mm-uminus-iff dest*: *all-lits-of-m-mono*)
apply ((*metis* (*no-types*, *lifting*) *in-clause-in-all-lits-of-m length-greater-0-conv mset-subset-eq-exists-conv nth-mem-mset union-iff*)+)[*4*]
apply (*blast dest*: *all-lits-of-m-mono*)
apply (*metis* (*no-types*, *lifting*) *add-mset-remove-trivial distinct-mset-add-mset distinct-mset-dom dom-m-fmdrop fmlookup-drop image-mset-cong2*)
apply (*blast dest*: *all-lits-of-m-mono*)
by (*metis* (*no-types*, *lifting*) *add-mset-remove-trivial distinct-mset-add-mset distinct-mset-dom dom-m-fmdrop fmlookup-drop image-mset-cong2*)
have *tauto*: ‹¬*tautology* (*mset* ((*get-clauses-l S*) ∝ *C*))›
using *list-invs C*
by (*auto simp*: *twl-list-invs-def*)
then show *?thesis*
supply [[*goals-limit*=*1*]]
using *assms*
unfolding *simplify-clause-with-unit-st-def simplify-clause-with-unit-def Let-def*
apply *refine-vcg*
subgoal using *assms* **by** *blast*
subgoal using *C* **by** *auto*
subgoal using *C* **by** *auto*
subgoal using *confl* **by** *auto*
subgoal using *clss* **by** *auto*
subgoal using *n-d* **by** *simp*
subgoal using *C list-invs* **by** (*auto simp*: *twl-list-invs-def*)
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal
by (*rule in-lits*)
subgoal
by (*rule in-lits*)
subgoal for *a b aa ba ab bb ac bc ad bd ae be af bf ag bg x ah bh*
supply[[*goals-limit*=*1*]]
using *count-decided-ge-get-level*[*of* ‹*get-trail-l S*›] *C0*
by (*auto simp*: *cdcl-twl-unitres-true-l.simps intro*!: *exI*[*of* - *C*])
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*

subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto* (*metis dom-m-fmdrop fmlookup-drop in-dom-m-lookup-iff*)+
subgoal
  by (*rule in-lits-prop*)
subgoal
  by (*rule in-lits-prop*)
subgoal by (*auto simp*: *all-init-lits-of-l-def*)
subgoal
  by (*subst add-new*, *subst* (*asm*) *eq-commute*[*of* ‹*set-mset* (*all-init-lits-of-l -*)›],
    subst (*asm*) *eq-commute*[*of* ‹*set-mset* (*all-learned-lits-of-l -*)›])
   (*auto simp*: *all-init-lits-of-l-def get-init-clss-l-def*
    all-learned-lits-of-l-def
    all-lits-of-mm-add-mset all-lits-of-m-add-mset)
subgoal for *a b aa ba ab bb ac bc ad bd ae be af bf ag bg ah bh ai bi x - - - - - - aj bj*
  using *ST st-invs C0 ent* **apply** −
  apply (*rule disjI2*, *rule disjI1*)
  apply (*auto simp*: *length-list-Suc-0*
    dest: *in-diffD*
    intro!: *cdcl-twl-unitres-l-intros2'' cdcl-twl-unitres-l-intros4''*
    intro: *cdcl-twl-unitres-l-intros2''*[**where** *C'* = ‹*mset* (*get-clauses-l S* ∝ *C*) − *mset* (*bj* ∝ *C*)›
    and *T* = *T*]
    cdcl-twl-unitres-l-intros4''[**where** *C'* = ‹*mset* (*get-clauses-l S* ∝ *C*) − *mset* (*bj* ∝ *C*)›
    and *T* = *T*])
  done
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto* (*metis dom-m-fmdrop fmlookup-drop in-dom-m-lookup-iff*)+
subgoal using *assms* **apply** (*auto simp*: *all-init-lits-of-l-def*
  all-lits-of-mm-union init-clss-l-fmdrop-if image-mset-remove1-mset-if
  get-init-clss-l-def all-lits-of-mm-add-mset all-lits-of-m-add-mset
  dest: *all-lits-of-mm-diffD in-lits*
  intro: *in-all-lits-of-mm-init-clss-l-single-out*)
  by (*metis all-lits-of-mm-add-mset diff-single-trivial insert-DiffM union-iff*)
subgoal using *assms* **apply** (*auto simp*: *all-learned-lits-of-l-def*
  all-lits-of-mm-union learned-clss-l-fmdrop-if image-mset-remove1-mset-if
  get-learned-clss-l-def all-lits-of-mm-add-mset all-lits-of-m-add-mset in-lits
  dest: *all-lits-of-mm-diffD*
  intro: *in-all-lits-of-mm-learned-clss-l-single-out*)
  by (*metis all-lits-of-mm-add-mset diff-single-trivial insert-DiffM union-iff*)
subgoal for *a b aa ba ab bb ac bc ad bd ae be af bf ag bg ah bh ai bi x - - - - - - aj bj*
  using *count-decided-ge-get-level*[*of* ‹*get-trail-l S*›] *ST st-invs C0 ent*
    cdcl-twl-unitres-false-entailed[*of C* ‹*get-clauses-l S*› ‹*get-trail-l S*›
     ‹*get-unkept-init-clauses-l S*› ‹*get-unkept-learned-clss-l S*›
     ‹*get-kept-init-clauses-l S*› ‹*get-kept-learned-clss-l S*›
    ‹*get-subsumed-init-clauses-l S*› ‹*get-subsumed-learned-clauses-l S*›
    ‹*get-init-clauses0-l S*› ‹*get-learned-clauses0-l S*›
    ‹*literals-to-update-l S*› *T*
    ‹*mset* (*get-clauses-l S* ∝ *C*) − *mset* (*bj* ∝ *C*)›]
    twl-st-l-struct-invs-distinct[*of S T None C*]
  apply (*auto simp*: *cdcl-twl-unitres-true-l.simps cdcl-twl-unitres-l.simps Un-assoc*
    dest: *distinct-mset-mono*[*of* ‹*mset* -› ‹*mset* -›, *unfolded distinct-mset-mset-distinct*]
    intro!: *exI*[*of* - *C*])
  done
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*

subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto* (*metis dom-m-fmdrop fmlookup-drop in-dom-m-lookup-iff*)+
subgoal using *assms* **apply** (*auto simp*: *all-init-lits-of-l-def*
all-lits-of-mm-union init-clss-l-fmdrop-if image-mset-remove1-mset-if
get-init-clss-l-def all-lits-of-mm-add-mset all-lits-of-m-add-mset
dest: *all-lits-of-mm-diffD in-lits2*
intro: *in-all-lits-of-mm-init-clss-l-single-out*)
apply (*smt* (*z3*) *Watched-Literals-Clauses.ran-m-mapsto-upd all-lits-of-mm-add-mset filter-mset-add-mset fmupd-same image-mset-add-mset prod.collapse ran-m-lf-fmdrop union-iff*)+
done
subgoal using *assms* **apply** (*auto simp*: *all-init-lits-of-l-def*
all-lits-of-mm-union init-clss-l-fmdrop-if image-mset-remove1-mset-if
get-init-clss-l-def all-lits-of-mm-add-mset all-lits-of-m-add-mset
get-learned-clss-l-def all-learned-lits-of-l-def
dest: *all-lits-of-mm-diffD in-lits2*
intro: *in-all-lits-of-mm-init-clss-l-single-out*)
apply (*smt* (*z3*) *Watched-Literals-Clauses.ran-m-mapsto-upd all-lits-of-mm-add-mset filter-mset-add-mset fmupd-same image-mset-add-mset prod.collapse ran-m-lf-fmdrop union-iff*)+
done
subgoal for *a b aa ba ab bb ac bc ad bd ae be af bf ag bg ah bh ai bi x - - - - - - aj bj*
using *assms*
by (*auto simp*: *list-length-2-isabelle-come-on twl-list-invs-def simplify-clause-with-unit-st-pre-def*
intro!: *cdcl-twl-unitres-l-intros3' cdcl-twl-unitres-l-intros1'*
dest: *distinct-mset-mono*[*of* ‹*mset* -› ‹*mset* -›, *unfolded distinct-mset-mset-distinct*]
intro!: *exI*[*of* - *C*] *fmdrop-eq-update-eq2*)
subgoal using *assms* **apply** (*auto simp*: *all-init-lits-of-l-def*
all-lits-of-mm-union init-clss-l-fmdrop-if image-mset-remove1-mset-if
get-init-clss-l-def all-lits-of-mm-add-mset all-lits-of-m-add-mset
dest: *all-lits-of-mm-diffD in-lits2*
intro: *in-all-lits-of-mm-init-clss-l-single-out*)
done
subgoal using *assms* **by** *auto*
(*metis dom-m-fmdrop insert-DiffM*)+
subgoal using *assms* **apply** *auto*
apply (*metis fmlookup-drop*)+
done
subgoal using *assms* **by** *auto*
done
qed

We implement forward subsumption (even if it is not a complete algorithm...). We initially tried to implement a very limited version similar to Splatz, i.e., putting all clauses in an array sorted by length and checking for sub-res in that array. This is more efficient in the sense that we know that all previous clauses are smaller... unless you want to strengthen binary clauses too. In this case list have to resorted.

After some time, we decided to implement forward subsumption directly. The implementation works in three steps:

- the *one−step* version tries to sub-res the clause with a given set of indices. Typically, one entry in the watch list.
- then the *try-to-forward-subsume* iterates over multiple of whatever. Typically, the iteration goes over all watch lists from the literals in the clause (and their negation) to find strengthening clauses.

- finally, the *all* version goes over all clauses sorted in a set of indices.

After much thinking, we decided to follow the version from CaDiCaL that ignores all clauses that contain a fixed literal. At first we tried to simplify the clauses, but this means that down in the real implementation, sorting clauses does not work (because the units are removed).

definition *forward-subsumption-one-pre* :: ‹*nat* ⇒ *nat multiset* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*forward-subsumption-one-pre* = (λ*C xs S*.
∃ *T*. *C* ≠ *0* ∧
(*S*, *T*) ∈ *twl-st-l None* ∧
twl-struct-invs T ∧
twl-list-invs S ∧
clauses-to-update-l S = {#} ∧
get-conflict-l S = *None* ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*) ∧
count-decided (*get-trail-l S*) = *0* ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
C ∈# *dom-m* (*get-clauses-l S*) ∧
(∀ *L* ∈# *mset* (*get-clauses-l S* ∝ *C*). *undefined-lit* (*get-trail-l S*) *L*) ∧
length (*get-clauses-l S* ∝ *C*) > *2*)›

datatype *'v subsumption* =
is-subsumed: *SUBSUMED-BY* (*subsumed-by*: *nat*) |
is-strengthened: *STRENGTHENED-BY* (*strengthened-on-lit*: ‹*'v literal*›)
(*strengthened-by*: *nat*) |
NONE

definition *try-to-subsume* :: ‹*nat* ⇒ *nat* ⇒ *'v clauses-l* ⇒ *'v subsumption* ⇒ *bool*› **where**
‹*try-to-subsume C C' N s* = (*case s of*
NONE ⇒ *True*
| *SUBSUMED-BY C''* ⇒ *mset* (*N* ∝ *C'*) ⊆# *mset* (*N* ∝ *C*) ∧ *C''* = *C'*
| *STRENGTHENED-BY L C''* ⇒ *L* ∈# *mset* (*N* ∝ *C'*) ∧ −*L* ∈# *mset* (*N* ∝ *C*) ∧
mset (*N* ∝ *C'*) − {#*L*#} ⊆# *mset* (*N* ∝ *C*) − {#−*L*#} ∧ *C''* = *C'*)›

definition *strengthen-clause-pre* **where**
‹*strengthen-clause-pre C C' L S* ⟷ (*C* ≠ *C'* ∧ *C*∈#*dom-m* (*get-clauses-l S*) ∧ *C'*∈#*dom-m* (*get-clauses-l S*))›

definition *strengthen-clause* :: ‹*nat* ⇒ *nat* ⇒ *'v literal* ⇒
'v twl-st-l ⇒ *'v twl-st-l nres*› **where**
‹*strengthen-clause* = (λ*C C' L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT (*strengthen-clause-pre C C' L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
E ← *SPEC* (λ*E*. *mset E* = *mset* (*remove1* (−*L*) (*N* ∝ *C*)));
if length (*N* ∝ *C*) = *2*
then do {
ASSERT (*length* (*remove1* (−*L*) (*N* ∝ *C*)) = *1*);
let L = *hd E*;
RETURN (*Propagated L 0* # *M*, *fmdrop C'* (*fmdrop C N*), *D*,
(*if irred N C' then add-mset* (*mset* (*N* ∝ *C'*)) *else id*) *NE*,
(*if* ¬*irred N C' then add-mset* (*mset* (*N* ∝ *C'*)) *else id*) *UE*,
(*if irred N C then add-mset* {#*L*#} *else id*) *NEk*, (*if* ¬*irred N C then add-mset* {#*L*#} *else id*) *UEk*,
((*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*)) *NS*,
((*if* ¬*irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*)) *US*,
N0, *U0*, *WS*, *add-mset* (−*L*) *Q*)
}

else if length $(N \propto C) =$ *length* $(N \propto C')$
then RETURN $(M$, *fmdrop* C' (*fmupd* C $(E$, *irred* N $C \vee$ *irred* N $C')$ $N)$, D, NE, UE, NEk, UEk,
((*if irred* N C' *then add-mset* (*mset* $(N \propto C')$) *else id*) o (*if irred* N C *then add-mset* (*mset* $(N \propto C)$) *else id*)) NS,
((*if* ¬*irred* N C' *then add-mset* (*mset* $(N \propto C')$) *else id*) o (*if* ¬*irred* N C *then add-mset* (*mset* $(N \propto C)$) *else id*)) US,
$N0$, $U0$, WS, Q)
else RETURN $(M$, *fmupd* C $(E$, *irred* N $C)$ N, D, NE, UE, NEk, UEk,
(*if irred* N C *then add-mset* (*mset* $(N \propto C)$) *else id*) NS,
(*if* ¬*irred* N C *then add-mset* (*mset* $(N \propto C)$) *else id*) US, $N0$, $U0$, WS, Q)
})›

definition *subsume-or-strengthen-pre* :: ‹*nat* ⇒ *'v subsumption* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*subsume-or-strengthen-pre* = (λC s S. ∃S'. *length* (*get-clauses-l* $S \propto C) \geq 2 \wedge C \in\#$ *dom-m* (*get-clauses-l* S) ∧
count-decided (*get-trail-l* S) = 0 ∧ *distinct* (*get-clauses-l* $S \propto C$) ∧ (∀ L∈*set* (*get-clauses-l* $S \propto C$). *undefined-lit* (*get-trail-l* S) L) ∧ *get-conflict-l* S = *None* ∧
$C \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S)) ∧ *clauses-to-update-l* $S = \{\#\}$ ∧
twl-list-invs S ∧
(S,S')∈*twl-st-l None* ∧ $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of* S') ∧
twl-struct-invs S' ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* S') ∧
(*case* s *of*
NONE ⇒ *True*
| *SUBSUMED-BY* C' ⇒ *mset* (*get-clauses-l* $S \propto C') \subseteq\#$ *mset* (*get-clauses-l* $S \propto C) \wedge C \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S)) ∧ *distinct* ((*get-clauses-l* S) $\propto C') \wedge C \neq C' \wedge$ ¬*tautology* (*mset* ((*get-clauses-l* S) $\propto C')) \wedge C' \in\#$ *dom-m* (*get-clauses-l* S)
| *STRENGTHENED-BY* L C' ⇒ $L \in\#$ *mset* ((*get-clauses-l* S) $\propto C') \wedge -L \in\#$ *mset* ((*get-clauses-l* S) $\propto C) \wedge$ ¬*tautology* (*mset* ((*get-clauses-l* S) $\propto C')) \wedge C' \neq 0 \wedge$
mset ((*get-clauses-l* S) $\propto C') - \{\#L\#\} \subseteq\#$ *mset* ((*get-clauses-l* S) $\propto C) - \{\#-L\#\} \wedge C' \in\#$ *dom-m* (*get-clauses-l* S) ∧ *distinct* ((*get-clauses-l* S) $\propto C')$ ∧
$C' \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S))∧ 2 ≤ *length* ((*get-clauses-l* S) $\propto C') \wedge$
¬ *tautology* (*remove1-mset* L (*mset* ((*get-clauses-l* S) $\propto C')$) + *remove1-mset* (− L) (*mset* ((*get-clauses-l* S) $\propto C)$)))))›

definition *subsume-or-strengthen* :: ‹*nat* ⇒ *'v subsumption* ⇒ *'v twl-st-l* ⇒ *- nres*› **where**
‹*subsume-or-strengthen* = (λC s (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, WS, Q). *do* {
ASSERT(*subsume-or-strengthen-pre* C s (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, WS, Q));
(*case* s *of*
NONE ⇒ *RETURN* (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, WS, Q)
| *SUBSUMED-BY* C' ⇒ *do* { *let* T= (M, *fmdrop* C (*if* ¬*irred* N $C' \wedge$ *irred* N C *then fmupd* C' ($N \propto C'$, *True*) N *else* N), D,
NE, UE, NEk, UEk, (*if irred* N C *then add-mset* (*mset* $(N \propto C)$) *else id*) NS,
(*if* ¬*irred* N C *then add-mset* (*mset* $(N \propto C)$) *else id*) US, $N0$, $U0$, WS, Q);
ASSERT (*set-mset* (*all-init-lits-of-l* T) = *set-mset* (*all-init-lits-of-l* (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, WS, Q)));
RETURN T
}
| *STRENGTHENED-BY* L C' ⇒ *strengthen-clause* C C' L (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, WS, Q))
})›

inductive *cdcl-twl-pure-remove-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*cdcl-twl-pure-remove-l* (M, N, *None*, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, $\{\#\}$, Q)
(*Propagated* L 0 # M, N, *None*, NE, UE, *add-mset* $\{\#L\#\}$ NEk, UEk, NS, US, $N0$, $U0$,
$\{\#\}$, *add-mset* $(-L) Q$)›

if
‹*undefined-lit M L*›
‹$-L \notin \bigcup$(*set-mset* ‘ *set-mset* (*mset* ‘# *init-clss-lf N*))›
‹*L* ∈# *all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE* + *NEk* + *NS* + *N0*)›
‹*count-decided M* = *0*›

lemma *cdcl-twl-pure-remove-l-cdcl-twl-pure-remove*:
assumes ‹*cdcl-twl-pure-remove-l S T*›
‹(*S*, *S′*) ∈ *twl-st-l None*›
shows ‹∃ *T′*. (*T*, *T′*) ∈ *twl-st-l None* ∧ *cdcl-twl-pure-remove S′ T′*›
using *assms*
by (*auto 5 3 simp*: *cdcl-twl-pure-remove-l.simps twl-st-l-def*
cdcl-twl-pure-remove.simps image-image in-all-lits-of-mm-ain-atms-of-iff
intro!: *convert-lit.intros*(*3*)
simp: *convert-lits-l-add-mset mset-take-mset-drop-mset′*
dest: *in-set-takeD in-set-dropD*)

lemma *rtranclp-cdcl-twl-pure-remove-l-cdcl-twl-pure-remove*:
assumes ‹*cdcl-twl-pure-remove-l*** *S T*›
‹(*S*, *S′*) ∈ *twl-st-l None*›
shows ‹∃ *T′*. (*T*, *T′*) ∈ *twl-st-l None* ∧ *cdcl-twl-pure-remove*** *S′ T′*›
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
subgoal by *blast*
subgoal for *T U*
using *cdcl-twl-pure-remove-l-cdcl-twl-pure-remove*[*of T U*] **by** *fastforce*
done

lemma *cdcl-twl-pure-remove-l-twl-list-invs*:
‹*cdcl-twl-pure-remove-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
by (*auto simp*: *cdcl-twl-pure-remove-l.simps*
twl-list-invs-def)

inductive *cdcl-twl-inprocessing-l* **for** *S T* **where**
‹*cdcl-twl-unitres-l S T* ⟹ *cdcl-twl-inprocessing-l S T*› |
‹*cdcl-twl-unitres-true-l S T* ⟹ *cdcl-twl-inprocessing-l S T*› |
‹*cdcl-twl-subsumed-l S T* ⟹ *cdcl-twl-inprocessing-l S T*›|
‹*cdcl-twl-subresolution-l S T* ⟹ *cdcl-twl-inprocessing-l S T*›|
‹*cdcl-twl-pure-remove-l S T* ⟹ *cdcl-twl-inprocessing-l S T*›

lemma *subseteq-mset-size-eql-iff*: ‹*size Aa* = *size A* ⟹ *Aa* ⊆# *A* ⟷ *Aa* = *A*›
by (*auto simp*: *subseteq-mset-size-eql*)

lemma *subresolution-strengtheningI*:
‹*C* ∈# *dom-m N* ⟹ *C′* ∈# *dom-m N* ⟹ −*L* ∈# *mset* (*N* ∝ *C*) ⟹ *L* ∈# *mset* (*N* ∝ *C′*) ⟹
count-decided M = *0* ⟹
remove1-mset (−*L*) (*mset* (*N* ∝ *C*)) ⊆# *remove1-mset L* (*mset* (*N* ∝ *C′*)) ⟹
distinct (*N* ∝ *C′*) ⟹ *3* ≤ *length* (*N* ∝ *C′*) ⟹
∀ *L*∈ *set* (*N* ∝ *C′*). *undefined-lit M L* ⟹
C′ ∉ *set* (*get-all-mark-of-propagated M*) ⟹
irred N C′ ⟹ *mset E* = *mset* (*remove1* (*L*) (*N* ∝ *C′*)) ⟹
*cdcl-twl-inprocessing-l*** (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *fmupd C′* (*E*, *True*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* (*N* ∝ *C′*)) *NS*,
US, *N0*, *U0*, {#}, *Q*)›
‹*C* ∈# *dom-m N* ⟹ *C′* ∈# *dom-m N* ⟹ −*L* ∈# *mset* (*N* ∝ *C*) ⟹ *L* ∈# *mset* (*N* ∝ *C′*) ⟹
count-decided M = *0* ⟹
remove1-mset (−*L*) (*mset* (*N* ∝ *C*)) ⊆# *remove1-mset L* (*mset* (*N* ∝ *C′*)) ⟹

$\neg$ *tautology* (*remove1-mset* ($-L$) (*mset* ($N \propto C$)) + *remove1-mset* L (*mset* ($N \propto C'$))) $\Longrightarrow$
distinct ($N \propto C'$) $\Longrightarrow$ 3 $\leq$ *length* ($N \propto C'$) $\Longrightarrow$
$\forall L \in$ *set* ($N \propto C'$). *undefined-lit* M L $\Longrightarrow$
$C' \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
$\neg$*irred* N C' $\Longrightarrow$ *mset* E = *mset* (*remove1* (L) ($N \propto C'$)) $\Longrightarrow$
*cdcl-twl-inprocessing-l*** (M, N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, Q)
(M, *fmupd* C' (E, *False*) N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *add-mset* (*mset* ($N \propto C'$)) *US*, *N0*, *U0*, {#}, Q)›
‹$C \in\#$ *dom-m* N $\Longrightarrow$ $C' \in\#$ *dom-m* N $\Longrightarrow$ $-L \in\#$ *mset* ($N \propto C$) $\Longrightarrow$ $L \in\#$ *mset* ($N \propto C'$) $\Longrightarrow$ *count-decided* M = 0 $\Longrightarrow$
remove1-mset ($-L$) (*mset* ($N \propto C$)) $\subseteq\#$ *remove1-mset* L (*mset* ($N \propto C'$)) $\Longrightarrow$
distinct ($N \propto C'$) $\Longrightarrow$ 3 $\leq$ *length* ($N \propto C'$) $\Longrightarrow$
$\forall L \in$ *set* ($N \propto C'$). *undefined-lit* M L $\Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
$C' \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$ *mset* E = *mset* (*remove1* (L) ($N \propto C'$)) $\Longrightarrow$
irred N C' $\Longrightarrow$ *irred* N C $\Longrightarrow$ *length* ($N \propto C'$) = *length* ($N \propto C$) $\Longrightarrow$
*cdcl-twl-inprocessing-l*** (M, N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, Q)
(M, *fmdrop* C (*fmupd* C' (E, *True*) N), *None*, *NE*, *UE*, *NEk*, *UEk*,
add-mset (*mset* ($N \propto C$)) (*add-mset* (*mset* ($N \propto C'$)) *NS*),
US, *N0*, *U0*, {#}, Q)›
‹$C \in\#$ *dom-m* N $\Longrightarrow$ $C' \in\#$ *dom-m* N $\Longrightarrow$ $-L \in\#$ *mset* ($N \propto C$) $\Longrightarrow$ $L \in\#$ *mset* ($N \propto C'$) $\Longrightarrow$ *count-decided* M = 0 $\Longrightarrow$
remove1-mset ($-L$) (*mset* ($N \propto C$)) $\subseteq\#$ *remove1-mset* L (*mset* ($N \propto C'$)) $\Longrightarrow$
distinct ($N \propto C'$) $\Longrightarrow$ 3 $\leq$ *length* ($N \propto C'$) $\Longrightarrow$
$\forall L \in$ *set* ($N \propto C'$). *undefined-lit* M L $\Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
$C' \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$ *mset* E = *mset* (*remove1* (L) ($N \propto C'$)) $\Longrightarrow$
$\neg$ *tautology* (*remove1-mset* ($-$ L) (*mset* ($N \propto C$)) + *remove1-mset* L (*mset* ($N \propto C'$))) $\Longrightarrow$
$\neg$*irred* N C' $\Longrightarrow$ $\neg$*irred* N C $\Longrightarrow$ *length* ($N \propto C'$) = *length* ($N \propto C$) $\Longrightarrow$
*cdcl-twl-inprocessing-l*** (M, N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, Q)
(M, *fmdrop* C (*fmupd* C' (E, *False*) N), *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*,
add-mset (*mset* ($N \propto C$)) (*add-mset* (*mset* ($N \propto C'$)) *US*), *N0*, *U0*, {#}, Q)›
‹$C \in\#$ *dom-m* N $\Longrightarrow$ $C' \in\#$ *dom-m* N $\Longrightarrow$ $-L \in\#$ *mset* ($N \propto C$) $\Longrightarrow$ $L \in\#$ *mset* ($N \propto C'$) $\Longrightarrow$ *count-decided* M = 0 $\Longrightarrow$
remove1-mset ($-L$) (*mset* ($N \propto C$)) $\subseteq\#$ *remove1-mset* L (*mset* ($N \propto C'$)) $\Longrightarrow$
distinct ($N \propto C'$) $\Longrightarrow$ 3 $\leq$ *length* ($N \propto C'$) $\Longrightarrow$
$\forall L \in$ *set* ($N \propto C'$). *undefined-lit* M L $\Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
$C' \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
$\neg$ *tautology* (*remove1-mset* ($-$ L) (*mset* ($N \propto C$)) + *remove1-mset* L (*mset* ($N \propto C'$))) $\Longrightarrow$
$\neg$*irred* N C' $\Longrightarrow$ *irred* N C $\Longrightarrow$ *length* ($N \propto C'$) = *length* ($N \propto C$) $\Longrightarrow$ *mset* E = *mset* (*remove1* (L) ($N \propto C'$)) $\Longrightarrow$
*cdcl-twl-inprocessing-l*** (M, N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, Q)
(M, *fmdrop* C (*fmupd* C' (E, *True*) N), *None*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* ($N \propto C$)) *NS*,
(*add-mset* (*mset* ($N \propto C'$)) *US*), *N0*, *U0*, {#}, Q)›
‹$C \in\#$ *dom-m* N $\Longrightarrow$ $C' \in\#$ *dom-m* N $\Longrightarrow$ $-L \in\#$ *mset* ($N \propto C$) $\Longrightarrow$ $L \in\#$ *mset* ($N \propto C'$) $\Longrightarrow$ *count-decided* M = 0 $\Longrightarrow$
remove1-mset ($-L$) (*mset* ($N \propto C$)) $\subseteq\#$ *remove1-mset* L (*mset* ($N \propto C'$)) $\Longrightarrow$
distinct ($N \propto C'$) $\Longrightarrow$ 3 $\leq$ *length* ($N \propto C'$) $\Longrightarrow$
$\forall L \in$ *set* ($N \propto C'$). *undefined-lit* M L $\Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
$C' \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$ *mset* E = *mset* (*remove1* (L) ($N \propto C'$)) $\Longrightarrow$
$\neg$ *tautology* (*remove1-mset* ($-$ L) (*mset* ($N \propto C$)) + *remove1-mset* L (*mset* ($N \propto C'$))) $\Longrightarrow$
irred N C' $\Longrightarrow$ $\neg$*irred* N C $\Longrightarrow$ *length* ($N \propto C'$) = *length* ($N \propto C$) $\Longrightarrow$
*cdcl-twl-inprocessing-l*** (M, N, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, Q)
(M, *fmdrop* C (*fmupd* C' (E, *True*) N), *None*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*mset* ($N \propto C'$)) *NS*,

(*add-mset* (*mset* (*N* ∝ *C*)) *US*), *N0*, *U0*, {#}, *Q*)›
and

subresolution-strengtheningI-binary:
‹*C* ∈# *dom-m N* ⟹ *C′* ∈# *dom-m N* ⟹ −*L* ∈# *mset* (*N* ∝ *C*) ⟹ *L* ∈# *mset* (*N* ∝ *C′*) ⟹ *count-decided M* = *0* ⟹
length (*N* ∝ *C*) = *2* ⟹ *remove1-mset* (*L*) (*mset* (*N* ∝ *C′*)) ⊆# *remove1-mset* (−*L*) (*mset* (*N* ∝ *C*)) ⟹
distinct (*N* ∝ *C′*) ⟹ *length* (*N* ∝ *C′*) ≥ *2* ⟹
∀ *L*∈ *set* (*N* ∝ *C*). *undefined-lit M L* ⟹
irred N C ⟹ *C′* ≠ *0* ⟹ *mset E* = *mset* (*remove1* (−*L*) (*N* ∝ *C*)) ⟹
irred N C′ ⟹ *C* ∉ *set* (*get-all-mark-of-propagated M*) ⟹ *C′* ∉ *set* (*get-all-mark-of-propagated M*) ⟹
*cdcl-twl-inprocessing-l***
(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated* (*hd E*) *0* # *M*, *fmdrop C′* (*fmdrop C N*), *None*, *add-mset* (*mset* (*N* ∝ *C′*)) *NE*, *UE*, *add-mset* {#*hd E*#} *NEk*, *UEk*,
(*add-mset* (*mset* (*N* ∝ *C*)) *NS*), *US*, *N0*, *U0*, {#}, *add-mset* (−*hd E*) *Q*)›
‹*C* ∈# *dom-m N* ⟹ *C′* ∈# *dom-m N* ⟹ −*L* ∈# *mset* (*N* ∝ *C*) ⟹ *L* ∈# *mset* (*N* ∝ *C′*) ⟹ *count-decided M* = *0* ⟹
length (*N* ∝ *C*) = *2* ⟹ *remove1-mset* (*L*) (*mset* (*N* ∝ *C′*)) ⊆# *remove1-mset* (−*L*) (*mset* (*N* ∝ *C*)) ⟹
distinct (*N* ∝ *C′*) ⟹ *length* (*N* ∝ *C′*) ≥ *2* ⟹
∀ *L*∈ *set* (*N* ∝ *C*). *undefined-lit M L* ⟹
¬*irred N C* ⟹ ¬*irred N C′* ⟹ *C′* ≠ *0* ⟹ *mset E* = *mset* (*remove1* (−*L*) (*N* ∝ *C*)) ⟹
C ∉ *set* (*get-all-mark-of-propagated M*) ⟹ *C′* ∉ *set* (*get-all-mark-of-propagated M*) ⟹
*cdcl-twl-inprocessing-l***
(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated* (*hd E*) *0* # *M*, *fmdrop C′* (*fmdrop C N*), *None*, *NE*, *add-mset* (*mset* (*N* ∝ *C′*)) *UE*, *NEk*,
add-mset {#*hd E*#} *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *C*)) *US*, *N0*, *U0*,
{#}, *add-mset* (− *hd E*) *Q*)›
‹*C* ∈# *dom-m N* ⟹ *C′* ∈# *dom-m N* ⟹ −*L* ∈# *mset* (*N* ∝ *C*) ⟹ *L* ∈# *mset* (*N* ∝ *C′*) ⟹ *count-decided M* = *0* ⟹
length (*N* ∝ *C*) = *2* ⟹ *remove1-mset* (*L*) (*mset* (*N* ∝ *C′*)) ⊆# *remove1-mset* (−*L*) (*mset* (*N* ∝ *C*)) ⟹
distinct (*N* ∝ *C′*) ⟹ *length* (*N* ∝ *C′*) ≥ *2* ⟹
∀ *L*∈ *set* (*N* ∝ *C*). *undefined-lit M L* ⟹
¬*irred N C* ⟹ *irred N C′* ⟹ *C′* ≠ *0* ⟹ *mset E* = *mset* (*remove1* (−*L*) (*N* ∝ *C*)) ⟹
C ∉ *set* (*get-all-mark-of-propagated M*) ⟹ *C′* ∉ *set* (*get-all-mark-of-propagated M*) ⟹
*cdcl-twl-inprocessing-l***
(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*Propagated* (*hd E*) *0* # *M*, *fmdrop C′* (*fmdrop C N*), *None*, *add-mset* (*mset* (*N* ∝ *C′*)) *NE*, *UE*, *NEk*,
add-mset {#*hd E*#} *UEk*, *NS*, *add-mset* (*mset* (*N* ∝ *C*)) *US*, *N0*, *U0*,
{#}, *add-mset* (− *hd E*) *Q*)›
‹*C* ∈# *dom-m N* ⟹ *C′* ∈# *dom-m N* ⟹ −*L* ∈# *mset* (*N* ∝ *C*) ⟹ *L* ∈# *mset* (*N* ∝ *C′*) ⟹ *count-decided M* = *0* ⟹
length (*N* ∝ *C*) = *2* ⟹ *remove1-mset* (*L*) (*mset* (*N* ∝ *C′*)) ⊆# *remove1-mset* (−*L*) (*mset* (*N* ∝ *C*)) ⟹
distinct (*N* ∝ *C′*) ⟹ *length* (*N* ∝ *C′*) ≥ *2* ⟹
∀ *L*∈ *set* (*N* ∝ *C*). *undefined-lit M L* ⟹
irred N C ⟹ ¬*irred N C′* ⟹ *C′* ≠ *0* ⟹ *mset E* = *mset* (*remove1* (−*L*) (*N* ∝ *C*)) ⟹
C ∉ *set* (*get-all-mark-of-propagated M*) ⟹ *C′* ∉ *set* (*get-all-mark-of-propagated M*) ⟹
*cdcl-twl-inprocessing-l***
(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)

(*Propagated* (*hd E*) *0* # *M*, *fmdrop C'* (*fmdrop C N*), *None*, *NE*,
add-mset (*mset* (*N* ∝ *C'*)) *UE*, *add-mset* {#*hd E*#} *NEk*, *UEk*,
add-mset (*mset* (*N* ∝ *C*)) *NS*, *US*, *N0*, *U0*, {#}, *add-mset* (− *hd E*) *Q*)›
unfolding *distinct-mset-mset-distinct*[*symmetric*] *set-mset-mset*[*symmetric*]
supply [*simp del*] = *distinct-mset-mset-distinct set-mset-mset*
supply [*simp*] = *distinct-mset-mset-distinct*[*symmetric*]
subgoal
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*cases* ‹*irred N C*›)
apply (*rule cdcl-twl-subresolution-l.intros*(*1*)[*of C N C'* ‹−*L*› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))›
‹*remove1-mset L* (*mset* (*N* ∝ *C'*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*;
fail)+
apply (*rule cdcl-twl-subresolution-l.intros*(*7*)[*of C N C'* ‹−*L*› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))›
‹*remove1-mset L* (*mset* (*N* ∝ *C'*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*;
fail)+
done
subgoal
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule r-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*cases* ‹¬*irred N C*›)
apply (*rule cdcl-twl-subresolution-l.intros*(*3*)[*of C N C'* ‹−*L*› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))›
‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C'*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*;
fail)+
apply (*rule cdcl-twl-subresolution-l.intros*(*5*)[*of C N C'* ‹−*L*› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))›
‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C'*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*;
fail)+
done
subgoal
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*cases* ‹*irred N C*›)
apply (*rule cdcl-twl-subresolution-l.intros*(*1*)[*of C N C'* ‹−*L*› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))›
‹*remove1-mset L* (*mset* (*N* ∝ *C'*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*;
fail)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*3*))
apply (*rule cdcl-twl-subsumed-l.intros*(*1*)[**where** *C*=*C'* **and** *C'*=*C*])
apply *normalize-goal*+
using *subseteq-mset-size-eql*[*of* ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset L* (*mset* (*N*
∝ *C'*))›] **apply** −
apply ((*subst* (*asm*) *size-mset*[*symmetric*])+; *hypsubst?*)
apply (*simp only*: *add-mset-remove-trivial size-add-mset nat.inject*)
apply (*simp*; *fail*)
apply (*auto*; *fail*)+
done
subgoal
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+

apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*3*)[*of C N C′ ‹−L› ‹remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*3*))
apply (*rule cdcl-twl-subsumed-l.intros*(*2*)[**where** *C*=*C′* **and** *C′*=*C*])
apply *normalize-goal*+
using *subseteq-mset-size-eql*[*of* ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset L* (*mset* (*N* ∝ *C′*))›] **apply** −
apply ((*subst* (*asm*) *size-mset*[*symmetric*])+; *hypsubst?*)
apply (*simp only*: *add-mset-remove-trivial size-add-mset nat.inject*)
apply (*simp*; *fail*)
apply (*auto*; *fail*)+
done
subgoal
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*5*)[*of C N C′ ‹−L› ‹remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*3*))
apply (*rule cdcl-twl-subsumed-l.intros*(*4*)[**where** *C*=*C′* **and** *C′*=*C*])
apply *normalize-goal*+
using *subseteq-mset-size-eql*[*of* ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset L* (*mset* (*N* ∝ *C′*))›] **apply** −
apply ((*subst* (*asm*) *size-mset*[*symmetric*])+; *hypsubst?*)
apply (*simp only*: *add-mset-remove-trivial size-add-mset nat.inject*)
apply (*simp*; *fail*)
apply (*auto simp*: *tautology-union*; *fail*)+
apply (*auto simp*: *fmdrop-fmupd*)
done
subgoal
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*7*)[*of C N C′ ‹−L› ‹remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))› - ‹*E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 simp flip*: *size-mset*; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*3*))
apply (*rule cdcl-twl-subsumed-l.intros*(*3*)[**where** *C*=*C′* **and** *C′*=*C*])
apply *normalize-goal*+
using *subseteq-mset-size-eql*[*of* ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› ‹*remove1-mset L* (*mset* (*N* ∝ *C′*))›] **apply** −
apply ((*subst* (*asm*) *size-mset*[*symmetric*])+; *hypsubst?*)
apply (*simp only*: *add-mset-remove-trivial size-add-mset nat.inject*)
apply (*simp*; *fail*)
apply (*auto simp*: *tautology-union*; *fail*)+
done

— Now binary:
subgoal
using *distinct-mset-dom*[*of N*] **apply** −
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*2*)[*of C′ N C* ‹*L*› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› - ‹*hd E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 length-list-2 add-mset-eq-add-mset subset-eq-mset-single-iff*; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*2*))
apply (*rule cdcl-twl-unitres-true-l.intros*(*1*)[**where** *N*=‹*fmdrop C N*› **and** *C*=*C′* **and** *L* = ‹*hd* (*remove1* (− *L*) (*N* ∝ *C*))›])
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff*)[]
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff* *dest*!: *multi-member-split*[*of* - ‹*dom-m N*›] ; *fail*)+
done
subgoal
using *distinct-mset-dom*[*of N*] **apply** −
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*4*)[*of C′ N C* ‹*L*› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› - ‹*hd* (*remove1* (−*L*) (*N* ∝ *C*))›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 length-list-2 add-mset-eq-add-mset*; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*2*))
apply (*rule cdcl-twl-unitres-true-l.intros*(*2*)[**where** *N*=‹*fmdrop C N*› **and** *C*=*C′* **and** *L* = ‹*hd* (*remove1* (− *L*) (*N* ∝ *C*))›])
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff*)[]
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff* *dest*!: *multi-member-split*[*of* - ‹*dom-m N*›] ; *fail*)+
done
subgoal
using *distinct-mset-dom*[*of N*] **apply** −
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*6*)[*of C′ N C* ‹*L*› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))› ‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› - ‹*hd E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 length-list-2 add-mset-eq-add-mset subset-eq-mset-single-iff*; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*2*))
apply (*rule cdcl-twl-unitres-true-l.intros*(*1*)[**where** *N*=‹*fmdrop C N*› **and** *C*=*C′* **and** *L* = ‹*hd E*›])
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff*)[]
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)

apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff*
dest!: *multi-member-split*[*of* - ‹*dom-m N*›]
; *fail*)+
done
subgoal
using *distinct-mset-dom*[*of N*] **apply** −
apply (*drule multi-member-split*[*of L*] *multi-member-split*[*of* ‹−*L*›])+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*4*))
apply (*rule cdcl-twl-subresolution-l.intros*(*8*)[*of C′ N C* ‹*L*› ‹*remove1-mset* (*L*) (*mset* (*N* ∝ *C′*))›
‹*remove1-mset* (−*L*) (*mset* (*N* ∝ *C*))› - ‹*hd E*›])
apply (*auto dest*!: *in-diffD simp*: *distinct-mset-remdups-mset-id length-remove1 length-list-2*
add-mset-eq-add-mset subset-eq-mset-single-iff; *fail*)+
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule cdcl-twl-inprocessing-l.intros*(*2*))
apply (*rule cdcl-twl-unitres-true-l.intros*(*2*)[**where** *N*=‹*fmdrop C N*› **and** *C*=*C′* **and**
L = ‹*hd E*›])
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff*)[]
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*metis add-mset-commute set-mset-mset union-single-eq-member*)
apply (*auto simp*: *add-mset-eq-add-mset length-list-2 subset-eq-mset-single-iff*
dest!: *multi-member-split*[*of* - ‹*dom-m N*›]
; *fail*)+
done
done

lemma *cdcl-twl-inprocessing-l-all-init-lits-of-l*:
assumes ‹*cdcl-twl-inprocessing-l S T*›
shows ‹*set-mset* (*all-init-lits-of-l S*) = *set-mset* (*all-init-lits-of-l T*)›
proof −
have [*simp*]: ‹*D* ∉# *A* ⟹ {#*the* (*if D* = *x then b else fmlookup N x*). *x* ∈# *A*#} =
{#*the* (*fmlookup N x*). *x* ∈# *A*#}›
‹*D* ∉# *A* ⟹ {#*the* (*if x* = *D then b else fmlookup N x*). *x* ∈# *A*#} =
{#*the* (*fmlookup N x*). *x* ∈# *A*#}› **for** *D E N A b*
by (*auto intro*!: *image-mset-cong*)
have *dups-uniq*[*dest*]: ‹*remdups-mset D′* = {#*K*#} ⟹ *set-mset* (*all-lits-of-m D′*) = {−*K*,*K*}› **for**
D′ K
by (*metis all-lits-of-m-add-mset all-lits-of-m-empty all-lits-of-m-remdups-mset*
insert-commute set-mset-add-mset-insert set-mset-empty)
have [*simp*]: ‹− *L* ∈# *all-lits-of-m a* ⟷ *L* ∈# *all-lits-of-m a*›
‹− *L* ∈# *all-lits-of-mm b* ⟷ *L* ∈# *all-lits-of-mm b*›
‹*L* ∈# *xb* ⟹ *L* ∈# *all-lits-of-m xb*› **for** *L a xb b*
by (*solves* ‹*cases L*, *auto simp*: *rev-image-eqI all-lits-of-m-def all-lits-of-mm-def*›)+

have [*simp*]: ‹*L* ∈# *all-lits-of-m* (*mset* (*N* ∝ *xa*)) ⟷ *L* ∈ *set* (*N* ∝ *xa*) ∨ −*L* ∈ *set* (*N* ∝ *xa*)›
for *L N xa xb*
by (*simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-all-lits-of-m-ain-atms-of-iff*)

show *?thesis*
using *assms*
using *distinct-mset-dom*[*of* ‹*get-clauses-l S*›] **apply** −
supply [[*goals-limit*=*1*]]
apply (*induction rule*: *cdcl-twl-inprocessing-l.induct*)
by (*auto 4 3 simp*: *cdcl-twl-unitres-l.simps*
cdcl-twl-unitres-true-l.simps

cdcl-twl-subsumed-l.simps
cdcl-twl-subresolution-l.simps
all-init-lits-of-l-def
add-mset-eq-add-mset removeAll-notin
get-init-clss-l-def init-clss-l-mapsto-upd-notin
init-clss-l-mapsto-upd ran-m-def all-lits-of-m-union
all-lits-of-m-add-mset distinct-mset-remove1-All
init-clss-l-fmupd-if all-lits-of-mm-add-mset
all-lits-of-m-remdups-mset
dest!: *multi-member-split*[*of* ‹- :: *nat*› ‹-›]
dest: *all-lits-of-m-mono*)[*4*]
(*auto simp*: *cdcl-twl-pure-remove-l.simps*
all-init-lits-of-l-def
add-mset-eq-add-mset removeAll-notin
get-init-clss-l-def init-clss-l-mapsto-upd-notin
init-clss-l-mapsto-upd ran-m-def all-lits-of-m-union
all-lits-of-m-add-mset distinct-mset-remove1-All
init-clss-l-fmupd-if all-lits-of-mm-add-mset all-lits-of-mm-union
all-lits-of-m-remdups-mset
dest!: *multi-member-split*[*of* ‹-› ‹- :: - *clauses*›]
multi-member-split[*of* ‹- :: *nat*› ‹-›]
dest: *all-lits-of-m-mono*
intro: *in-clause-in-all-lits-of-m*)
qed

lemma *rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*:
assumes ‹*cdcl-twl-inprocessing-l*** *S T*›
shows ‹*set-mset* (*all-init-lits-of-l S*) = *set-mset* (*all-init-lits-of-l T*)›
using *assms*
by (*induction rule*: *rtranclp-induct*) (*auto dest*: *cdcl-twl-inprocessing-l-all-init-lits-of-l*)

lemma *subsume-or-strengthen*:
assumes ‹*subsume-or-strengthen-pre C s S*› ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
shows
‹*subsume-or-strengthen C s S* ≤⇓*Id* (*SPEC*(λ*T*. *cdcl-twl-inprocessing-l*** *S T* ∧
(*s* = *NONE* ⟶ *T* = *S*) ∧
(*length* (*get-clauses-l S* ∝ *C*) > *2* ⟶ *get-trail-l S* = *get-trail-l T*) ∧
(∀ *D*∈#*remove1-mset C* (*dom-m* (*get-clauses-l T*)).
D ∈# *dom-m* (*get-clauses-l S*) ∧ *get-clauses-l T* ∝ *D* = *get-clauses-l S* ∝ *D*) ∧
(∀ *D*. *D* ≠ *C* ⟶ *is-strengthened s* ⟶ *D* ≠ *strengthened-by s* ⟶ (*D* ∈# *dom-m* (*get-clauses-l T*)) = (*D* ∈# *dom-m* (*get-clauses-l S*))) ∧
(∀ *D*. *D* ≠ *C* ⟶ *is-subsumed s* ⟶ *D* ≠ *subsumed-by s* ⟶ (*D* ∈# *dom-m* (*get-clauses-l T*)) = (*D* ∈# *dom-m* (*get-clauses-l S*)))))›
proof −
have [*iff*]: ‹*C* ∈# *remove1-mset C* (*dom-m N*) ⟷ *False*›
‹*C* ∈# *dom-m N* − {#*C*, *C*#} ⟷ *False*›**for** *N*
using *distinct-mset-dom*[*of N*] **by** (*cases* ‹*C*∈#*dom-m N*›; *auto dest*: *multi-member-split*)+
have [*iff*]: ‹*C* ∈# *dom-m N* − {#*C*, *x22*, *C*#} ⟷ *False*› ‹*C* ∈# *dom-m N* − {#*C*, *x22*#} ⟷ *False*›
‹*x22* ∈# *dom-m N* − {#*C*, *x22*, *C*#} ⟷ *False*›
‹*x22* ∈# *dom-m N* − {#*C*, *x22*#} ⟷ *False*›
for *N x22*
using *distinct-mset-dom*[*of N*] **by** (*cases* ‹*C*∈#*dom-m N*›;*cases* ‹*x22*∈#*dom-m N*›; *cases* ‹*C*=*x22*›; *auto dest*: *multi-member-split*; *fail*)+
have [*iff*]: ‹*C* ∈# *add-mset C'* (*remove1-mset C'* (*dom-m N*)) − {#*C*, *C*#} ⟷ *False*› **for** *C'* *N*
using *distinct-mset-dom*[*of N*] **by** (*cases* ‹*C*∈#*dom-m N*›;*cases* ‹*C'*∈#*dom-m N*›; *cases* ‹*C*=*C'*›;

auto dest: *multi-member-split*)
show *?thesis*
using *assms* **unfolding** *subsume-or-strengthen-def*
apply *refine-vcg+*
subgoal by *auto*
subgoal for *M b N ba NE bb UE bc NEk bd UEk be NS bf US bg NS0 bh US0 bi WS bj Q occs*
apply (*cases s*)
subgoal for *C'*
apply (*simp only*: *Let-def subsumption.case*(*1*))
apply *refine-vcg*
subgoal
apply (*subgoal-tac* ‹*cdcl-twl-inprocessing-l***
(*M*, *N*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *NS0*, *US0*, *WS*, *Q*, *occs*)
(*M*, *fmdrop C* (*if* ¬ *irred N C'* ∧ *irred N C then fmupd C'* (*N* ∝ *C'*, *True*) *N else N*),
NE, *UE*, *NEk*, *UEk*, *NS*,
(*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *US*,
(*if* ¬ *irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *NS0*, *US0*, *WS*, *Q*, *occs*)›)
subgoal
by (*frule rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*)
presburger
subgoal by *auto* (*auto 8 8 simp*: *subsume-or-strengthen-pre-def cdcl-twl-subsumed-l.simps fm-drop-fmupd*
intro!: *cdcl-twl-inprocessing-l.intros*(*3*) *r-into-rtranclp dest*!: *in-diffD*)
done
subgoal by *auto* (*auto 8 8 simp*: *subsume-or-strengthen-pre-def cdcl-twl-subsumed-l.simps fm-drop-fmupd*
intro!: *cdcl-twl-inprocessing-l.intros*(*3*) *r-into-rtranclp dest*!: *in-diffD*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto* (*auto 8 8 simp*: *subsume-or-strengthen-pre-def cdcl-twl-subsumed-l.simps fm-drop-fmupd*
intro!: *cdcl-twl-inprocessing-l.intros*(*3*) *r-into-rtranclp dest*!: *in-diffD*)
subgoal by *auto*
subgoal by *auto*
done
subgoal
apply (*clarsimp simp only*: *strengthen-clause-def subsumption.case Let-def*
intro!: *ASSERT-leI impI*)
apply (*intro ASSERT-leI intro-spec-iff*[*THEN iffD2*]; (*split if-splits*[*of* - ‹- = (*2*::*nat*)›])*?*; (*intro conjI impI ballI ASSERT-leI*)*?*)
subgoal by (*auto simp*: *length-remove1 strengthen-clause-pre-def subsume-or-strengthen-pre-def*)
subgoal
by (*auto simp*: *strengthen-clause-def subsume-or-strengthen-pre-def length-remove1 Let-def*
subresolution-strengtheningI-binary
intro!: *ASSERT-leI*
dest: *in-diffD*)
subgoal
by (*auto simp*: *strengthen-clause-def subsume-or-strengthen-pre-def length-remove1 Let-def*
intro: *subresolution-strengtheningI-binary*
intro!: *ASSERT-leI*
dest: *in-diffD*)
subgoal
by (*auto simp*: *strengthen-clause-def subsume-or-strengthen-pre-def length-remove1 Let-def*
intro!: *subresolution-strengtheningI*(*1*)[*of* ‹*strengthened-by s*›]
subresolution-strengtheningI(*2*)[*of* ‹*strengthened-by s*›]
subresolution-strengtheningI(*3*−)

intro!: *ASSERT-leI*
dest: *in-diffD*)
done
subgoal
by (*auto dest*: *in-diffD*)
done
done
qed

lemma
assumes ‹*cdcl-twl-inprocessing-l*** *S T*›
shows
rtranclp-cdcl-twl-inprocessing-l-count-decided:
‹*count-decided* (*get-trail-l T*) = *count-decided* (*get-trail-l S*)› (**is** *?A*) **and**
rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l:
‹*clauses-to-update-l T* = *clauses-to-update-l S*› (**is** *?B*) **and**
rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated:
‹*set* (*get-all-mark-of-propagated* (*get-trail-l T*)) ⊆ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∪ {*0*}› (**is** *?C*)
proof −
have [*dest*]:
‹*cdcl-twl-inprocessing-l S T* ⟹ *count-decided* (*get-trail-l T*) = *count-decided* (*get-trail-l S*)›
‹*cdcl-twl-inprocessing-l S T* ⟹ *clauses-to-update-l T* = *clauses-to-update-l S*›
‹*cdcl-twl-inprocessing-l S T* ⟹ *set* (*get-all-mark-of-propagated* (*get-trail-l T*)) ⊆ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∪ {*0*}› **for** *S T*
by (*auto simp*: *cdcl-twl-inprocessing-l.simps cdcl-twl-subsumed-l.simps*
cdcl-twl-unitres-l.simps cdcl-twl-subresolution-l.simps cdcl-twl-unitres-true-l.simps
cdcl-twl-pure-remove-l.simps)

show *?A ?B ?C*
using *assms*
by (*induction rule*: *rtranclp-induct*; *auto*; *fail*)+
qed

lemma *cdcl-twl-inprocessing-l-twl-st-l0*:
assumes ‹*cdcl-twl-inprocessing-l S U*› **and**
‹(*S*, *T*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs T*› **and**
‹*twl-list-invs S*›
obtains *V* **where**
‹(*U*, *V*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-unitres T V* ∨ *cdcl-twl-unitres-true T V* ∨ *cdcl-twl-subsumed T V* ∨
cdcl-twl-subresolution T V ∨ *cdcl-twl-pure-remove T V*›
using *assms*
apply (*induction rule*: *cdcl-twl-inprocessing-l.induct*)
using *cdcl-twl-unitres-l-cdcl-twl-unitres* **apply** *blast*
using *cdcl-twl-unitres-true-l-cdcl-twl-unitres-true* **apply** *blast*
using *cdcl-twl-subsumed-l-cdcl-twl-subsumed* **apply** *blast*
using *cdcl-twl-subresolution-l-cdcl-twl-subresolution* **apply** *blast*
using *cdcl-twl-pure-remove-l-cdcl-twl-pure-remove* **by** *blast*

lemma *cdcl-twl-unitres-l-list-invs*:
‹*cdcl-twl-unitres-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
by (*induction rule*: *cdcl-twl-unitres-l.induct*)

(*auto simp*: *twl-list-invs-def cdcl$_W$-restart-mset.in-get-all-mark-of-propagated-in-trail*
ran-m-mapsto-upd
dest: *in-diffD*)

lemma *cdcl-twl-inprocessing-l-twl-list-invs*:
assumes ‹*cdcl-twl-inprocessing-l S U*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs U*›
using *assms* **by** (*induction rule*: *cdcl-twl-inprocessing-l.induct*)
(*auto simp*: *cdcl-twl-unitres-True-l-list-invs*
cdcl-twl-unitres-l-list-invs cdcl-twl-subsumed-l-list-invs
cdcl-twl-subresolution-l-list-invs cdcl-twl-pure-remove-l-twl-list-invs)

lemma *rtranclp-cdcl-twl-inprocessing-l-twl-list-invs*:
assumes ‹*cdcl-twl-inprocessing-l*** *S U*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs U*›
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto simp*: *cdcl-twl-inprocessing-l-twl-list-invs*)

lemma *cdcl-twl-inprocessing-l-twl-st-l*:
assumes ‹*cdcl-twl-inprocessing-l S U*› **and**
‹(*S*, *T*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs T*› **and**
‹*twl-list-invs S*› **and**
‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of T*)›
obtains *V* **where**
‹(*U*, *V*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-unitres T V* ∨ *cdcl-twl-unitres-true T V* ∨ *cdcl-twl-subsumed T V* ∨
cdcl-twl-subresolution T V ∨ *cdcl-twl-pure-remove T V*› **and**
‹*twl-struct-invs V*› **and**
‹*twl-list-invs U*›**and**
‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of V*)›
apply (*rule cdcl-twl-inprocessing-l-twl-st-l0*[*OF assms*(*1*−*4*)])
subgoal premises *p* **for** *V*
using *p*(*2*−) *cdcl-twl-inprocessing-l-twl-list-invs*[*OF assms*(*1*)] **apply** −
apply (*rule that*[*of V*])
apply (*auto intro*: *cdcl-twl-unitres-true-twl-struct-invs*
cdcl-twl-unitres-struct-invs assms cdcl-twl-subsumed-struct-invs
cdcl-twl-subresolution-twl-struct-invs cdcl-twl-unitres-twl-stgy-invs
cdcl-twl-subresolution-twl-stgy-invs cdcl-twl-unitres-true-twl-stgy-invs
cdcl-twl-subsumed-twl-stgy-invs
cdcl-twl-pure-remove-twl-struct-invs)
apply (*metis assms*(*3*) *assms*(*5*) *cdcl-twl-inp.intros cdcl-twl-inp-invs*(*3*) *state$_W$-of-def*)+
done
done

lemma *rtranclp-cdcl-twl-inprocessing-l-twl-st-l*:
assumes ‹*cdcl-twl-inprocessing-l*** *S U*› **and**
‹(*S*, *T*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs T*› **and**
‹*twl-list-invs S*› **and**

‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of T)›
obtains V **where**
‹$(U,\ V) \in$ twl-st-l None› **and**
‹twl-struct-invs V› **and**
‹twl-list-invs U› **and**
‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of V)›
using *assms*
apply (*induction rule*: *rtranclp-induct*)
subgoal by *blast*
subgoal premises p **for** $X\ Y$
apply (*rule* $p(3)$[*OF* - $p(5-)$])
apply (*rule cdcl-twl-inprocessing-l-twl-st-l*[*of* $X\ Y$, *OF* $p(2)$]; *assumption?*)
apply (*rule* $p(4)$)
apply *assumption*+
done
done

Forward subsumption is done in two steps. First we work on the binary clauses (also deduplicationg them), and only then we work on other clauses.

Short version:

1. first, we work only on binary clauses ;
2. second, we work on all other clauses.

Longer version: We already implement the functions towards how we will need implement it (although this just slightly more general that what would be needed to implement the Splatz version).

The *forward-all* schedules all clauses. This functions leaves the work to subsume one clause to the function *try-to-subsume*. This is the function that is slightly more specialized: it allows to test subsumption on n different times (potentially only once). Finally it is the funtion *forward-one* that compares two clauses and checks for subsumption. We assume that no new unit clause is produced (as only binary clauses can produce new clauses).

The names follow the corresponding functions from CaDiCaL. In newer minimal solvers from Armin (like satch or Gimsatul), only vivification is implemented.

definition *clause-remove-duplicate-clause-pre* :: ‹-› **where**
‹*clause-remove-duplicate-clause-pre* $C\ S \longleftrightarrow (\exists\, C'\ S'.$
$(S,S') \in$ *twl-st-l None* $\wedge$
mset ((*get-clauses-l* S) $\propto$ C) = *mset* ((*get-clauses-l* S) $\propto$ C') $\wedge$
($\neg$*irred* (*get-clauses-l* S) $C' \longrightarrow \neg$*irred* (*get-clauses-l* S) C) $\wedge$
$C' \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S)) $\wedge$
$C \in\#$ *dom-m* (*get-clauses-l* S) $\wedge$
clauses-to-update-l $S = \{\#\} \wedge$
$C' \in\#$ *dom-m* (*get-clauses-l* S) $\wedge$
$C \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S)) $\wedge$
$C \neq C' \wedge$
twl-list-invs $S \wedge$ *twl-struct-invs* $S' \wedge$
$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of S'))›

definition *clause-remove-duplicate-clause* :: ‹*nat* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*clause-remove-duplicate-clause* C = ($\lambda(M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ WS,\ Q)$. *do*
{
ASSERT (*clause-remove-duplicate-clause-pre* C (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, WS, Q));

RETURN (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *UEk*, (*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *NS*, (*if irred N C then id else add-mset* (*mset* (*N* ∝ *C*))) *US*, *N0*, *U0*, *WS*, *Q*)
})›

definition *clause-remove-duplicate-clause-spec* **where**
‹*clause-remove-duplicate-clause-spec C S T* ⟷ *cdcl-twl-inprocessing-l S T* ∧
get-clauses-l T = *fmdrop C* (*get-clauses-l S*) ∧ *get-trail-l T* = *get-trail-l S*›

lemma *clause-remove-duplicate-clause*:
assumes ‹*clause-remove-duplicate-clause-pre C S*›
shows ‹*clause-remove-duplicate-clause C S* ≤ *SPEC*(*clause-remove-duplicate-clause-spec C S*)›
using *assms*
unfolding *clause-remove-duplicate-clause-def clause-remove-duplicate-clause-pre-def*
clause-remove-duplicate-clause-spec-def **apply** −
apply *normalize-goal*+
apply (*cases S*; *hypsubst*)
unfolding *prod.simps*
apply (*intro ASSERT-leI*, *rule-tac x*=*x* **in** *exI*, *rule-tac x*=*xa* **in** *exI*)
apply *auto* []
apply *simp*
apply (*intro impI conjI cdcl-twl-inprocessing-l.intros*(*3*))
using *cdcl-twl-subsumed-l.intros*[*of* ‹*get-clauses-l S*› - *C* ‹*get-trail-l S*› ‹*get-conflict-l S*›
‹*get-unkept-init-clauses-l S*› ‹*get-unkept-learned-clss-l S*› ‹*get-kept-init-clauses-l S*› ‹*get-kept-learned-clss-l S*›
‹*get-subsumed-init-clauses-l S*› ‹*get-subsumed-learned-clauses-l S*› ‹*get-init-clauses0-l S*› ‹*get-learned-clauses0-l S*›
‹*literals-to-update-l S*›]
apply (*auto simp*:)
done

definition *binary-clause-subres-lits-pre* :: ‹-› **where**
‹*binary-clause-subres-lits-pre C L L′ S* ⟷ (∃ *C′ S′*.
(*S*,*S′*) ∈ *twl-st-l None* ∧
mset (*get-clauses-l S* ∝ *C*) = {#*L*, −*L′*#} ∧ *mset* (*get-clauses-l S* ∝ *C′*)= {#*L*, *L′*#} ∧
get-conflict-l S = *None* ∧
clauses-to-update-l S = {#} ∧
C ∈# *dom-m* (*get-clauses-l S*) ∧
C′ ∈# *dom-m* (*get-clauses-l S*) ∧
count-decided (*get-trail-l S*) = *0* ∧
undefined-lit (*get-trail-l S*) *L* ∧
C ∉ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∧
twl-list-invs S ∧
twl-struct-invs S′ ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S′*))›

definition *binary-clause-subres* :: ‹*nat* ⇒ *'v literal* ⇒ *'v literal* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*binary-clause-subres C L L′* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT (*binary-clause-subres-lits-pre C L L′* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
RETURN (*Propagated L 0* # *M*, *fmdrop C N*, *D*, *NE*, *UE*,
(*if irred N C then add-mset* {#*L*#} *else id*) *NEk*, (*if irred N C then id else add-mset* {#*L*#}) *UEk*,
(*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *NS*, (*if irred N C then id else add-mset* (*mset* (*N* ∝ *C*))) *US*,
N0, *U0*, *WS*, *add-mset* (−*L*) *Q*)
})›

lemma *binary-clause-subres-binary-clause*:
assumes ‹*binary-clause-subres-lits-pre C L L′ S*›
shows ‹*binary-clause-subres C L L′ S* ≤ *SPEC*(*cdcl-twl-inprocessing-l S*)›
using *assms* **unfolding** *binary-clause-subres-def binary-clause-subres-lits-pre-def binary-clause-subres-lits-pre-def*
apply −
apply *normalize-goal+*
subgoal for *C′ xa*
apply (*cases S*; *hypsubst*)
unfolding *prod.simps*
apply (*intro ASSERT-leI exI*[*of - C′*] *exI*[*of - xa*])
apply *auto* []
using *cdcl-twl-subresolution-l.intros*(*2*,*4*,*6*,*8*)[*of C′* ‹*get-clauses-l S*› *C* ‹*L′*› ‹{#*L*#}› ‹{#*L*#}›
‹*get-trail-l S*›
L ‹*get-unkept-init-clauses-l S*› ‹*get-unkept-learned-clss-l S*› ‹*get-kept-init-clauses-l S*› ‹*get-kept-learned-clss-l S*›
‹*get-subsumed-init-clauses-l S*› ‹*get-subsumed-learned-clauses-l S*› ‹*get-init-clauses0-l S*› ‹*get-learned-clauses0-l S*›
‹*literals-to-update-l S*›, *unfolded assms*, *simplified*]
apply (*cases* ‹*irred* (*get-clauses-l S*) *C′*›)
apply (*auto simp add*: *add-mset-commute cdcl-twl-inprocessing-l.intros*(*4*))
done
done

definition *deduplicate-binary-clauses-pre* **where**
‹*deduplicate-binary-clauses-pre L S* ⟷
(∃ *T*. (*S*,*T*) ∈ *twl-st-l None* ∧ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*) ∧ *count-decided* (*get-trail-l S*) = *0* ∧
clauses-to-update-l S = {#} ∧ *twl-list-invs S* ∧ *twl-struct-invs T* ∧ *L* ∈# *all-init-lits-of-l S*)›

definition *deduplicate-binary-clauses-correctly-marked* **where**
‹*deduplicate-binary-clauses-correctly-marked L xs0 xs CS T* ⟷
(∀ *C L′ b*. *CS L′* = *Some* (*C*, *b*) ⟶ (*C* ∈# *dom-m* (*get-clauses-l T*) ∧ *C*∈# *xs0* − *xs* ∧ *mset* (*get-clauses-l T* ∝ *C*) = {#*L*,*L′*#} ∧ *irred* (*get-clauses-l T*) *C* = *b*))›

definition *deduplicate-binary-clauses-inv*
:: ‹′*v literal* ⇒ - ⇒ ′*v twl-st-l* ⇒ *bool* × *nat multiset* × - × ′*v twl-st-l* ⇒ *bool*›
where
‹*deduplicate-binary-clauses-inv L xs0 S* = (λ(*abort*, *xs*, *CS*, *T*).
∃ *S′*. (*S*,*S′*) ∈ *twl-st-l None* ∧ *twl-struct-invs S′* ∧ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S′*) ∧
*cdcl-twl-inprocessing-l*** *S T* ∧ *xs* ⊆# *xs0* ∧
(¬*abort* ⟶ *deduplicate-binary-clauses-correctly-marked L xs0 xs CS T*) ∧
twl-list-invs S ∧
count-decided (*get-trail-l S*) = *0* ∧
clauses-to-update-l S = {#} ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
(¬*abort* ⟶ *undefined-lit* (*get-trail-l T*) *L*))›

lemma *deduplicate-binary-clauses-inv-alt-def*:
‹*deduplicate-binary-clauses-inv L xs0 S* = (λ(*abort*, *xs*, *CS*, *T*).
∃ *S′ T′*. (*S*,*S′*) ∈ *twl-st-l None* ∧ *twl-struct-invs S′* ∧ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S′*) ∧
(*T*,*T′*) ∈ *twl-st-l None* ∧ *twl-struct-invs T′* ∧ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init*

(*state$_W$-of T'*) ∧
  *cdcl-twl-inprocessing-l*** *S T* ∧ *xs* ⊆# *xs0* ∧
  (¬*abort* ⟶ *deduplicate-binary-clauses-correctly-marked L xs0 xs CS T*) ∧
  twl-list-invs S ∧
  count-decided (*get-trail-l S*) = *0* ∧
  clauses-to-update-l S = {#} ∧
  set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
  twl-list-invs T ∧
  count-decided (*get-trail-l T*) = *0* ∧
  clauses-to-update-l T = {#} ∧
  set (*get-all-mark-of-propagated* (*get-trail-l T*)) ⊆ {*0*} ∧
  (¬*abort* ⟶*undefined-lit* (*get-trail-l T*) *L*))›
  unfolding *deduplicate-binary-clauses-inv-def*
  apply (*intro ext iffI*)
  unfolding *case-prod-beta* **apply** *normalize-goal+*
  apply (*rule-tac x=xa* **in** *exI*)
  apply (*frule rtranclp-cdcl-twl-inprocessing-l-twl-st-l*; *assumption?*)
  apply (*rule-tac x=V* **in** *exI*)
  apply (*auto 5 3 dest*: *rtranclp-cdcl-twl-inprocessing-l-count-decided*
    rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l rtranclp-cdcl-twl-inprocessing-l-twl-list-invs
    rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated
    elim: *rtranclp-cdcl-twl-inprocessing-l-twl-st-l*)[]
  unfolding *case-prod-beta* **apply** *normalize-goal+*
  apply (*rule-tac x=xa* **in** *exI*)
  apply (*auto 5 3 dest*: *rtranclp-cdcl-twl-inprocessing-l-count-decided*
    rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l rtranclp-cdcl-twl-inprocessing-l-twl-list-invs
    rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated rtranclp-cdcl-twl-inprocessing-l-twl-st-l)[]
  done

lemma *deduplicate-binary-clauses-inv-alt-def2*:
  ‹ *deduplicate-binary-clauses-pre L S* ⟹
  deduplicate-binary-clauses-inv L xs0 S = (λ(*abort*, *xs*, *CS*, *T*).
    *cdcl-twl-inprocessing-l*** *S T* ∧ *xs* ⊆# *xs0* ∧
    (¬*abort* ⟶ *deduplicate-binary-clauses-correctly-marked L xs0 xs CS T*) ∧
    (¬*abort* ⟶ *undefined-lit* (*get-trail-l T*) *L*))›
  unfolding *deduplicate-binary-clauses-pre-def deduplicate-binary-clauses-inv-def case-prod-beta*
  apply (*intro ext iffI*)
  apply *normalize-goal+*
  apply *simp*
  apply *normalize-goal+*
  apply (*rule-tac x=x* **in** *exI*)
  apply *simp*
  done

definition *deduplicate-binary-clauses* :: ‹'*v literal* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*deduplicate-binary-clauses L S* = *do* {
    ASSERT (*deduplicate-binary-clauses-pre L S*);
    let CS = (λ-. *None*);
    xs ← *SPEC* (λ*CS*. ∀ *C*. (*C* ∈# *CS* ⟶ *C* ∈# *dom-m* (*get-clauses-l S*) ⟶ *L* ∈ *set* (*get-clauses-l S* ∝ *C*)) ∧ *distinct-mset CS*);
    (-, -, -, *S*) ← *WHILE*$_T$$^{deduplicate\text{-}binary\text{-}clauses\text{-}inv\ L\ xs\ S}$ (λ(*abort*, *xs*, *CS*, *S*). ¬*abort* ∧ *xs* ≠ {#} ∧ *get-conflict-l S* = *None*)
      (λ(*abort*, *xs*, *CS*, *S*).
      do {
        C ← *SPEC* (λ*C*. *C* ∈# *xs*);

```
        if C ∉# dom-m (get-clauses-l S) ∨ length (get-clauses-l S ∝ C) ≠ 2 then
          RETURN (abort, xs − {#C#}, CS, S)
        else do {
          L' ← SPEC (λL'. mset (get-clauses-l S ∝ C) = {#L, L'#});
          if defined-lit (get-trail-l S) L' then do {
            U ← simplify-clause-with-unit-st C S;
            ASSERT (cdcl-twl-inprocessing-l** S U);
            RETURN (defined-lit (get-trail-l U) L, xs − {#C#}, CS, U)
          }
          else if CS L' ≠ None then do {
             let C' = (if (¬snd (the (CS L')) ⟶ ¬irred (get-clauses-l S) C) then C else fst (the (CS
L')));
             let CS = (if (¬snd (the (CS L')) ⟶ ¬irred (get-clauses-l S) C) then CS else CS (L' :=
Some (C, irred (get-clauses-l S) C)));
            S ← clause-remove-duplicate-clause C' S;
            RETURN (abort, xs − {#C#}, CS, S)
          } else if CS (−L') ≠ None then do {
            S ← binary-clause-subres C L (−L') S;
            RETURN (True, xs − {#C#}, CS, S)
          } else do {
            RETURN (abort, xs − {#C#}, CS (L' := Some (C, irred (get-clauses-l S) C)), S)
          }
        }
      })
      (defined-lit (get-trail-l S) L, xs, CS, S);
    RETURN S
}›

lemma deduplicate-binary-clauses-correctly-marked-remove1:
  ‹deduplicate-binary-clauses-correctly-marked L xs0 xs CS T ⟹ distinct-mset xs0 ⟹ xs ⊆# xs0 ⟹
  deduplicate-binary-clauses-correctly-marked L xs0 (remove1-mset C xs) CS T›
  apply (cases ‹C ∈# xs›)
  apply (auto simp: deduplicate-binary-clauses-correctly-marked-def dest!: multi-member-split)
  apply (metis Multiset.diff-add add-mset-add-single in-diffD)+
  done

lemma fmdrop-eq-dom-m-iff: ‹C ∈# dom-m N ⟹ fmdrop C N = fmdrop C' N ⟷ C = C'›
  by (metis distinct-mset-add-mset distinct-mset-dom dom-m-fmdrop in-remove1-mset-neq insert-DiffM)

lemma deduplicate-binary-clauses-inv-cdcl-twl-unitres-l:
  ‹cdcl-twl-inprocessing-l bb xc ⟹ deduplicate-binary-clauses-pre L S ⟹
  deduplicate-binary-clauses-inv L x S (False, aa, ab, bb) ⟹
  xa ∈# aa ⟹
     xa ∈# dom-m (get-clauses-l xc) ⟹
     defined-lit (get-trail-l bb) xb ⟹
     ∀ C. C ∈# x ⟶ C ∈# dom-m (get-clauses-l S) ⟶ L ∈ set (get-clauses-l S ∝ C) ⟹
     distinct-mset x ⟹
     ¬ a ⟹
     aa ≠ {#} ⟹
     get-conflict-l bb = None ⟹
     set (get-all-mark-of-propagated (get-trail-l xc))
     ⊆ insert 0 (set (get-all-mark-of-propagated (get-trail-l bb))) ⟹
     ∀ L∈set (get-clauses-l xc ∝ xa). undefined-lit (get-trail-l xc) L ⟹
    dom-m (get-clauses-l bb) = dom-m (get-clauses-l xc) ⟹
   ∀ C'∈#dom-m (get-clauses-l xc). xa ≠ C' ⟶ fmlookup (get-clauses-l bb) C' = fmlookup (get-clauses-l
xc) C' ⟹
```

deduplicate-binary-clauses-inv L x S
(*defined-lit* (*get-trail-l xc*) *L*, *remove1-mset xa aa*, *ab*, *xc*)›

apply (*subst deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
apply (*subst* (*asm*)*deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
unfolding *prod.simps deduplicate-binary-clauses-pre-def*
apply *normalize-goal+*
apply (*auto dest*:)[]
apply (*clarsimp simp*: *deduplicate-binary-clauses-correctly-marked-def*)
apply (*intro conjI impI allI*)
apply (*auto dest*: *in-diffD simp*: *distinct-mset-in-diff*)
apply (*metis*)+
done

lemma *deduplicate-binary-clauses-inv-deleted*:
‹*deduplicate-binary-clauses-pre L S* ⟹
deduplicate-binary-clauses-inv L x S (*False*, *aa*, *ab*, *bb*) ⟹
s = (*False*, *aa*, *ab*, *bb*) ⟹
b = (*aa*, *ab*, *bb*) ⟹
ba = (*ab*, *bb*) ⟹
xa ∈# *aa* ⟹
xa ∈# *dom-m* (*get-clauses-l bb*) ⟹
mset (*get-clauses-l bb* ∝ *xa*) = {#*L*, *xb*#} ⟹
defined-lit (*get-trail-l bb*) *xb* ⟹
∀ *C*. *C* ∈# *x* ⟶ *C* ∈# *dom-m* (*get-clauses-l S*) ⟶ *L* ∈ *set* (*get-clauses-l S* ∝ *C*) ⟹
distinct-mset x ⟹
¬ *a* ⟹
aa ≠ {#} ⟹
get-conflict-l bb = *None* ⟹
set (*get-all-mark-of-propagated* (*get-trail-l xc*)) ⊆ *insert 0* (*set* (*get-all-mark-of-propagated* (*get-trail-l bb*))) ⟹
∀ *C'*∈#*dom-m* (*get-clauses-l xc*). *xa* ≠ *C'* ⟶ *fmlookup* (*get-clauses-l bb*) *C'* = *fmlookup* (*get-clauses-l xc*) *C'* ⟹
xa ∈# *dom-m* (*get-clauses-l xc*) ⟶ (∀ *L*∈*set* (*get-clauses-l xc* ∝ *xa*). *undefined-lit* (*get-trail-l xc*) *L*) ⟹
remove1-mset xa (*dom-m* (*get-clauses-l bb*)) = *dom-m* (*get-clauses-l xc*) ⟹
cdcl-twl-inprocessing-l bb xc ⟹
deduplicate-binary-clauses-inv L x S (*defined-lit* (*get-trail-l xc*) *L*, *remove1-mset xa aa*, *ab*, *xc*)›

apply (*subst deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
apply (*subst* (*asm*)*deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
unfolding *prod.simps deduplicate-binary-clauses-pre-def*
apply *normalize-goal+*
apply (*auto dest*:)[]
apply (*clarsimp simp*: *deduplicate-binary-clauses-correctly-marked-def*)
apply (*intro conjI impI allI*)
apply (*auto dest*: *in-diffD simp*: *distinct-mset-in-diff*)
apply (*metis in-dom-m-lookup-iff in-remove1-msetI*)
apply (*metis in-dom-m-lookup-iff insert-DiffM insert-iff set-mset-add-mset-insert*)
apply (*metis in-dom-m-lookup-iff insert-DiffM insert-iff set-mset-add-mset-insert*)
apply (*metis in-dom-m-lookup-iff insert-DiffM insert-iff set-mset-add-mset-insert*)
apply (*clarsimp simp*: *deduplicate-binary-clauses-correctly-marked-def*)
apply (*intro conjI impI allI*)
apply (*auto dest*: *in-diffD simp*: *distinct-mset-in-diff*)
apply (*metis in-dom-m-lookup-iff in-remove1-msetI*)
apply (*metis in-dom-m-lookup-iff insert-DiffM insert-iff set-mset-add-mset-insert*)

apply (*metis in-dom-m-lookup-iff insert-DiffM insert-iff set-mset-add-mset-insert*)
apply (*metis in-dom-m-lookup-iff insert-DiffM insert-iff set-mset-add-mset-insert*)
done

lemma *deduplicate-binary-clauses*:
assumes ‹*deduplicate-binary-clauses-pre L S*›
shows ‹*deduplicate-binary-clauses L S* ≤ *SPEC*(*cdcl-twl-inprocessing-l*** *S*)›
proof −
have *clause-remove-duplicate-clause-pre*:
‹*clause-remove-duplicate-clause-pre* (*if* ¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa then xa*
else fst (*the* (*ab xb*))) *bb*›
if
pre: ‹*deduplicate-binary-clauses-pre L S*› **and**
x-spec: ‹∀ *C*. (*C* ∈# *x* ⟶ *C* ∈# *dom-m* (*get-clauses-l S*) ⟶ *L* ∈ *set* (*get-clauses-l S* ∝ *C*)) ∧
distinct-mset x› **and**
inv: ‹*deduplicate-binary-clauses-inv L x S s*› **and**
abort: ‹*case s of* (*abort*, *xs*, *CS*, *S*) ⇒ ¬ *abort* ∧ *xs* ≠ {#} ∧ *get-conflict-l S* = *None*› **and**
st: ‹*s* = (*a*, *b*)›
‹*b* = (*aa*, *ba*)›
‹*ba* = (*ab*, *bb*)› **and**
aa: ‹*xa* ∈# *aa*› **and**
xa: ‹¬ (*xa* ∉# *dom-m* (*get-clauses-l bb*) ∨ *length* (*get-clauses-l bb* ∝ *xa*) ≠ *2*)› **and**
xa-L: ‹*mset* (*get-clauses-l bb* ∝ *xa*) = {#*L*, *xb*#}› **and**
‹*undefined-lit* (*get-trail-l bb*) *xb*› **and**
ab: ‹*ab xb* ≠ *None*›
for *x s a b aa ba ab bb xa xb*
proof −
let *?C* = ‹(*if* ¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa then xa*
else fst (*the* (*ab xb*)))›
let *?C′* = ‹(*if* ¬(¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa*) *then xa*
else fst (*the* (*ab xb*)))›
obtain *T′* **where**
T′: ‹(*bb*, *T′*) ∈ *twl-st-l None* ∧ *twl-struct-invs T′* ∧ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init*
(*state*$_W$*-of T′*)›
‹*twl-list-invs bb*›
using *inv pre xa abort* **apply** −
unfolding *clause-remove-duplicate-clause-pre-def st prod.simps deduplicate-binary-clauses-inv-alt-def*
by *blast*
have *H*: ‹*ab xb* = *Some* (*C*, *b*) ⟹ *C* ≠ *0* ∧ *C* ∉ *set* (*get-all-mark-of-propagated* (*get-trail-l bb*)) ∧
C ∈# *dom-m* (*get-clauses-l bb*) ∧
C ∈# *x* − *aa* ∧ *mset* (*get-clauses-l bb* ∝ *C*) = {#*L*, *xb*#} ∧ *irred* (*get-clauses-l bb*) *C* = *b*›
‹*clauses-to-update-l bb* = {#}›
‹*xa* ∉ *set* (*get-all-mark-of-propagated* (*get-trail-l bb*))› **for** *C b*
using *inv pre xa abort T′* **apply** −
unfolding *clause-remove-duplicate-clause-pre-def st prod.simps*
apply (*clarsimp-all simp*: *deduplicate-binary-clauses-inv-alt-def deduplicate-binary-clauses-correctly-marked-def*
twl-list-invs-def)
apply (*intro conjI impI*)
apply (*metis neq0-conv*)
apply (*metis empty-iff in-mono insert-iff*)
by (*metis empty-iff singletonD subset-singletonD*)

show *?thesis*
using *pre H ab xa xa-L x-spec aa T′* **apply** −
unfolding *clause-remove-duplicate-clause-pre-def st*
apply (*rule exI*[*of* - ‹*?C′*›])

apply (*rule exI*[*of* - ‹*T'*›])
apply (*auto simp*: *deduplicate-binary-clauses-inv-alt-def deduplicate-binary-clauses-correctly-marked-def*)
apply (*meson distinct-mset-in-diff*)+
done
qed
have *clause-remove-duplicate-clause-post*:
‹*deduplicate-binary-clauses-inv L x S* (*a*, *remove1-mset xa aa*, *if* ¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa then ab*
else ab(*xb* ↦ (*xa*, *irred* (*get-clauses-l bb*) *xa*)), *xc*)›
if
‹*deduplicate-binary-clauses-pre L S*› **and**
‹∀ *C*. (*C* ∈# *x* ⟶ *C* ∈# *dom-m* (*get-clauses-l S*) ⟶ *L* ∈ *set* (*get-clauses-l S* ∝ *C*)) ∧ *distinct-mset x*› **and**
‹*deduplicate-binary-clauses-inv L x S s*› **and**
‹*case s of* (*abort*, *xs*, *CS*, *S*) ⇒ ¬ *abort* ∧ *xs* ≠ {#} ∧ *get-conflict-l S* = *None*› **and**
st: ‹*s* = (*a*, *b*)›
‹*b* = (*aa*, *ba*)›
‹*ba* = (*ab*, *bb*)› **and**
‹*xa* ∈# *aa*› **and**
‹¬ (*xa* ∉# *dom-m* (*get-clauses-l bb*) ∨ *length* (*get-clauses-l bb* ∝ *xa*) ≠ *2*)› **and**
‹*mset* (*get-clauses-l bb* ∝ *xa*) = {#*L*, *xb*#}› **and**
‹*undefined-lit* (*get-trail-l bb*) *xb*› **and**
‹*ab xb* ≠ *None*› **and**
‹*clause-remove-duplicate-clause-spec* (*if* ¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa then xa*
else fst (*the* (*ab xb*))) *bb xc*›
for *x s a b aa ba ab bb xa xb xc*
proof −
let *?C* = ‹(*if* ¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa then xa*
else fst (*the* (*ab xb*)))›
let *?C'* = ‹(*if* ¬(¬ *snd* (*the* (*ab xb*)) ⟶ ¬ *irred* (*get-clauses-l bb*) *xa*) *then xa*
else fst (*the* (*ab xb*)))›
show *?thesis*
using *that*
apply (*subst deduplicate-binary-clauses-inv-alt-def2*)
apply *assumption*
apply (*subst* (*asm*)*deduplicate-binary-clauses-inv-alt-def*)
unfolding *st prod.simps clause-remove-duplicate-clause-spec-def*
apply *normalize-goal*+
apply (*intro conjI*)
apply (*auto simp add*: *deduplicate-binary-clauses-correctly-marked-def*
distinct-mset-remove1-All distinct-mset-dom distinct-mset-in-diff dest: *in-diffD*)
apply (*metis in-multiset-nempty member-add-mset*)
by (*meson Duplicate-Free-Multiset.distinct-mset-mono distinct-mem-diff-mset union-single-eq-member*)
qed
have *binary-clause-subres-lits-pre*: ‹*binary-clause-subres-lits-pre xa L* (−*xb*) *bb*›
if
pre: ‹*deduplicate-binary-clauses-pre L S*› **and**
‹∀ *C*. (*C* ∈# *x* ⟶ *C* ∈# *dom-m* (*get-clauses-l S*) ⟶ *L* ∈ *set* (*get-clauses-l S* ∝ *C*)) ∧ *distinct-mset x*› **and**
inv: ‹*deduplicate-binary-clauses-inv L x S s*› **and**
abort: ‹*case s of* (*abort*, *xs*, *CS*, *S*) ⇒ ¬ *abort* ∧ *xs* ≠ {#} ∧ *get-conflict-l S* = *None*› **and**
st: ‹*s* = (*a*, *b*)›
‹*b* = (*aa*, *ba*)›
‹*ba* = (*ab*, *bb*)› **and**
‹*xa* ∈# *aa*› **and**

‹¬ (xa ∉# dom-m (get-clauses-l bb)∨ length (get-clauses-l bb ∝ xa) ≠ 2)› **and**
‹mset (get-clauses-l bb ∝ xa) = {#L, xb#}› **and**
‹undefined-lit (get-trail-l bb) xb› **and**
‹¬ ab xb ≠ None› **and**
xb: ‹ab (− xb) ≠ None›
for x s a b aa ba ab bb xa xb
proof −
obtain T′ **where**
T′: ‹(bb, T′) ∈ twl-st-l None› ‹twl-struct-invs T′›‹cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (state$_W$-of T′)›
using inv pre xb abort **apply** −
unfolding clause-remove-duplicate-clause-pre-def st prod.simps deduplicate-binary-clauses-inv-alt-def
apply normalize-goal+
by blast
have H: ‹ab (− xb) = Some (C, b) ⟹ C ≠ 0 ∧ C ∉ set (get-all-mark-of-propagated (get-trail-l bb)) ∧
C ∈# dom-m (get-clauses-l bb) ∧
C ∈# x − aa ∧ mset (get-clauses-l bb ∝ C) = {#L, (− xb)#} ∧ irred (get-clauses-l bb) C = b›
‹clauses-to-update-l bb = {#}›
‹xa ∉ set (get-all-mark-of-propagated (get-trail-l bb))› **for** C b
using inv pre xb abort **apply** −
unfolding clause-remove-duplicate-clause-pre-def st prod.simps deduplicate-binary-clauses-inv-alt-def
apply normalize-goal+
apply (clarsimp-all simp: deduplicate-binary-clauses-inv-alt-def deduplicate-binary-clauses-correctly-marked-def
twl-list-invs-def)
apply (intro conjI)
apply (metis neq0-conv)
apply (metis empty-iff in-mono insert-iff)
by (metis in-mono singletonD that(9))
show ?thesis
using that H T′ **apply** −
unfolding binary-clause-subres-lits-pre-def
by (rule exI[of - ‹fst (the (ab (−xb)))›], rule exI[of - T′])
(auto simp: deduplicate-binary-clauses-inv-alt-def deduplicate-binary-clauses-correctly-marked-def)
qed
have new-lit: ‹deduplicate-binary-clauses-inv L x S
(a, remove1-mset xa aa, ab(xb ↦ (xa, irred (get-clauses-l bb) xa)), bb)›
if
‹deduplicate-binary-clauses-pre L S› **and**
‹∀ C. (C ∈# x ⟶ C ∈# dom-m (get-clauses-l S) ⟶ L ∈ set (get-clauses-l S ∝ C)) ∧ distinct-mset x› **and**
‹deduplicate-binary-clauses-inv L x S s› **and**
‹case s of
(abort, xs, CS, S) ⇒ ¬ abort ∧ xs ≠ {#} ∧ get-conflict-l S = None› **and**
st: ‹s = (a, b)›
‹b = (aa, ba)›
‹ba = (ab, bb)› **and**
‹xa ∈# aa› **and**
‹¬ (xa ∉# dom-m (get-clauses-l bb) ∨ length (get-clauses-l bb ∝ xa) ≠ 2)› **and**
‹mset (get-clauses-l bb ∝ xa) = {#L, xb#}› **and**
‹undefined-lit (get-trail-l bb) xb› **and**
‹¬ (ab xb ≠ None)› **and**
‹¬ ab (− xb) ≠ None›
for x s a b aa ba ab bb xa xb
proof −
show ?thesis

using *that* **unfolding** *prod.simps st* **apply** −
apply (*subst deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
apply (*subst* (*asm*) *deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
apply *normalize-goal*+
apply (*auto simp*: *deduplicate-binary-clauses-correctly-marked-def distinct-mset-in-diff*
dest!: *multi-member-split dest*: *distinct-mset-mono*)
apply (*meson distinct-mset-add-mset distinct-mset-mono member-add-mset*)
done
qed
let *?R* = ‹*measure* (λ(*abort*, *xs*, *CS*, *S*). *size xs*)›
show *?thesis*
unfolding *deduplicate-binary-clauses-def Let-def*
apply (*refine-vcg assms WHILEIT-rule*[**where** *R* = *?R*] *simplify-clause-with-unit-st-spec*[*THEN order-trans*]
clause-remove-duplicate-clause[*THEN order-trans*] *binary-clause-subres-binary-clause*[*THEN order-trans*])
subgoal
by *auto*
subgoal
by (*fastforce simp*: *deduplicate-binary-clauses-inv-def deduplicate-binary-clauses-pre-def deduplicate-binary-clauses-correctly-marked-def*)
subgoal
by (*subst deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
(*auto simp*: *deduplicate-binary-clauses-inv-def deduplicate-binary-clauses-correctly-marked-remove1*)
subgoal
by (*auto dest*: *multi-member-split*)
subgoal
unfolding *deduplicate-binary-clauses-inv-alt-def simplify-clause-with-unit-st-pre-def*
case-prod-beta ex-simps(*2*)[*symmetric*]
apply *normalize-goal*+
by (*rule-tac x*=*xd* **in** *exI*) *simp*
subgoal
apply *simp*
apply *standard*
apply *simp*
apply *normalize-goal*+
apply (*intro ASSERT-leI*)
apply (*metis cdcl-twl-inprocessing-l.intros*(*1*,*2*) *r-into-rtranclp rtranclp.rtrancl-refl*)
apply *simp*
apply (*elim disjE*; *intro conjI*; *subst* (*asm*) *eq-commute*[*of* ‹*dom-m* -›])
apply (*subst deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
apply (*subst* (*asm*)*deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
unfolding *prod.simps*
apply (*intro conjI*)
apply (*simp-all add*: *deduplicate-binary-clauses-correctly-marked-remove1 size-mset-remove1-mset-le-iff*)
apply (*force dest*: *multi-member-split*)
apply (*rule deduplicate-binary-clauses-inv-cdcl-twl-unitres-l*)
apply (*rule cdcl-twl-inprocessing-l.intros*; *assumption*)
apply *assumption*+
apply (*rule deduplicate-binary-clauses-inv-cdcl-twl-unitres-l*)
apply (*rule cdcl-twl-inprocessing-l.intros*; *assumption*)
apply *assumption*+
apply (*rule deduplicate-binary-clauses-inv-deleted*)
apply *assumption*+
apply (*rule cdcl-twl-inprocessing-l.intros*; *assumption*)
apply (*rule deduplicate-binary-clauses-inv-deleted*)

apply *assumption+*
apply (*rule cdcl-twl-inprocessing-l.intros*; *assumption*)
done
subgoal by (*rule clause-remove-duplicate-clause-pre*)
subgoal by (*rule clause-remove-duplicate-clause-post*)
subgoal
by (*auto dest*!: *multi-member-split*)
subgoal by (*rule binary-clause-subres-lits-pre*)
subgoal
by (*subst deduplicate-binary-clauses-inv-alt-def2*, *assumption*,
subst (*asm*)*deduplicate-binary-clauses-inv-alt-def2*, *assumption*)
(*auto simp*:)
subgoal by (*auto dest*!: *multi-member-split*)
subgoal by (*rule new-lit*)
subgoal by (*auto dest*!: *multi-member-split*)
subgoal unfolding *deduplicate-binary-clauses-inv-def* **by** *fast*
done
qed

definition *mark-duplicated-binary-clauses-as-garbage-pre* :: ‹*'v twl-st-l* ⇒ *bool*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-pre* = (λ*S*. (∃ *T*. (*S*,*T*) ∈ *twl-st-l None* ∧ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*) ∧ *count-decided* (*get-trail-l S*) = *0* ∧
clauses-to-update-l S = {#} ∧ *twl-list-invs S* ∧ *twl-struct-invs T*))›

definition *mark-duplicated-binary-clauses-as-garbage-inv* :: ‹*'v multiset* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l* × *'v multiset* ⇒ *bool*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-inv* = (λ*xs S* (*U*, *ys*). *ys* ⊆# *xs* ∧ *cdcl-twl-inprocessing-l*** *S U*)›

lemma *mark-duplicated-binary-clauses-as-garbage-inv-alt-def*:
assumes ‹*mark-duplicated-binary-clauses-as-garbage-pre S*›
shows ‹*mark-duplicated-binary-clauses-as-garbage-inv xs S* (*U*, *Ls*) ⟷
(∃ *T*. (*U*,*T*) ∈ *twl-st-l None* ∧ *set* (*get-all-mark-of-propagated* (*get-trail-l U*)) ⊆ {*0*} ∧
Ls ⊆# *xs* ∧*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*) ∧ *count-decided* (*get-trail-l U*) = *0* ∧
clauses-to-update-l U = {#} ∧ *twl-list-invs U* ∧ *twl-struct-invs T* ∧ *cdcl-twl-inprocessing-l*** *S U*)›
using *assms rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated*[*of S U*] *rtranclp-cdcl-twl-inprocessing-l-clauses-to*
S U]
rtranclp-cdcl-twl-inprocessing-l-count-decided[*of S U*] *rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated*[*of S U*]
unfolding *mark-duplicated-binary-clauses-as-garbage-pre-def*
mark-duplicated-binary-clauses-as-garbage-inv-def prod.simps **apply** −
apply *normalize-goal+*
apply (*rule iffI*)
apply *normalize-goal+*
apply (*rule rtranclp-cdcl-twl-inprocessing-l-twl-st-l*; *assumption?*)
apply (*rename-tac x V*; *rule-tac x*=*V* **in** *exI*)
apply (*solves auto*)
by *meson*

definition *mark-duplicated-binary-clauses-as-garbage* :: ‹- ⇒ *'v twl-st-l nres*› **where**
‹*mark-duplicated-binary-clauses-as-garbage S* = *do* {
ASSERT (*mark-duplicated-binary-clauses-as-garbage-pre S*);
Ls ← *SPEC* (λ*Ls*:: *'v multiset*. ∀ *L*∈#*Ls*. *L* ∈# *atm-of* ‘# *all-init-lits-of-l S*);

```
  (S, -) ← WHILE_T^{mark-duplicated-binary-clauses-as-garbage-inv Ls S}(λ(S, Ls). Ls ≠ {#} ∧ get-conflict-l
S = None)
    (λ(S, Ls). do {
      L ← SPEC (λL. L ∈# Ls);
      ASSERT (L ∈# atm-of `# all-init-lits-of-l S);
      skip ← RES (UNIV :: bool set);
      if skip then RETURN (S, remove1-mset L Ls)
      else do {
        S ← deduplicate-binary-clauses (Pos L) S;
        S ← deduplicate-binary-clauses (Neg L) S;
        RETURN (S, remove1-mset L Ls)
      }
    })
    (S, Ls);
   RETURN S
 }›

lemma cdcl-twl-inprocessing-l-all-learned-lits-of-l:
 assumes ‹cdcl-twl-inprocessing-l S T›
 shows ‹set-mset (all-learned-lits-of-l T) ⊆ set-mset (all-learned-lits-of-l S)›
proof −
 have [simp]: ‹D ∉# A ⟹ {#the (if D = x then b else fmlookup N x). x ∈# A#} =
   {#the (fmlookup N x). x ∈# A#}›
   ‹D ∉# A ⟹ {#the (if x = D then b else fmlookup N x). x ∈# A#} =
   {#the (fmlookup N x). x ∈# A#}› for D E N A b
   by (auto intro!: image-mset-cong)
 have dups-uniq[dest]: ‹remdups-mset D′ = {#K#} ⟹ set-mset (all-lits-of-m D′) = {−K,K}› for
D′ K
   by (metis all-lits-of-m-add-mset all-lits-of-m-empty all-lits-of-m-remdups-mset
     insert-commute set-mset-add-mset-insert set-mset-empty)
 show ?thesis
   using assms
   using distinct-mset-dom[of ‹get-clauses-l S›] apply −
   supply [[goals-limit=1]]
   by (induction rule: cdcl-twl-inprocessing-l.induct)
    (auto 4 3 simp: cdcl-twl-unitres-l.simps
     cdcl-twl-unitres-true-l.simps
     cdcl-twl-subsumed-l.simps
     cdcl-twl-subresolution-l.simps cdcl-twl-pure-remove-l.simps
     all-learned-lits-of-l-def
     add-mset-eq-add-mset removeAll-notin
     get-init-clss-l-def init-clss-l-mapsto-upd-notin
     init-clss-l-mapsto-upd ran-m-def all-lits-of-m-union
     all-lits-of-m-add-mset distinct-mset-remove1-All
     init-clss-l-fmupd-if all-lits-of-mm-add-mset
     all-lits-of-m-remdups-mset
     dest!: multi-member-split[of ‹- :: nat› ‹-›]
     dest: all-lits-of-m-mono)
qed

lemma rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l:
 assumes ‹cdcl-twl-inprocessing-l** S T›
 shows ‹set-mset (all-learned-lits-of-l T) ⊆ set-mset (all-learned-lits-of-l S)›
 using assms
 by (induction rule: rtranclp-induct) (auto dest: cdcl-twl-inprocessing-l-all-learned-lits-of-l)
```

lemma *mark-duplicated-binary-clauses-as-garbage*:
fixes S :: ‹'v twl-st-l›
assumes *pre*: ‹*mark-duplicated-binary-clauses-as-garbage-pre S*›
shows ‹*mark-duplicated-binary-clauses-as-garbage S* ≤ ⇓ *Id* (*SPEC*(*cdcl-twl-inprocessing-l*** *S*))›
proof −
have *deduplicate-binary-clauses-pre*: ‹*deduplicate-binary-clauses-pre* (*Pos xa*) *xb*› (**is** *?A*)
‹*deduplicate-binary-clauses-pre* (*Neg xa*) *xb*› (**is** *?B*)
if
‹*mark-duplicated-binary-clauses-as-garbage-inv xs S s*› **and**
‹*case s of* (*S*, *Ls*) ⇒ *Ls* ≠ {#} ∧ *get-conflict-l S* = *None*› **and**
st: ‹*s* = (*a*, *b*)› **and**
‹*cdcl-twl-inprocessing-l*** *a xb*› **and**
xs: ‹∀ *L*∈#*xs*. *L* ∈# *atm-of* '# *all-init-lits-of-l S*›
‹*xa* ∈# *b*›
for *s a xa xb* **and** *xs b* :: ‹'v multiset›
proof −
have ‹*mark-duplicated-binary-clauses-as-garbage-inv xs S* (*xb*, *b*)› ‹*xa* ∈# *xs*›
using *that* **unfolding** *mark-duplicated-binary-clauses-as-garbage-inv-def*
by *auto*
moreover have ‹*set-mset* (*all-init-lits-of-l xb*) = *set-mset* (*all-init-lits-of-l S*)›
using *that* **by** (*auto dest*!: *rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*
simp: *mark-duplicated-binary-clauses-as-garbage-inv-def*)
then have ‹*Pos xa* ∈# *all-init-lits-of-l xb*› ‹*Neg xa* ∈# *all-init-lits-of-l xb*›
using ‹*xa* ∈# *xs*› *xs* **by** (*auto dest*!: *multi-member-split*[*of xa xs*]
simp: *all-init-lits-of-l-def*)
(*metis in-all-lits-of-mm-ain-atms-of-iff literal.sel*)+
ultimately show *?A ?B*
using *pre* **unfolding** *deduplicate-binary-clauses-pre-def st*
by (*meson mark-duplicated-binary-clauses-as-garbage-inv-alt-def*)+
qed
let *?R* = ‹*measure* (λ(-, *Ls*). *size Ls*)›
have *wf*: ‹*wf ?R*›
by *auto*
show *?thesis*
unfolding *mark-duplicated-binary-clauses-as-garbage-def*
apply (*refine-vcg deduplicate-binary-clauses*[*THEN order-trans*] *wf assms*)
subgoal by (*auto simp*: *mark-duplicated-binary-clauses-as-garbage-inv-def*)
subgoal by (*auto simp*: *mark-duplicated-binary-clauses-as-garbage-inv-def*
dest!: *rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*)
subgoal by (*auto simp*: *mark-duplicated-binary-clauses-as-garbage-inv-def*)
subgoal by (*auto dest*!: *multi-member-split*)
subgoal unfolding *deduplicate-binary-clauses-pre-def*
by *hypsubst*
(*metis deduplicate-binary-clauses-pre deduplicate-binary-clauses-pre-def rtranclp.rtrancl-refl*)
subgoal by (*rule deduplicate-binary-clauses-pre*)
subgoal by (*auto simp*: *mark-duplicated-binary-clauses-as-garbage-inv-def*)
subgoal by (*auto dest*!: *multi-member-split*)
subgoal by (*auto simp*: *mark-duplicated-binary-clauses-as-garbage-inv-def*)
done
qed

definition *forward-subsumption-one-inv* :: ‹*nat* ⇒ 'v *twl-st-l* ⇒ *nat multiset* ⇒ - ⇒ *bool*› **where**
‹*forward-subsumption-one-inv* = (λ*C S zs* (*xs*, *s*).
(∃ *S'* . (*S*, *S'*) ∈ *twl-st-l None* ∧
twl-struct-invs S' ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S'*) ∧

$C \notin\# xs \wedge C \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S) \wedge count\text{-}decided\ (get\text{-}trail\text{-}l\ S) = 0 \wedge$
$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\} \wedge twl\text{-}list\text{-}invs\ S \wedge twl\text{-}struct\text{-}invs\ S' \wedge$
$set\ (get\text{-}all\text{-}mark\text{-}of\text{-}propagated\ (get\text{-}trail\text{-}l\ S)) \subseteq \{0\} \wedge$
$(\forall C' \in\# xs.\ C' \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S) \longrightarrow length\ (get\text{-}clauses\text{-}l\ S \propto C') \leq length\ (get\text{-}clauses\text{-}l\ S \propto C)) \wedge$
$cdcl_W\text{-}restart\text{-}mset.cdcl_W\text{-}learned\text{-}clauses\text{-}entailed\text{-}by\text{-}init\ (state_W\text{-}of\ S') \wedge$
$subsume\text{-}or\text{-}strengthen\text{-}pre\ C\ s\ S \wedge xs \subseteq\# zs \wedge$
$(is\text{-}subsumed\ s \longrightarrow subsumed\text{-}by\ s \in\# zs) \wedge$
$(is\text{-}strengthened\ s \longrightarrow strengthened\text{-}by\ s \in\# zs))\rangle$

definition *forward-subsumption-one-select* **where**
$\langle forward\text{-}subsumption\text{-}one\text{-}select\ C\ ys\ S = (\lambda xs.\ C \notin\# xs \wedge ys \cap\# xs = \{\#\} \wedge$
$(\forall D \in\# xs.\ D \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S) \longrightarrow$
$(\forall L \in set\ (get\text{-}clauses\text{-}l\ S \propto D).\ undefined\text{-}lit\ (get\text{-}trail\text{-}l\ S)\ L) \wedge$
$(length\ (get\text{-}clauses\text{-}l\ S \propto D) \leq length\ (get\text{-}clauses\text{-}l\ S \propto C))))\rangle$

definition *forward-subsumption-one* :: $\langle nat \Rightarrow nat\ multiset \Rightarrow 'v\ twl\text{-}st\text{-}l \Rightarrow ('v\ twl\text{-}st\text{-}l \times bool)\ nres\rangle$ **where**
$\langle forward\text{-}subsumption\text{-}one = (\lambda C\ cands\ S_0.\ do\ \{$
$ASSERT(forward\text{-}subsumption\text{-}one\text{-}pre\ C\ cands\ S_0);$
$xs_0 \leftarrow SPEC\ (forward\text{-}subsumption\text{-}one\text{-}select\ C\ cands\ S_0);$
$(xs,\ s) \leftarrow$
$WHILE_T^{\ forward\text{-}subsumption\text{-}one\text{-}inv\ C\ S_0\ xs_0}\ (\lambda(xs,\ s).\ xs \neq \{\#\} \wedge s = NONE)$
$(\lambda(xs,\ s).\ do\ \{$
$C' \leftarrow SPEC(\lambda C'.\ C' \in\# xs);$
$if\ C' \notin\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S_0)$
$then\ RETURN\ (remove1\text{-}mset\ C'\ xs,\ s)$
$else\ do\ \{$
$s \leftarrow SPEC(try\text{-}to\text{-}subsume\ C\ C'\ (get\text{-}clauses\text{-}l\ S_0));$
$RETURN\ (remove1\text{-}mset\ C'\ xs,\ s)$
$\}$
$\})$
$(xs_0,\ NONE);$
$ASSERT\ (forward\text{-}subsumption\text{-}one\text{-}inv\ C\ S_0\ xs_0\ (xs,\ s));$
$S \leftarrow subsume\text{-}or\text{-}strengthen\ C\ s\ S_0;$
$ASSERT\ (set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ S) \subseteq set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ S_0) \wedge set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ S) = set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ S_0));$
$RETURN\ (S,\ s \neq NONE)$
$\}$
$)\rangle$

lemma *forward-subsumption-one*:
assumes $\langle forward\text{-}subsumption\text{-}one\text{-}pre\ C\ cands\ S\rangle$
shows $\langle forward\text{-}subsumption\text{-}one\ C\ cands\ S \leq \Downarrow\{((st,\ SR),\ st').\ st = st' \wedge (\neg SR \longrightarrow st = S)\}$
$(SPEC(\lambda T.\ cdcl\text{-}twl\text{-}inprocessing\text{-}l^{**}\ S\ T \wedge get\text{-}trail\text{-}l\ T = get\text{-}trail\text{-}l\ S \wedge$
$(\forall D \in\# cands.\ D \neq C \longrightarrow (D \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ T)) = (D \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S))) \wedge$
$(\forall D \in\# remove1\text{-}mset\ C\ (dom\text{-}m\ (get\text{-}clauses\text{-}l\ T)).$
$D \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S) \wedge get\text{-}clauses\text{-}l\ T \propto D = get\text{-}clauses\text{-}l\ S \propto D)))\rangle$
proof −
obtain x **where**
Sx: $\langle (S,\ x) \in twl\text{-}st\text{-}l\ None\rangle$ **and**
$struct$: $\langle twl\text{-}struct\text{-}invs\ x\rangle$ **and**
$st\text{-}inv$: $\langle twl\text{-}st\text{-}inv\ x\rangle$ **and**
$list\text{-}invs$: $\langle twl\text{-}list\text{-}invs\ S\rangle$ **and**
C: $\langle C \in\# dom\text{-}m\ (get\text{-}clauses\text{-}l\ S)\rangle$ **and**
$all\text{-}struct\text{-}x$: $\langle cdcl_W\text{-}restart\text{-}mset.cdcl_W\text{-}all\text{-}struct\text{-}inv\ (state_W\text{-}of\ x)\rangle$ **and**

clss: ‹*clauses-to-update-l S = {#}*› ‹*get-conflict-l S = None*› **and**
no-annot: ‹*set (get-all-mark-of-propagated (get-trail-l S)) ⊆ {0}*› **and**
init: ‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init (state$_W$-of x)*› **and**
dec: ‹*count-decided (get-trail-l S) = 0*› **and**
undef-C: ‹*∀ L∈#mset (get-clauses-l S ∝ C). undefined-lit (get-trail-l S) L*› **and**
le-C: ‹*2 < length (get-clauses-l S ∝ C)*›
using *assms* **unfolding** *forward-subsumption-one-pre-def twl-struct-invs-def*
pcdcl-all-struct-invs-def state$_W$-of-def
apply −
by *normalize-goal+ blast* **have** ‹*C ≠ 0*›
using *list-invs C* **by** (*auto simp*: *twl-list-invs-def*)
then have *C-annot*: ‹*C ∉ set (get-all-mark-of-propagated (get-trail-l S))*›
using *no-annot* **by** *blast*

have [*refine0*]: ‹*wf (measure (λ(d, -). size d))*›
by *auto*
have *H*: ‹*length (get-clauses-l S ∝ D) ≥2*› ‹*distinct (get-clauses-l S ∝ D)*› ‹*D ≠ 0*›
‹*D ∉ set (get-all-mark-of-propagated (get-trail-l S))*›
‹*¬tautology (mset (get-clauses-l S ∝ D))*›
if ‹*D ∈# dom-m (get-clauses-l S)*›**for** *D*
using *st-inv that Sx list-invs no-annot* **unfolding** *twl-struct-invs-def twl-list-invs-def* **apply** −
by (*auto simp*: *twl-struct-invs-def twl-st-l-def ran-m-def conj-disj-distribR*
twl-st-inv.simps Collect-disj-eq image-image image-Un Collect-conv-if mset-take-mset-drop-mset′
dest!: *multi-member-split split*: *if-splits*
simp flip: *insert-compr*)

have *H2*: ‹*¬tautology (mset (get-clauses-l S ∝ D))*› **if** ‹*D ∈# dom-m (get-clauses-l S)*› **for** *D*
using *list-invs that* **by** (*auto simp*: *twl-list-invs-def ran-m-def*)

have *forward-subsumption-one-inv*: ‹*forward-subsumption-one-inv C S xs$_0$ (remove1-mset xa aa, xb)*›
if
select: ‹*forward-subsumption-one-select C cands S xs*› **and**
inv: ‹*forward-subsumption-one-inv C S xs$_0$ s*› **and**
‹*case s of (xs, s) ⇒ xs ≠ {#} ∧ s = NONE*› **and**
st[*simp*]: ‹*s = (aa, ba)*› **and**
xa-aa: ‹*xa ∈# aa*› **and**
xa: ‹*¬ xa ∉# dom-m (get-clauses-l S)*› **and**
subs: ‹*try-to-subsume C xa (get-clauses-l S) xb*›
for *xs s b aa ba xa xb xs$_0$*
proof −
have [*simp*]: ‹*C ≠ xa*›
using *xa-aa select inv* **unfolding** *forward-subsumption-one-select-def forward-subsumption-one-inv-def*
by *auto*

have [*intro*]: ‹*x21 ∈ set (get-clauses-l S ∝ xa) ⟹*
− x21 ∈ set (get-clauses-l S ∝ C) ⟹
remove1-mset x21 (mset (get-clauses-l S ∝ xa)) ⊆# remove1-mset (− x21) (mset (get-clauses-l S ∝ C)) ⟹
tautology (mset (get-clauses-l S ∝ xa) + mset (get-clauses-l S ∝ C) − {#− x21, x21#}) ⟹ False›
for *a x21*
using *multi-member-split*[*of x21* ‹*mset (get-clauses-l S ∝ xa)*›]
multi-member-split[*of* ‹*−x21*› ‹*mset (get-clauses-l S ∝ C)*›] *H*[*of C*] *C*
by (*auto simp*: *tautology-add-subset tautology-add-mset*)

have ‹*subsume-or-strengthen-pre C xb S*›
using *undef-C le-C dec clss C-annot subs H xa C Sx struct init all-struct-x st-inv list-invs*

```
      H[of C] unfolding subsume-or-strengthen-pre-def
     apply −
     apply (rule exI[of - x])
     by (auto simp: try-to-subsume-def split: subsumption.splits intro!: exI[of - x])
   then show ?thesis
     using inv
     unfolding forward-subsumption-one-inv-def prod.simps st apply −
     apply normalize-goal+
     apply (rule exI[of - x])
     apply (use Sx struct st-inv init dec subs xa-aa in ‹auto dest!: in-diffD›)
      using subs xa-aa
     apply (case-tac xb; auto simp: try-to-subsume-def; fail)+
     done
 qed
 show ?thesis
   unfolding forward-subsumption-one-def conc-fun-RES
   apply (rewrite at ‹- ≤ ⌑› RES-SPEC-conv)
   apply (refine-vcg subsume-or-strengthen[unfolded Down-id-eq] if-rule)
   subgoal using assms by auto
   subgoal for ax
     unfolding forward-subsumption-one-inv-def prod.simps forward-subsumption-one-select-def
       subsume-or-strengthen-pre-def ex-simps[symmetric]
     apply (rule-tac x=x in exI)+
      using all-struct-x Sx H struct st-inv C init by (auto simp add: subsume-or-strengthen-pre-def
forward-subsumption-one-pre-def )
   subgoal for xs
     unfolding simplify-clause-with-unit-st-pre-def forward-subsumption-one-inv-def case-prod-beta
     apply normalize-goal+
     by (rule-tac x=x in exI)
      (auto dest: in-diffD)
   subgoal
     unfolding simplify-clause-with-unit-st-pre-def forward-subsumption-one-inv-def case-prod-beta
     apply normalize-goal+
     by  (auto dest: multi-member-split)
   subgoal by (rule forward-subsumption-one-inv)
   subgoal by (auto dest!: multi-member-split)
   subgoal by auto
   subgoal
     unfolding forward-subsumption-one-inv-def
     by fast
   subgoal using C by auto
   subgoal by (simp add: rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l)
   subgoal by (auto dest:  rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l)
   subgoal premises p for x s a b xa
     using p(1−7) le-C unfolding forward-subsumption-one-inv-def prod.simps p(5) apply −
     apply normalize-goal+
     apply standard
     apply (auto simp add: forward-subsumption-one-inv-def)[]
     apply simp
     apply (intro ballI)
     subgoal for xx D
     apply (subgoal-tac ‹(is-strengthened (b) ⟶ D ≠ strengthened-by (b)) ∧
       (is-subsumed (b) ⟶ D ≠ subsumed-by (b))›)
     apply (cases b)
     apply (simp add:)
     apply (simp add:)
```

```
    apply blast
    apply (cases b)
    apply (intro conjI)
    apply (auto simp add: forward-subsumption-one-select-def dest!: multi-member-split[of D])[3]
    apply (simp only: subsumption.disc simp-thms)
    done
  done
  done
qed


definition simplify-clauses-with-unit-st-pre where
  ‹simplify-clauses-with-unit-st-pre S ⟷ (∃ T.
  (S, T) ∈ twl-st-l None ∧
  twl-struct-invs T ∧
  twl-list-invs S ∧
  clauses-to-update-l S = {#} ∧
  cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of T) ∧
  count-decided (get-trail-l S) = 0 ∧
  set (get-all-mark-of-propagated (get-trail-l S)) ⊆ {0})›

definition simplify-clauses-with-unit-st-inv :: ‹'v twl-st-l ⇒ nat set ⇒ 'v twl-st-l ⇒ bool› where
  ‹simplify-clauses-with-unit-st-inv S_0 it S ⟷ (
    cdcl-twl-inprocessing-l** S_0 S ∧
  set (get-all-mark-of-propagated (get-trail-l S)) ⊆
    set (get-all-mark-of-propagated (get-trail-l S_0)) ∪ {0})›
```

Hard-coding the invariants was a lot faster than the alternative way, namely proving that the loop invariants still holds at the end of the loop. No this makes not sense, I know.

```
definition simplify-clauses-with-unit-st :: ‹'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹simplify-clauses-with-unit-st S =
  do {
  ASSERT(simplify-clauses-with-unit-st-pre S);
  xs ← SPEC(λxs. finite xs);
  T ← FOREACHci(λit T. simplify-clauses-with-unit-st-inv S it T)
  (xs)
  (λS. get-conflict-l S = None)
  (λi S. if i ∈# dom-m (get-clauses-l S) then simplify-clause-with-unit-st i S else RETURN S)
  S;
  ASSERT(set-mset (all-learned-lits-of-l T) ⊆ set-mset (all-learned-lits-of-l S));
  ASSERT(set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S));
  RETURN T
  }›

lemma simplify-clauses-with-unit-st-inv-simplify-clauses-with-unit-st-preD:
  assumes
    inv: ‹simplify-clauses-with-unit-st-inv S_0 it T› and
    pre: ‹simplify-clauses-with-unit-st-pre S_0›
  shows ‹simplify-clauses-with-unit-st-pre T›
proof −
  have st: ‹cdcl-twl-inprocessing-l** S_0 T›
    using inv unfolding simplify-clauses-with-unit-st-inv-def
    by blast
  obtain S where
    S_0S: ‹(S_0, S) ∈ twl-st-l None› and
    struct-S: ‹twl-struct-invs S› and
```

list-S: ‹*twl-list-invs* S_0› **and**
clss-upd-S: ‹*clauses-to-update-l* $S_0 = \{\#\}$› **and**
ent-S: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S*)› **and**
lvl-S: ‹*count-decided* (*get-trail-l* S_0) = *0*› **and**
empty: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l* S_0)) $\subseteq \{0\}$›
using *pre* **unfolding** *simplify-clauses-with-unit-st-pre-def*
by *blast*

obtain *U* **where**
TU: ‹(*T*, *U*) ∈ *twl-st-l None*› **and**
struct-T: ‹*twl-struct-invs U*› **and**
list-T: ‹*twl-list-invs T*› **and**
ent-T: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of U*)›
using *rtranclp-cdcl-twl-inprocessing-l-twl-st-l*[*OF st* S_0*S struct-S list-S ent-S*]
by *blast*
have
clss-upd-T: ‹*clauses-to-update-l T* = {#}› **and**
lvl-S: ‹*count-decided* (*get-trail-l T*) = *0*›
subgoal
using *st* **apply** (*subst* (*asm*) *rtranclp-unfold*)
apply (*elim disjE*)
subgoal using *clss-upd-S* **by** *auto*
subgoal
by (*subst* (*asm*) *tranclp-unfold-end*)
(*auto simp*: *cdcl-twl-inprocessing-l.simps cdcl-twl-pure-remove-l.simps*
cdcl-twl-unitres-l.simps cdcl-twl-unitres-true-l.simps
cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps)
done
subgoal
using *st* **apply** (*induction rule*: *rtranclp-induct*)
subgoal using *lvl-S* **by** *auto*
subgoal
by (*auto simp*: *cdcl-twl-inprocessing-l.simps cdcl-twl-pure-remove-l.simps*
cdcl-twl-unitres-l.simps cdcl-twl-unitres-true-l.simps
cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps)
done
done
have *empty*: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l T*)) $\subseteq \{0\}$›
using *st* **apply** (*induction rule*: *rtranclp-induct*)
subgoal using *empty* **by** *auto*
subgoal by (*auto simp*: *cdcl-twl-inprocessing-l.simps*
cdcl-twl-unitres-l.simps cdcl-twl-unitres-true-l.simps cdcl-twl-pure-remove-l.simps
cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps)
done
show *?thesis*
unfolding *simplify-clauses-with-unit-st-pre-def*
by (*rule exI*[*of - U*])
(*use TU struct-T list-T ent-T clss-upd-T lvl-S empty* **in** *auto*)
qed

lemma *simplify-clauses-with-unit-st-spec*:
assumes ‹*count-decided* (*get-trail-l S*) = *0*›
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}› **and**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
st-invs: ‹*twl-struct-invs T*› **and**

list-invs: ‹*twl-list-invs S*› **and**
empty: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*}› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*))›
shows ‹*simplify-clauses-with-unit-st S* ≤ ⇓*Id* (*SPEC*(λ*T*. *cdcl-twl-inprocessing-l*** *S T* ∧
simplify-clauses-with-unit-st-inv S {} *T*))›
proof −
show *?thesis*
unfolding *simplify-clauses-with-unit-st-def*
apply (*refine-vcg*)
subgoal using *assms* **apply** −
unfolding *simplify-clauses-with-unit-st-pre-def*
by (*rule exI*[*of* - *T*]) *auto*
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
simplify-clauses-with-unit-st-pre-def
by *blast*
subgoal for *x xa it σ*
apply (*frule simplify-clauses-with-unit-st-inv-simplify-clauses-with-unit-st-preD*,
assumption)
apply (*unfold simplify-clauses-with-unit-st-pre-def*)
apply *normalize-goal+*
apply (*rule simplify-clause-with-unit-st-spec*[*THEN order-trans*])
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def simplify-clause-with-unit-st-pre-def*
by *normalize-goal+ fast*
subgoal
using *empty*
by (*auto 5 3 simp*: *simplify-clauses-with-unit-st-inv-def distinct-mset-dom*
rtranclp.rtrancl-into-rtrancl[*of* - *S*] *distinct-mset-remove1-All*
dest!: *cdcl-twl-inprocessing-l.intros*)
done
subgoal for *x σ*
unfolding *simplify-clauses-with-unit-st-inv-def*
by (*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l*)
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by (*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l*)
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by (*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*)
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by *auto*
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by (*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l*)
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by (*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*)
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by *auto*
subgoal
unfolding *simplify-clauses-with-unit-st-inv-def*
by *auto*
done

qed

definition *simplify-clauses-with-units-st-pre* **where**
‹*simplify-clauses-with-units-st-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧ *twl-list-invs S* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*)) ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
clauses-to-update-l S = {#} ∧
count-decided (*get-trail-l S*) = *0*)›

definition *simplify-clauses-with-units-st* **where**
‹*simplify-clauses-with-units-st S* = *do* {
ASSERT(*simplify-clauses-with-units-st-pre S*);
new-units ← *SPEC* (λ*b*. *b* ⟶ *get-conflict-l S* = *None*);
if new-units
then simplify-clauses-with-unit-st S
else RETURN S}›

lemma *simplify-clauses-with-units-st-spec*:
assumes ‹*count-decided* (*get-trail-l S*) = *0*›
‹*clauses-to-update-l S* = {#}› **and**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
st-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*))› **and**
‹*set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*}›
shows ‹*simplify-clauses-with-units-st S* ≤ ⇓*Id* (*SPEC*(λ*T*. *cdcl-twl-inprocessing-l*** *S T* ∧
simplify-clauses-with-unit-st-inv S {} *T*))›
using *assms* **unfolding** *simplify-clauses-with-units-st-def*
apply (*refine-vcg simplify-clauses-with-unit-st-spec*[*THEN order-trans*])
subgoal unfolding *simplify-clauses-with-units-st-pre-def* **by** *blast*
subgoal by *auto*
apply *assumption*+
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *simplify-clauses-with-unit-st-inv-def* **by** *auto*
done

definition *try-to-forward-subsume-inv* :: ‹*'v twl-st-l* ⇒ *nat multiset* ⇒ *nat* ⇒ *nat* × *bool* × *'v twl-st-l*
⇒ *bool*› **where**
‹*try-to-forward-subsume-inv S0* = (λ*cands C* (*i*,*brk*,*S*).
(*cdcl-twl-inprocessing-l*** *S0 S* ∧
clauses-to-update-l S = {#} ∧
count-decided (*get-trail-l S*) = *0* ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
(¬*brk* ⟶ (*get-conflict-l S* = *None* ∧ *C* ∈# *dom-m* (*get-clauses-l S*) ∧
(∀ *L* ∈# *mset* (*get-clauses-l S* ∝ *C*). *undefined-lit* (*get-trail-l S*) *L*) ∧
length (*get-clauses-l S* ∝ *C*) > *2*)) ∧
(*get-trail-l S* = *get-trail-l S0*) ∧
(∀ *D*∈#*remove1-mset C* (*dom-m* (*get-clauses-l S*)).
D ∈# *dom-m* (*get-clauses-l S0*) ∧ *get-clauses-l S* ∝ *D* = *get-clauses-l S0* ∝ *D*) ∧
(∀ *D*∈#*cands*. *D* ≠ *C* ⟶ (*D* ∈# *dom-m* (*get-clauses-l S0*)) = (*D* ∈# *dom-m* (*get-clauses-l S*)))))›

definition *try-to-forward-subsume-pre* :: ‹*nat* ⇒ *nat multiset* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*try-to-forward-subsume-pre* = (λ*C xs S*.

$\exists\, T.\ C \neq 0 \wedge$
$(S,\ T) \in \textit{twl-st-l None} \wedge$
$\textit{twl-struct-invs}\ T \wedge$
$\textit{twl-list-invs}\ S \wedge$
$\textit{clauses-to-update-l}\ S = \{\#\} \wedge$
$\textit{get-conflict-l}\ S = \textit{None} \wedge$
$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-learned-clauses-entailed-by-init}\ (\textit{state}_W\textit{-of}\ T) \wedge$
$\textit{count-decided}\ (\textit{get-trail-l}\ S) = 0 \wedge$
$\textit{set}\ (\textit{get-all-mark-of-propagated}\ (\textit{get-trail-l}\ S)) \subseteq \{0\} \wedge$
$C \in\#\ \textit{dom-m}\ (\textit{get-clauses-l}\ S) \wedge$
$(\forall\, L \in\#\ \textit{mset}\ (\textit{get-clauses-l}\ S \propto C).\ \textit{undefined-lit}\ (\textit{get-trail-l}\ S)\ L) \wedge$
$\textit{length}\ (\textit{get-clauses-l}\ S \propto C) > 2\rangle$

definition *try-to-forward-subsume* :: ‹*nat* ⇒ *nat multiset* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*try-to-forward-subsume C xs* S_0 = *do* {
ASSERT (*try-to-forward-subsume-pre C xs* S_0);
n ← *RES* {-::*nat*. *True*};
ebreak ← *RES* {-::*bool*. *True*};
(-, -, *S*) ← $\textit{WHILE}_T$ *try-to-forward-subsume-inv* S_0 *xs C*
(λ(*i*, *break*, *S*). ¬*break* ∧ *i* < *n*)
(λ(*i*, *break*, *S*). *do* {
(*S*, *subs*) ← *forward-subsumption-one C xs S*;
ebreak ← *RES* {-::*bool*. *True*};
RETURN (*i*+*1*, *subs* ∨ *ebreak*, *S*)
})
(*0*, *ebreak*, S_0);
RETURN S
}
›

lemma *try-to-forward-forward-subsumption-one-pre*:
‹*try-to-forward-subsume-pre C xs S* ⟹
try-to-forward-subsume-inv S xs C (*a*, *False*, *ba*) ⟹ *forward-subsumption-one-pre C xs ba*›
unfolding *try-to-forward-subsume-inv-def forward-subsumption-one-pre-def case-prod-beta*
try-to-forward-subsume-pre-def
apply *normalize-goal+*
apply (*elim rtranclp-cdcl-twl-inprocessing-l-twl-st-l*)
apply *assumption+*
apply (*rule-tac x*=*V* **in** *exI*)
apply *simp*
done

lemma *in-remove1-mset-dom-m-iff*[*simp*]: ‹*a* ∈# *remove1-mset C* (*dom-m N*) ⟷ *a* ≠ *C* ∧ *a* ∈# *dom-m N*›
using *distinct-mset-dom*[*of N*]
by (*cases* ‹*C* ∈# *dom-m N*›; *auto dest*: *multi-member-split*)

lemma *try-to-forward-subsume*:
assumes ‹*try-to-forward-subsume-pre C cands S*›
shows ‹*try-to-forward-subsume C cands S* ≤ ⇓*Id* (*SPEC*(λ*T*. *cdcl-twl-inprocessing-l*** *S T* ∧
(*length* (*get-clauses-l S* ∝ *C*) > *2* ⟶ *get-trail-l T* = *get-trail-l S*) ∧
(∀ *D*∈#*cands*. *D* ≠ *C* ⟶((*D* ∈# *dom-m* (*get-clauses-l T*)) = (*D* ∈# *dom-m* (*get-clauses-l S*)))) ∧
(∀ *D*∈#*remove1-mset C* (*dom-m* (*get-clauses-l T*)).
D ∈# *dom-m* (*get-clauses-l S*) ∧ *get-clauses-l T* ∝ *D* = *get-clauses-l S* ∝ *D*)))›
proof −
have *wf*: ‹*wf* (*measure* (λ(*i*, -, -). *Suc n* − *i*))› **for** *n*

by *auto*
show *?thesis*
unfolding *try-to-forward-subsume-def*
apply (*refine-vcg forward-subsumption-one*[*THEN order-trans*])
subgoal using *assms* **by** *auto*
apply (*rule-tac n1*=*x* **in** *wf*)
subgoal unfolding *try-to-forward-subsume-inv-def try-to-forward-subsume-pre-def prod.simps*
by (*auto dest*: *in-diffD*)
subgoal for *x s a b aa ba*
by (*auto intro*!: *try-to-forward-forward-subsumption-one-pre*)
subgoal
unfolding *conc-fun-RES RES-RETURN-RES*
apply (*rule RES-rule*, *unfold Image-iff*)
apply (*elim bexE*, *unfold mem-Collect-eq*)
apply (*case-tac xa*, *hypsubst*, *unfold prod.simps*)
apply (*rule RES-rule*)
apply (*simp add*: *try-to-forward-subsume-inv-def*)
apply (*intro conjI impI*)
apply (*auto simp*: *conc-fun-RES RES-RETURN-RES*
try-to-forward-subsume-inv-def dest: *in-diffD*
dest: *rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l*
rtranclp-cdcl-twl-inprocessing-l-count-decided rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated)
done
subgoal by (*auto simp*: *try-to-forward-subsume-inv-def*)
subgoal by (*auto simp*: *try-to-forward-subsume-inv-def*)
subgoal by (*auto simp*: *try-to-forward-subsume-inv-def*)
subgoal by (*auto simp*: *try-to-forward-subsume-inv-def*)
done
qed

definition *forward-subsumption-all-pre* :: ‹*'v twl-st-l* ⇒ *bool*› **where**
‹*forward-subsumption-all-pre* = (λ*S*.
∃ *T*.
(*S*, *T*) ∈ *twl-st-l None* ∧
twl-struct-invs T ∧
twl-list-invs S ∧
clauses-to-update-l S = {#} ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*) ∧
count-decided (*get-trail-l S*) = *0* ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*})›

definition *forward-subsumption-all-inv* :: ‹*'v twl-st-l* ⇒ *nat multiset* × *'v twl-st-l* ⇒ *bool*› **where**
‹*forward-subsumption-all-inv S* = (λ(*xs*, *T*). ∃ *S'*. (*S*,*S'*)∈ *twl-st-l None* ∧ *cdcl-twl-inprocessing-l*** *S* *T* ∧ *xs* ⊆# *dom-m* (*get-clauses-l S*) ∧
twl-struct-invs S' ∧
twl-list-invs S ∧
clauses-to-update-l S = {#} ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S'*) ∧
count-decided (*get-trail-l S*) = *0* ∧
get-trail-l T = *get-trail-l S* ∧ (∀ *C*∈#*xs*. *get-clauses-l T* ∝ *C* = *get-clauses-l S* ∝ *C*) ∧ *xs* ⊆# *dom-m* (*get-clauses-l T*))›

definition *forward-subsumption-all-cands* **where**
‹*forward-subsumption-all-cands S xs* ⟷ *xs* ⊆# *dom-m* (*get-clauses-l S*) ∧
(∀ *C*∈#*xs*. (∀ *L*∈*set* (*get-clauses-l S* ∝ *C*). *undefined-lit* (*get-trail-l S*) *L*) ∧

```
  length (get-clauses-l S ∝ C) > 2)›

definition forward-subsumption-all :: ‹'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹forward-subsumption-all = (λS. do {
  ASSERT (forward-subsumption-all-pre S);
  xs ← SPEC (forward-subsumption-all-cands S);
  (xs, S) ←
    WHILE_T forward-subsumption-all-inv S  (λ(xs, S). xs ≠ {#} ∧ get-conflict-l S = None)
    (λ(xs, S). do {
      C ← SPEC(λC'. C' ∈# xs);
      T ← try-to-forward-subsume C xs S;
      ASSERT (∀ D∈#remove1-mset C xs. get-clauses-l T ∝ D = get-clauses-l S ∝ D);
      RETURN (remove1-mset C xs, T)
    })
    (xs, S);
  RETURN S
  }
)›


lemma forward-subsumption-all:
  assumes ‹forward-subsumption-all-pre S›
  shows ‹forward-subsumption-all S ≤ ⇓Id (SPEC(cdcl-twl-inprocessing-l** S))›
proof −
  let ?R = ‹measure (λ(xs, -). size xs)›
  have simplify-clause-with-unit-st-pre: ‹simplify-clause-with-unit-st-pre D T›
    if
      ‹forward-subsumption-all-pre S› and
      ‹C ⊆# dom-m (get-clauses-l S)› and
      ‹forward-subsumption-all-inv S xsS› and
      ‹case xsS of (xs, S) ⇒ xs ≠ {#} ∧ get-conflict-l S = None› and
      st: ‹xsS = (Cs, T)› and
      ‹D ∈# Cs› and
      ‹¬ D ∉# dom-m (get-clauses-l T)›
    for C xsS Cs T D
  proof −
    have ‹cdcl-twl-inprocessing-l** S T›
    using that unfolding simplify-clause-with-unit-st-pre-def forward-subsumption-all-inv-def st prod.simps
      apply − apply normalize-goal+
      by auto
  show ?thesis
    using rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated[of S T] rtranclp-cdcl-twl-inprocessing-l-twl-st-l[of
S T]
    using that unfolding simplify-clause-with-unit-st-pre-def forward-subsumption-all-inv-def st prod.simps
        forward-subsumption-all-pre-def
      apply − apply normalize-goal+
      apply (auto dest: rtranclp-cdcl-twl-inprocessing-l-count-decided
        rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l
        rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated)
      apply (meson rtranclp-cdcl-twl-inprocessing-l-twl-list-invs)
      by metis
  qed
  have 0: ‹⇓Id  (do {
  ASSERT (forward-subsumption-all-pre S);
  xs ← SPEC (λxs. xs ⊆# (dom-m (get-clauses-l S)));
  (xs, S) ←
```

WHILE$_T$ $\lambda(xs,\ T).$ *cdcl-twl-inprocessing-l*** $S\ T \wedge xs \subseteq\#$ *dom-m* (*get-clauses-l S*) ($\lambda(xs,\ S).$ $xs \neq \{\#\} \wedge$ *get-conflict-l* $S =$ *None*)
($\lambda(xs,\ S').$ *do* {
$C \leftarrow$ *SPEC*($\lambda C'.\ C' \in\# xs$);
$S \leftarrow$ *SPEC*($\lambda T.$ *cdcl-twl-inprocessing-l*** $S'\ T \wedge$ *get-trail-l* $T =$ *get-trail-l* $S \wedge$ *length* (*get-clauses-l* $S \propto C) > 2 \wedge$
remove1-mset $C\ xs \subseteq\#$ *dom-m* (*get-clauses-l* $T) \wedge$
($\forall D \in\#$ *remove1-mset* C (*dom-m* (*get-clauses-l* T)). $D \in\#$ *dom-m* (*get-clauses-l* $S') \wedge$ *get-clauses-l* $T \propto D =$ *get-clauses-l* $S' \propto D$));
RETURN (*remove1-mset* $C\ xs,\ S$)
})
($xs,\ S$);
RETURN S
}) $\leq \Downarrow Id$ (*SPEC*(*cdcl-twl-inprocessing-l*** S))›
apply (*subst* (*1*) *Down-id-eq*)
apply (*refine-vcg WHILEIT-rule*[**where** *R*= *?R*] *forward-subsumption-one*[*THEN order-trans*]
simplify-clause-with-unit-st-spec[*THEN order-trans*])
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by (*auto simp*: *forward-subsumption-all-inv-def*)
subgoal by (*auto simp*: *forward-subsumption-all-inv-def*)
subgoal by (*auto simp*: *size-mset-remove1-mset-le-iff*)
subgoal by (*auto simp*: *forward-subsumption-all-inv-def*)
subgoal by (*auto simp*: *size-mset-remove1-mset-le-iff*)
subgoal by (*auto simp*: *forward-subsumption-all-inv-def*)
done
let *?R* = ‹$\lambda xs_0.$ {((*xs*, *U*), (*ys*,*V*)). (*xs*,*ys*)$\in$*Id* $\wedge$ *xs* $\subseteq\#$ $xs_0 \wedge$ *distinct-mset xs* $\wedge$ ($U,\ V$) $\in$ *Id* $\wedge$ *get-trail-l* $U =$ *get-trail-l* $S \wedge$
($\forall C \in\#$ *xs*. $C \in\#$*dom-m* (*get-clauses-l* $U) \wedge$ *get-clauses-l* $U \propto C =$ *get-clauses-l* $S \propto C) \wedge xs \subseteq\#$ *dom-m* (*get-clauses-l* U)}›
have *try-to-forward-subsume-pre*: ‹*try-to-forward-subsume-pre C x1a x2a*›
if
pre: ‹*forward-subsumption-all-pre S*› **and**
‹(*xs*, *xsa*) $\in$ *Id*› **and**
cands: ‹*xs* $\in$ *Collect* (*forward-subsumption-all-cands S*)› **and**
‹*xsa* $\in$ {*xs*. *xs* $\subseteq\#$ *dom-m* (*get-clauses-l S*)}› **and**
xx': ‹($x,\ x'$) $\in$ *?R xsa*› **and**
break: ‹*case x of* (*xs*, *S*) $\Rightarrow$ *xs* $\neq \{\#\} \wedge$ *get-conflict-l S* = *None*› **and**
‹*case x' of* (*xs*, *S*) $\Rightarrow$ *xs* $\neq \{\#\} \wedge$ *get-conflict-l S* = *None*› **and**
inv: ‹*forward-subsumption-all-inv S x*› **and**
st: ‹$x' = (x1,\ x2)$› ‹$x = (x1a,\ x2a)$› **and**
CCa: ‹($C,\ Ca$) $\in$ *nat-rel*› **and**
‹$C \in \{C'.\ C' \in\#$ *x1a*}› **and**
‹*Ca* $\in \{C'.\ C' \in\#$ *x1*}›
for *xs xsa x x' x1 x2 x1a x2a C Ca T U*
proof −
have *TU*: ‹*x2* = *x2a*› **and**
Sx2a: ‹*cdcl-twl-inprocessing-l*** *S x2*› **and**
st': ‹*x1* = *x1a*› ‹*x2* = *x2a*› ‹*C*=*Ca*› **and**
C: ‹*C* $\in\#$ *dom-m* (*get-clauses-l x2a*)› **and**
undef: ‹$\forall$ *L*$\in$*set* (*get-clauses-l x2* $\propto$ *C*). *undefined-lit* (*get-trail-l x2a*) *L*› **and**
[*intro*]: ‹*2* < *length* (*get-clauses-l x2a* $\propto$ *Ca*)› **and**
confl: ‹*get-conflict-l x2a* = *None*›
using *inv xx' CCa cands break*
unfolding *forward-subsumption-all-inv-def st forward-subsumption-all-cands-def*
by (*auto dest*!: *multi-member-split simp*: *ball-conj-distrib*)

```
    (metis mem-Collect-eq pair-in-Id-conv subset-mset.le-iff-add that(13) that(2) union-iff)+
  obtain x where
    Sx: ‹(S, x) ∈ twl-st-l None› and
    struct-S: ‹twl-struct-invs x› and
    list-S: ‹twl-list-invs S› and
    [simp]: ‹clauses-to-update-l S = {#}› and
    ent-S: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of x)› and
    [simp]: ‹count-decided (get-trail-l S) = 0› and
    marks: ‹set (get-all-mark-of-propagated (get-trail-l S)) ⊆ {0}›
    using pre unfolding forward-subsumption-all-pre-def by fast
  have [intro]: ‹2 < length (get-clauses-l x2a ∝ Ca)›
    using CCa by auto

  show ?thesis
    using undef C confl apply −
    apply (rule rtranclp-cdcl-twl-inprocessing-l-twl-st-l[OF Sx2a Sx struct-S list-S ent-S])
    subgoal for V
    unfolding st′ TU
    unfolding try-to-forward-subsume-pre-def TU
    apply (rule-tac x=V in exI)
    using rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l[OF Sx2a]
      rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated[OF Sx2a]
      rtranclp-cdcl-twl-inprocessing-l-count-decided[OF Sx2a]
      rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated[OF Sx2a]
      marks
    by (auto simp add: twl-list-invs-def TU)
    done
 qed
 have [refine]: ‹(xs,xsa)∈Id ⟹xsa ⊆# dom-m (get-clauses-l S)⟹((xs, S), xsa, S) ∈ ?R xsa› for xs
xsa
  by (auto simp add: Duplicate-Free-Multiset.distinct-mset-mono distinct-mset-dom)
 show ?thesis
  apply (rule order-trans)
  prefer 2
  apply (rule 0)
  unfolding forward-subsumption-all-def
  apply (refine-vcg WHILEIT-refine[where R=‹?R xs› for xs]
    try-to-forward-subsume
    forward-subsumption-one[THEN order-trans])
  subgoal using assms by (auto simp: forward-subsumption-all-cands-def)
  subgoal by auto
  subgoal unfolding forward-subsumption-all-pre-def forward-subsumption-all-inv-def case-prod-beta
apply normalize-goal+
    by (rename-tac xa, rule-tac x=xa in exI) (auto simp: forward-subsumption-all-inv-def)
  subgoal by (auto simp: forward-subsumption-all-inv-def)
  subgoal by (auto simp: size-mset-remove1-mset-le-iff)
  apply (rule try-to-forward-subsume[THEN order-trans])
  subgoal by (rule try-to-forward-subsume-pre)
  subgoal unfolding Down-id-eq
    by (rule SPEC-rule) (auto simp: forward-subsumption-all-cands-def Ball-def
      forward-subsumption-all-inv-def distinct-mset-remove1-All
      intro!: distinct-subseteq-iff[THEN iffD1])
  subgoal
    by (auto dest: in-diffD simp: distinct-mset-remove1-All)
  subgoal
    by (auto dest: in-diffD simp: distinct-mset-remove1-All)
```

subgoal by *auto*
done
qed

definition *forward-subsumption-needed-l* :: ‹-› **where**
‹*forward-subsumption-needed-l S* = *SPEC* (λ-. *True*)›

definition *forward-subsume-l* :: ‹-› **where**
‹*forward-subsume-l S* = *do* {
ASSERT (*forward-subsumption-all-pre S*);
b ← *forward-subsumption-needed-l S*;
if b then forward-subsumption-all S else RETURN S
}›

lemma *forward-subsume-l*:
assumes ‹*forward-subsumption-all-pre S*›
shows ‹*forward-subsume-l S* ≤ ⇓*Id* (*SPEC*(*cdcl-twl-inprocessing-l*** *S*))›
unfolding *forward-subsume-l-def forward-subsumption-needed-l-def*
by (*refine-vcg forward-subsumption-all*[*unfolded Down-id-eq*]) (*use assms* **in** *auto*)

5.3.1 Pure Literal Deletion

definition *propagate-pure-l-pre*:: ‹'*v literal* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
‹*propagate-pure-l-pre L S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *L* ∈# *all-init-lits-of-l S* ∧ *undefined-lit* (*get-trail-l S*) *L* ∧
clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None* ∧
count-decided (*get-trail-l S*) = 0 ∧ −*L* ∉ ⋃(*set-mset* ‘ *set-mset* (*mset* ‘# *get-init-clss-l S*)))›

definition *propagate-pure-bt-l* :: ‹'*v literal* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*propagate-pure-bt-l* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*propagate-pure-l-pre L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
M ← *cons-trail-propagate-l L 0 M*;
RETURN (*M*, *N*, *D*, *NE*, *UE*, *add-mset* {#*L*#} *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* (−*L*) *Q*)})›

lemma *in-all-lits-of-mm-Pos-Neg*:
‹*L* ∈# *all-lits-of-mm A* ⟷ *L* ∈ ⋃(*set-mset* ‘ *set-mset A*) ∨ −*L* ∈ ⋃(*set-mset* ‘ *set-mset A*)›
by (*cases L*)
(*auto simp*: *all-lits-of-mm-def image-UN dest*!: *multi-member-split*)

lemma *propagate-pure-bt-l-spec*:
assumes
‹*propagate-pure-l-pre L S*› **and**
‹*undefined-lit* (*get-trail-l S*) *L*›
shows ‹*propagate-pure-bt-l L S* ≤ *SPEC* (λ*T*. *cdcl-twl-pure-remove-l S T*)›
proof −
have [*iff*]: ‹¬(∀ *b*. ¬ *b*)› ‹(∃ *b*. *b*)›
by *blast*+
show *?thesis*
using *assms*
unfolding *propagate-pure-bt-l-def cons-trail-propagate-l-def*
apply (*cases S*)
apply *refine-vcg*
subgoal by *simp*
subgoal by *auto*

```
    subgoal
      unfolding propagate-pure-l-pre-def
      apply (subst cdcl-twl-pure-remove-l.simps)
      apply simp
      apply (clarsimp simp add: propagate-pure-l-pre-def get-init-clss-l-def
        all-init-lits-of-l-def all-lits-of-mm-union)
      apply metis
      done
    done
qed


lemma in-all-lits-of-mm-direct-inI: ‹−L ∈ ⋃(set-mset ‘ set-mset A) ⟹ L ∈# all-lits-of-mm A›
  by (auto simp: all-lits-of-mm-add-mset all-lits-of-m-add-mset
    dest!: multi-member-split)
lemma cdcl-twl-pure-remove-l-same:
  ‹cdcl-twl-pure-remove-l S T ⟹ count-decided (get-trail-l T) = count-decided (get-trail-l S)›
  ‹cdcl-twl-pure-remove-l S T ⟹ clauses-to-update-l T = clauses-to-update-l S›
  ‹cdcl-twl-pure-remove-l S T ⟹ get-conflict-l T = get-conflict-l S›
  ‹cdcl-twl-pure-remove-l S T ⟹ get-init-clss-l T = get-init-clss-l S›
  ‹cdcl-twl-pure-remove-l S T ⟹ get-subsumed-init-clauses-l T = get-subsumed-init-clauses-l S›
  ‹cdcl-twl-pure-remove-l S T ⟹ get-init-clauses0-l T = get-init-clauses0-l S›
  ‹cdcl-twl-pure-remove-l S T ⟹ set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S)› and
 cdcl-twl-pure-remove-l-same-unit-init-subsetD:
  ‹cdcl-twl-pure-remove-l S T ⟹ get-unit-init-clauses-l S ⊆# get-unit-init-clauses-l T›
proof −
  have [iff]: ‹¬(∀ b. ¬ b)›
    by blast
  have [simp]: ‹L ∈# a ⟹−L ∈# all-lits-of-m (a)›‹L ∈# a ⟹L ∈# all-lits-of-m (a)› for L a
    by (simp-all add: atm-of-lit-in-atms-of in-all-lits-of-m-ain-atms-of-iff in-clause-in-all-lits-of-m)
  have [simp]: ‹− L ∈# all-lits-of-mm b ⟷ L ∈# all-lits-of-mm b› for b L
    by (cases L; auto simp: all-lits-of-mm-def)
  show
    ‹cdcl-twl-pure-remove-l S T ⟹ count-decided (get-trail-l T) = count-decided (get-trail-l S)›
    ‹cdcl-twl-pure-remove-l S T ⟹ clauses-to-update-l T = clauses-to-update-l S›
    ‹cdcl-twl-pure-remove-l S T ⟹ get-conflict-l T = get-conflict-l S›
    ‹cdcl-twl-pure-remove-l S T ⟹ get-init-clss-l T = get-init-clss-l S›
    ‹cdcl-twl-pure-remove-l S T ⟹ get-subsumed-init-clauses-l T = get-subsumed-init-clauses-l S›
    ‹cdcl-twl-pure-remove-l S T ⟹ get-init-clauses0-l T = get-init-clauses0-l S›
    ‹cdcl-twl-pure-remove-l S T ⟹ set-mset (all-init-lits-of-l T) = set-mset (all-init-lits-of-l S)›
    ‹cdcl-twl-pure-remove-l S T ⟹ get-unit-init-clauses-l S ⊆# get-unit-init-clauses-l T›
    by (solves ‹auto simp: cdcl-twl-pure-remove-l.simps all-init-lits-of-l-def get-init-clss-l-def
      all-lits-of-mm-add-mset all-lits-of-m-add-mset
      intro: in-all-lits-of-mm-direct-inI
      dest!: multi-member-split[of ‹-› ‹ran-m -›]
        multi-member-split[of ‹-› ‹- :: - clauses›]›)+
     (auto simp: cdcl-twl-pure-remove-l.simps all-init-lits-of-l-def get-init-clss-l-def
      all-lits-of-mm-add-mset all-lits-of-m-add-mset all-lits-of-mm-union
      intro: in-all-lits-of-mm-direct-inI
      dest!: multi-member-split[of ‹-› ‹ran-m -›]
        multi-member-split[of ‹-› ‹- :: - clauses›])
qed

lemma rtranclp-cdcl-twl-pure-remove-l-same:
  ‹cdcl-twl-pure-remove-l** S T ⟹ count-decided (get-trail-l T) = count-decided (get-trail-l S)›
  ‹cdcl-twl-pure-remove-l** S T ⟹ clauses-to-update-l T = clauses-to-update-l S›
```

⟨*cdcl-twl-pure-remove-l*** *S T* ⟹ *get-conflict-l T* = *get-conflict-l S*⟩
⟨*cdcl-twl-pure-remove-l*** *S T* ⟹ *get-init-clss-l T* = *get-init-clss-l S*⟩
⟨*cdcl-twl-pure-remove-l*** *S T* ⟹ *get-subsumed-init-clauses-l T* = *get-subsumed-init-clauses-l S*⟩
⟨*cdcl-twl-pure-remove-l*** *S T* ⟹ *get-init-clauses0-l T* = *get-init-clauses0-l S*⟩
⟨*cdcl-twl-pure-remove-l*** *S T* ⟹ *set-mset* (*all-init-lits-of-l T*) = *set-mset* (*all-init-lits-of-l S*)⟩ **and**
rtranclp-cdcl-twl-pure-remove-l-same-unit-init-subsetD:
⟨*cdcl-twl-pure-remove-l*** *S T* ⟹ *get-unit-init-clauses-l S* ⊆# *get-unit-init-clauses-l T*⟩
by (*solves* ⟨*induction rule*: *rtranclp-induct*; *auto simp*: *cdcl-twl-pure-remove-l-same*
dest: *cdcl-twl-pure-remove-l-same-unit-init-subsetD*⟩)+

5.3.2 Pure Literal deletion

Pure literal deletion is not really used nowadays (as it is subsumed by variable elimination), but it is the first non-model preserving model we intend to implement and it should be easy to implement.

In the implementation, it would better to simplify also when going over the clauses, instead of either (i) no simplifying anything or (ii) simplify all clauses. However, in the current version, I can reuse the other proofs.

definition *pure-literal-deletion-pre* **where**
⟨*pure-literal-deletion-pre S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧
clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None* ∧ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S'*) ∧
count-decided (*get-trail-l S*) = *0* ∧ *twl-list-invs S* ∧ *twl-struct-invs S'*)⟩

definition *pure-literal-deletion-candidates* **where**
⟨*pure-literal-deletion-candidates S* = *SPEC* (λ*Ls*. *set-mset Ls* ⊆ *atms-of-mm* (*get-all-init-clss-l S*))⟩

definition *pure-literal-deletion-l-inv* **where**
⟨*pure-literal-deletion-l-inv S xs0* = (λ(*T*, *xs*).
∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *cdcl-twl-pure-remove-l*** *S T* ∧ *xs* ⊆# *xs0*)⟩

definition *pure-literal-deletion-l* :: ⟨'*v twl-st-l* ⇒ '*v twl-st-l nres*⟩ **where**
⟨*pure-literal-deletion-l S* = *do* {
ASSERT (*pure-literal-deletion-pre S*);
let As = ⋃(*set-mset* ‘ *set-mset* (*mset* ‘# *get-init-clss-l S*));
xs ← *pure-literal-deletion-candidates S*;
(*S*, *xs*) ← *WHILE*$_T$*pure-literal-deletion-l-inv S xs* (λ(*S*, *xs*). *xs* ≠ {#})
(λ(*S*, *xs*). *do* {
L ← *SPEC* (λ*L*. *L* ∈# *xs*);
A ← *RES* {*Pos L*, *Neg L*};
if −*A* ∉ *As* ∧ *undefined-lit* (*get-trail-l S*) *A*
then do {*S* ← *propagate-pure-bt-l A S*;
RETURN (*S*, *remove1-mset L xs*)}
else RETURN (*S*, *remove1-mset L xs*)
})
(*S*, *xs*);
RETURN S
}⟩

lemma *pure-literal-deletion-l-spec*:
assumes ⟨*pure-literal-deletion-pre S*⟩

```
  shows
    ‹pure-literal-deletion-l S ≤ SPEC (λT. cdcl-twl-pure-remove-l** S T)›
proof −
  have [simp]: ‹a − Suc b < a − b ⟷ Suc b ≤ a› for a b
    by auto
  have [refine0]: ‹wf (measure (λ(-, x). size x))›
    by auto
  show ?thesis
    unfolding pure-literal-deletion-l-def Let-def pure-literal-deletion-candidates-def
    apply (refine-vcg propagate-pure-bt-l-spec[THEN order-trans])
    subgoal using assms by fast
    subgoal unfolding pure-literal-deletion-l-inv-def pure-literal-deletion-pre-def
      by fast
    subgoal for x s T xs L unfolding propagate-pure-l-pre-def pure-literal-deletion-l-inv-def
       case-prod-beta pure-literal-deletion-pre-def
      apply normalize-goal+
      apply (frule rtranclp-cdcl-twl-pure-remove-l-cdcl-twl-pure-remove)
      apply assumption+
      apply normalize-goal+
      apply (rule-tac x=xd in exI)
      using rtranclp-cdcl-twl-pure-remove-l-same-unit-init-subsetD[of S T]
      unfolding atms-of-ms-def[symmetric]
      apply (auto simp add: rtranclp-cdcl-twl-pure-remove-l-same
        all-init-lits-of-l-def in-all-lits-of-mm-ain-atms-of-iff
        dest!: multi-member-split[of L xs] multi-member-split[of L x]
        mset-subset-eq-insertD)
      apply (metis UnCI atms-of-ms-union set-mset-union subset-mset.le-iff-add)
      by (metis Un-iff atms-of-ms-union set-mset-union subset-mset.le-iff-add)
    subgoal by simp
    subgoal by (auto dest!: multi-member-split
      simp: pure-literal-deletion-l-inv-def)
    subgoal by (auto dest!: multi-member-split
      simp: pure-literal-deletion-l-inv-def)
    subgoal
      by (simp add: pure-literal-deletion-l-inv-def)
    subgoal
      by (auto dest!: multi-member-split)
    subgoal by (simp add: pure-literal-deletion-l-inv-def)
    done
qed
```

```
definition pure-literal-count-occs-l-clause-invs :: ‹nat ⇒ 'v twl-st-l ⇒-⇒ nat × ('v literal ⇒ bool) ⇒
bool› where
  ‹pure-literal-count-occs-l-clause-invs C S occs = (λ(i, occs2). i ≤ length (get-clauses-l S ∝ C) ∧
     (∀ L. occs2 L = (occs L ∨ (L ∈ set (take i (get-clauses-l S ∝ C))))))›

definition pure-literal-count-occs-l-clause-pre :: ‹nat ⇒ 'v twl-st-l ⇒ - ⇒ bool› where
  ‹pure-literal-count-occs-l-clause-pre C S occs = (C ∈# dom-m (get-clauses-l S) ∧ irred (get-clauses-l
S) C)›

definition pure-literal-count-occs-l-clause :: ‹nat ⇒ 'v twl-st-l ⇒ - ⇒ ('v literal ⇒ bool) nres› where
  ‹pure-literal-count-occs-l-clause C S occs = do {
    ASSERT (pure-literal-count-occs-l-clause-pre C S occs);
```

```
    (i, occs) ← WHILE_T^pure-literal-count-occs-l-clause-invs C S occs (λ(i, occs). i < length (get-clauses-l
S ∝ C))
      (λ(i, occs). do {
        let L = get-clauses-l S ∝ C ! i;
        let occs = occs (L := True);
        RETURN (i+1, occs)
      })
      (0, occs);
   RETURN occs
 }›


lemma pure-literal-count-occs-l-clause-spec:
  assumes ‹pure-literal-count-occs-l-clause-pre C S occs›
  shows ‹pure-literal-count-occs-l-clause C S occs ≤ SPEC (λoccs2.
     (occs2 = (λL. if L ∈ set (get-clauses-l S ∝ C) then True else occs L)))›
proof −
  have [iff]: ‹(∀ j. ¬ j < a) ⟷ a = 0› for a :: nat
    by auto
  have [simp]: ‹filter-mset P xs − xs = {#}› for P xs
    by (induction xs) auto
  have [refine0]: ‹wf (measure (λ(A, -). length (get-clauses-l S ∝ C) − A))›
    by auto
  show ?thesis
    unfolding pure-literal-count-occs-l-clause-def
    apply (refine-vcg)
    subgoal using assms by fast
    subgoal unfolding pure-literal-count-occs-l-clause-invs-def
      by (auto dest: in-diffD dest: multi-member-split)
    subgoal unfolding pure-literal-count-occs-l-clause-invs-def
      apply simp
      by (metis in-set-conv-iff less-Suc-eq)
    subgoal
      by auto
    subgoal unfolding pure-literal-count-occs-l-clause-invs-def
      by auto
    done
qed

definition pure-literal-count-occs-l-pre :: ‹- ⇒ -› where
  ‹pure-literal-count-occs-l-pre S ⟷
    (∃ S'. (S, S') ∈ twl-st-l None ∧
    clauses-to-update-l S = {#} ∧ get-conflict-l S = None ∧
    set (get-all-mark-of-propagated (get-trail-l S)) ⊆ {0} ∧
    cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S') ∧
    count-decided (get-trail-l S) = 0 ∧ twl-list-invs S ∧ twl-struct-invs S')›

definition pure-literal-count-occs-l-inv :: ‹'v twl-st-l ⇒ nat multiset ⇒ nat multiset × ('v literal ⇒ bool)
× bool ⇒ -› where
  ‹pure-literal-count-occs-l-inv S xs0 = (λ(A, occs, -). A ⊆# xs0 ∧
    (occs = (λL. L∈⋃(set-mset ‘ set-mset (mset ‘# (λC. get-clauses-l S ∝ C) ‘# ({#C ∈# xs0. (C
∈# dom-m (get-clauses-l S) ∧ irred (get-clauses-l S) C)#} − A))))))›
```

We allow the solver to abort the search (for example if you already know that there are no pure literals).

```
definition pure-literal-count-occs-l :: ‹'v twl-st-l ⇒ -› where
```

```
  ‹pure-literal-count-occs-l S = do {
  ASSERT (pure-literal-count-occs-l-pre S);
  xs ← SPEC (λxs. distinct-mset xs ∧ (∀ C∈#dom-m (get-clauses-l S). irred (get-clauses-l S) C ⟶ C
∈# xs));
  abort ← RES (UNIV :: bool set);
  let occs = (λ-. False);
  (-, occs, abort) ← WHILE_T^pure-literal-count-occs-l-inv S xs(λ(A, occs, abort). A ≠ {#} ∧ ¬abort)
     (λ(A, occs, abort). do {
       ASSERT (A ≠ {#});
       C ← SPEC (λC. C ∈# A);
       if (C ∈# dom-m (get-clauses-l S) ∧ irred (get-clauses-l S) C)
       then do {
           occs ← pure-literal-count-occs-l-clause C S occs;
           abort ← RES (UNIV :: bool set);
           RETURN (remove1-mset C A, occs, abort)
       } else RETURN (remove1-mset C A, occs, abort)
     })
     (xs, occs, abort);
   RETURN (abort, occs)
  }›
```

```
lemma filter-mset-remove-itself [simp]: ‹filter-mset P xs − xs = {#}› for P xs
  by (induction xs) auto
```

```
lemma pure-literal-count-occs-l-spec:
  assumes ‹pure-literal-count-occs-l-pre S›
  shows ‹pure-literal-count-occs-l S ≤ SPEC (λ(abort, occs). ¬abort ⟶
     (∀ L. occs L = (L ∈ ⋃(set-mset ‘ set-mset (mset ‘# get-init-clss-l S)))))›
proof −
  have H: ‹y∉# A ⟹ x ∈# A − remove1-mset y B ⟷ x ∈# A − B › for x y A B
    apply (auto simp: minus-remove1-mset-if)
    done
  have [refine0]: ‹wf (measure (λ(A, -). size A))›
    by auto
  show ?thesis
    unfolding pure-literal-count-occs-l-def
    apply (refine-vcg pure-literal-count-occs-l-clause-spec)
    subgoal using assms by fast
    subgoal unfolding pure-literal-count-occs-l-inv-def
      by (auto dest: in-diffD dest: multi-member-split)
    subgoal by auto
    subgoal
      unfolding pure-literal-count-occs-l-clause-pre-def
      by simp
    subgoal for x abort s a b aa ba xa xb xc
      using distinct-mset-dom[of ‹get-clauses-l S›]
     apply (auto dest!: multi-member-split[of ‹- :: nat› a] multi-member-split[of xa] dest: mset-subset-eq-insertD
        simp add: pure-literal-count-occs-l-inv-def intro!: ext split: if-splits)
      apply (meson in-multiset-minus-notin-snd mset-subset-eqD union-single-eq-member)
      apply (metis (no-types, lifting) add-mset-remove-trivial distinct-mem-diff-mset distinct-mset-filter
in-diffD in-multiset-minus-notin-snd)
     by (metis (no-types, lifting) diff-add-mset-swap insert-DiffM insert-noteq-member mset-subset-eqD)
    subgoal
      by (auto dest!: multi-member-split)
```

```
    subgoal for x abort s a b aa ba xa
      by (auto simp: pure-literal-count-occs-l-inv-def minus-remove1-mset-if)
    subgoal
      by (auto dest!: multi-member-split)
    subgoal
      by (auto simp: pure-literal-count-occs-l-inv-def get-init-clss-l-def ran-m-def
        eq-commute[of - ‹the ->] all-conj-distrib conj-disj-distribR ex-disj-distrib
        dest!: multi-member-split)
    done
qed


definition pure-literal-deletion-l2 :: ‹- ⇒ 'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹pure-literal-deletion-l2 occs S = do {
    ASSERT (pure-literal-deletion-pre S);
    let As = ⋃(set-mset ‘ set-mset (mset ‘# get-init-clss-l S));
    xs ← pure-literal-deletion-candidates S;
    (S, xs) ← WHILE_T^pure-literal-deletion-l-inv S xs (λ(S, xs). xs ≠ {#})
      (λ(S, xs). do {
        L ← SPEC (λL. L ∈# xs);
        let A = (if occs (Pos L) ∧ ¬occs (Neg L) then Pos L else Neg L);
        if ¬occs (−A) ∧ undefined-lit (get-trail-l S) A
        then do {S ← propagate-pure-bt-l A S;
          RETURN (S, remove1-mset L xs)}
        else RETURN (S, remove1-mset L xs)
      })
    (S, xs);
  RETURN S
  }›

lemma pure-literal-deletion-l2-pure-literal-deletion-l:
  assumes ‹(∀ L. occs L = (L ∈ ⋃(set-mset ‘ set-mset (mset ‘# get-init-clss-l S))))›
  shows ‹pure-literal-deletion-l2 occs S ≤ ⇓Id (pure-literal-deletion-l S)›
proof −
  have 1: ‹a=b ⟹ (a,b) ∈ Id› and 2: ‹c ∈ Φ ⟹ RETURN c ≤⇓Id (RES Φ)› and
    3: ‹e = f ⟹ e ≤⇓Id f› for a b c Φ e f
    by auto
  show ?thesis
    unfolding pure-literal-deletion-l2-def pure-literal-deletion-l-def
    apply (refine-vcg)
    apply (rule 1)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    apply (rule 2)
    subgoal by auto
    subgoal using assms by (auto split: if-splits)
    apply (rule 3)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
qed
```

definition *pure-literal-elimination-round-pre* **where**
‹*pure-literal-elimination-round-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧ *twl-list-invs S* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*)) ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
clauses-to-update-l S = {#} ∧
count-decided (*get-trail-l S*) = *0*)›

definition *pure-literal-elimination-round* **where**
‹*pure-literal-elimination-round S* = *do* {
ASSERT (*pure-literal-elimination-round-pre S*);
S ← *simplify-clauses-with-units-st S*;
if get-conflict-l S = *None*
then do {
(*abort*, *occs*) ← *pure-literal-count-occs-l S*;
if ¬*abort then pure-literal-deletion-l2 occs S*
else RETURN S}
else RETURN S
}›

lemma *pure-literal-elimination-round*:
assumes ‹*pure-literal-elimination-round-pre S*›
shows ‹*pure-literal-elimination-round S* ≤ *SPEC* (λ*T*. *cdcl-twl-inprocessing-l*** *S T*)›
proof −
have *pure-literal-deletion-pre*: ‹*pure-literal-elimination-round-pre S* ⟹
*cdcl-twl-inprocessing-l*** *S x* ∧ *simplify-clauses-with-unit-st-inv S* {} *x* ⟹
get-conflict-l x = *None* ⟹
pure-literal-deletion-pre x› **for** *x*
unfolding *pure-literal-deletion-pre-def pure-literal-elimination-round-pre-def* **apply** *normalize-goal+*
apply (*frule rtranclp-cdcl-twl-inprocessing-l-twl-st-l*; *assumption?*)
apply (*rule-tac x*=*V* **in** *exI*)
apply (*simp add*: *rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l*
rtranclp-cdcl-twl-inprocessing-l-count-decided)
done
have *pure-literal-count-occs-l-pre*: ‹*pure-literal-elimination-round-pre S* ⟹
*cdcl-twl-inprocessing-l*** *S x* ∧ *simplify-clauses-with-unit-st-inv S* {} *x* ⟹
get-conflict-l x = *None* ⟹
pure-literal-count-occs-l-pre x› **for** *x*
unfolding *pure-literal-count-occs-l-pre-def pure-literal-elimination-round-pre-def* **apply** *normalize-goal+*
apply (*frule rtranclp-cdcl-twl-inprocessing-l-twl-st-l*; *assumption?*)
apply (*rule-tac x*=*V* **in** *exI*)
apply (*auto simp add*: *rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l simplify-clauses-with-unit-st-inv-def*
rtranclp-cdcl-twl-inprocessing-l-count-decided)
done

obtain *T* **where**
T: ‹ (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧ *twl-list-invs S* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*)) ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
clauses-to-update-l S = {#} ∧
count-decided (*get-trail-l S*) = *0*›
using *assms* **unfolding** *pure-literal-elimination-round-pre-def* **by** *blast*
show *?thesis*
unfolding *pure-literal-elimination-round-def*
apply (*refine-vcg*

simplify-clauses-with-units-st-spec[*THEN order-trans*, *of* - *T*])
subgoal using *assms* **by** *auto*
subgoal using *T* **by** *blast*
subgoal using *T* **by** *blast*
subgoal using *T* **by** *blast*
subgoal using *T* **by** *blast*
subgoal using *T* **by** *blast*
subgoal using *T* **by** *blast*
subgoal using *T* **by** *blast*
apply (*subst Down-id-eq*)
apply (*refine-vcg pure-literal-count-occs-l-spec*
pure-literal-deletion-l2-pure-literal-deletion-l[*THEN order-trans*]
pure-literal-deletion-l-spec[*THEN order-trans*])
subgoal by (*rule pure-literal-count-occs-l-pre*)
subgoal by *auto*
subgoal
apply (*subst Down-id-eq*)
apply (*rule pure-literal-deletion-l-spec*[*THEN order-trans*])
apply (*simp-all add*: *pure-literal-deletion-pre*)
by (*smt* (*verit*, *best*) *Collect-mono cdcl-twl-inprocessing-l.intros*(*5*) *mono-rtranclp rtranclp-trans*)
subgoal by *auto*
subgoal by *auto*
done
qed

definition *pure-literal-elimination-l-pre* **where**
‹*pure-literal-elimination-l-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧ *twl-list-invs S* ∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* ((*state*$_W$*-of T*)) ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
clauses-to-update-l S = {#} ∧
count-decided (*get-trail-l S*) = *0*)›

definition *pure-literal-elimination-l-inv* **where**
‹*pure-literal-elimination-l-inv S max-rounds* = (λ(*T*,*n*,-). *n* ≤ *max-rounds* ∧ *cdcl-twl-inprocessing-l*** *S T*)›

definition *pure-literal-elimination-l* :: ‹-› **where**
‹*pure-literal-elimination-l S* = *do* {
ASSERT (*pure-literal-elimination-l-pre S*);
max-rounds ← *RES* (*UNIV* :: *nat set*);
(*S*, -, -) ← *WHILE*$_T$*pure-literal-elimination-l-inv S max-rounds* (λ(*S*, *m*, *abort*). *m* < *max-rounds* ∧ ¬*abort*)
(λ(*S*, *m*, *abort*). *do* {
S ← *pure-literal-elimination-round S*;
abort ← *RES* (*UNIV* :: *bool set*);
RETURN (*S*, *m*+*1*, *abort*)
})
(*S*, *0*, *False*);
RETURN S
}›

lemma *pure-literal-elimination-l*:
assumes ‹*pure-literal-elimination-l-pre S*›

```
  shows ‹pure-literal-elimination-l S ≤ SPEC (λT. cdcl-twl-inprocessing-l** S T)›
proof −
  have [refine0]: ‹max-rounds ∈ UNIV ⟹ wf (measure (λ(S,n,m). max-rounds − n))› for max-rounds
    by auto
  have H: ‹pure-literal-elimination-l-pre S ⟹
    x ∈ UNIV ⟹
    pure-literal-elimination-l-inv S x s ⟹
    case s of (S, m, abort) ⇒ m < x ∧ ¬ abort ⟹
      s = (a, b) ⟹ b = (aa, ba) ⟹ pure-literal-elimination-round-pre a› for x s a aa ba b S
    apply hypsubst
    unfolding pure-literal-elimination-round-pre-def pure-literal-elimination-l-inv-def
       pure-literal-elimination-l-pre-def prod.simps
    apply normalize-goal+
    apply (frule rtranclp-cdcl-twl-inprocessing-l-twl-st-l; assumption?)
    apply (rule-tac x=V in exI)
   apply (auto simp add: rtranclp-cdcl-twl-inprocessing-l-clauses-to-update-l simplify-clauses-with-unit-st-inv-def
      rtranclp-cdcl-twl-inprocessing-l-count-decided
      dest: rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated)
    done
  show ?thesis
    unfolding pure-literal-elimination-l-def
    apply (refine-vcg pure-literal-elimination-round)
    subgoal using assms by auto
    apply assumption
    subgoal unfolding pure-literal-elimination-l-inv-def by auto
    subgoal by (rule H)
    subgoal by (auto simp: pure-literal-elimination-l-inv-def)
    subgoal by auto
    subgoal by (auto simp: pure-literal-elimination-l-inv-def)
    done
qed


definition pure-literal-elimination-l-needed :: ‹'v twl-st-l ⇒ bool nres› where
  ‹pure-literal-elimination-l-needed S = SPEC (λ-. True)›

definition pure-literal-eliminate-l :: ‹-› where
  ‹pure-literal-eliminate-l S = do {
     ASSERT (pure-literal-elimination-l-pre S);
     b ← pure-literal-elimination-l-needed S;
     if b then pure-literal-elimination-l S else RETURN S
  }›


lemma pure-literal-eliminate-l:
  assumes ‹pure-literal-elimination-l-pre S›
  shows ‹pure-literal-eliminate-l S ≤ SPEC (λT. cdcl-twl-inprocessing-l** S T)›
  unfolding pure-literal-eliminate-l-def pure-literal-elimination-l-needed-def
  by (refine-vcg pure-literal-elimination-l)
   (use assms in auto)

end
theory Watched-Literals-List-Restart
  imports Watched-Literals-List Watched-Literals-Algorithm-Reduce
begin
```

Unlike most other refinements steps we have done, we don't try to refine our specification to our code directly: We first introduce an intermediate transition system which is closer to what we want to implement. Then we refine it to code.

lemma
  assumes ‹*no-dup M*›
  shows
    no-dup-same-annotD:
    ‹*Propagated L C* ∈ *set M* ⟹ *Propagated L C'* ∈ *set M* ⟹ *C* = *C'*› **and**
    no-dup-no-propa-and-dec:
    ‹*Propagated L C* ∈ *set M* ⟹ *Decided L* ∈ *set M* ⟹ *False*›
  using *assms*
  by (*auto dest*!: *split-list elim*: *list-match-lel-lel*)

This invariant abstract over the restart operation on the trail. There can be a backtracking on the trail and there can be a renumbering of the indexes.

inductive *valid-trail-reduction* **for** *M M'* :: ‹('*v* ,'*c*) *ann-lits*› **where**
backtrack-red:
  ‹*valid-trail-reduction M M'*›
  if
    ‹(*Decided K* # *M''*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
    ‹*map lit-of M''* = *map lit-of M'*› **and**
    ‹*map is-decided M''* = *map is-decided M'*› |
keep-red:
  ‹*valid-trail-reduction M M'*›
  if
    ‹*map lit-of M* = *map lit-of M'*› **and**
    ‹*map is-decided M* = *map is-decided M'*›

lemma *valid-trail-reduction-simps*: ‹*valid-trail-reduction M M'* ⟷
  ((∃ *K M'' M2*. (*Decided K* # *M''*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
     map lit-of M'' = *map lit-of M'* ∧ *map is-decided M''* = *map is-decided M'* ∧
    length M' = *length M''*) ∨
   map lit-of M = *map lit-of M'* ∧ *map is-decided M* = *map is-decided M'* ∧ *length M* = *length M'*)›
 apply (*auto simp*: *valid-trail-reduction.simps dest*: *arg-cong*[*of* ‹*map lit-of* ›› - *length*])
 apply (*force dest*: *arg-cong*[*of* ‹*map lit-of* ›› - *length*])+
 done

lemma *valid-trail-reduction-refl*[*simp*]:
  ‹*valid-trail-reduction M M*›
  by (*rule valid-trail-reduction.keep-red*)
    auto

lemma *trail-changes-same-decomp*:
  assumes
    M'-lit: ‹*map lit-of M'* = *map lit-of ysa* @ *L* # *map lit-of zsa*› **and**
    M'-dec: ‹*map is-decided M'* = *map is-decided ysa* @ *False* # *map is-decided zsa*›
  obtains *C' M2 M1* **where** ‹*M'* = *M2* @ *Propagated L C'* # *M1*› **and**
    ‹*map lit-of M2* = *map lit-of ysa*›**and**
    ‹*map is-decided M2* = *map is-decided ysa*› **and**
    ‹*map lit-of M1* = *map lit-of zsa*› **and**
    ‹*map is-decided M1* = *map is-decided zsa*›
proof −
  define *M1 M2 K* **where** ‹*M1* ≡ *drop* (*Suc* (*length ysa*)) *M'*› **and** ‹*M2* ≡ *take* (*length ysa*) *M'*› **and**
    ‹*K* ≡ *hd* (*drop* (*length ysa*) *M'*)›
  have

```
    M': ‹M' = M2 @ K # M1›
    using arg-cong[OF M'-lit, of length] unfolding M1-def M2-def K-def
    by (simp add: Cons-nth-drop-Suc hd-drop-conv-nth)
  have [simp]:
    ‹length M2 = length ysa›
    ‹length M1 = length zsa›
    using arg-cong[OF M'-lit, of length] unfolding M1-def M2-def K-def by auto

  obtain C' where
    [simp]: ‹K = Propagated L C'›
    using M'-lit M'-dec unfolding M'
    by (cases K) auto

  show ?thesis
    using that[of M2 C' M1] M'-lit M'-dec unfolding M'
    by auto
qed

lemma
  assumes
    ‹map lit-of M = map lit-of M'› and
    ‹map is-decided M = map is-decided M'›
  shows
    trail-renumber-count-dec:
      ‹count-decided M = count-decided M'› and
    trail-renumber-get-level:
      ‹get-level M L = get-level M' L›
proof −
  have [dest]: ‹count-decided M = count-decided M'›
    if ‹map is-decided M = map is-decided M'› for M M'
    using that
    apply (induction M arbitrary: M' rule: ann-lit-list-induct)
    subgoal by auto
    subgoal for L M M'
      by (cases M')
        (auto simp: get-level-cons-if)
    subgoal for L C M M'
      by (cases M')
        (auto simp: get-level-cons-if)
    done
  then show ‹count-decided M = count-decided M'› using assms by blast
  show ‹get-level M L = get-level M' L›
    using assms
    apply (induction M arbitrary: M' rule: ann-lit-list-induct)
    subgoal by auto
    subgoal for L M M'
      by (cases M'; cases ‹hd M'›)
        (auto simp: get-level-cons-if)
    subgoal for L C M M'
      by (cases M')
        (auto simp: get-level-cons-if)
    done
qed

lemma valid-trail-reduction-Propagated-inD:
  ‹valid-trail-reduction M M' ⟹ Propagated L C ∈ set M' ⟹ ∃ C'. Propagated L C' ∈ set M›
```

```
  by (induction rule: valid-trail-reduction.induct)
    (force dest!: get-all-ann-decomposition-exists-prepend
      dest!: split-list[of ‹Propagated L C›] elim!: trail-changes-same-decomp)+

lemma valid-trail-reduction-Propagated-inD2:
  ‹valid-trail-reduction M M' ⟹ length M = length M' ⟹ Propagated L C ∈ set M ⟹
    ∃ C'. Propagated L C' ∈ set M'›
  apply (induction rule: valid-trail-reduction.induct)
  apply (auto dest!: get-all-ann-decomposition-exists-prepend
      dest!: split-list[of ‹Propagated L C›] elim!: trail-changes-same-decomp)+
    apply (metis add-Suc-right le-add2 length-Cons length-append length-map not-less-eq-eq)
  by (metis (no-types, lifting) in-set-conv-decomp trail-changes-same-decomp)

lemma get-all-ann-decomposition-change-annotation-exists:
  assumes
    ‹(Decided K # M', M2') ∈ set (get-all-ann-decomposition M2)› and
    ‹map lit-of M1 = map lit-of M2› and
    ‹map is-decided M1 = map is-decided M2›
  shows ‹∃ M'' M2'. (Decided K # M'', M2') ∈ set (get-all-ann-decomposition M1) ∧
    map lit-of M'' = map lit-of M' ∧  map is-decided M'' = map is-decided M'›
  using assms
proof (induction M1 arbitrary: M2 M2' rule: ann-lit-list-induct)
  case Nil
  then show ?case by auto
next
  case (Decided L xs M2)
  then show ?case
    by (cases M2; cases ‹hd M2›) fastforce+
next
  case (Propagated L m xs M2) note IH = this(1) and prems = this(2−)
  show ?case
    using IH[of - ‹tl M2›] prems get-all-ann-decomposition-decomp[of xs]
      get-all-ann-decomposition-decomp[of M2 ‹Decided K # M'›]
    by (cases M2; cases ‹hd M2›; cases ‹(get-all-ann-decomposition (tl M2))›;
        cases ‹hd (get-all-ann-decomposition xs)›; cases ‹get-all-ann-decomposition xs›)
      fastforce+
qed

lemma valid-trail-reduction-trans:
  assumes
    M1-M2: ‹valid-trail-reduction M1 M2› and
    M2-M3: ‹valid-trail-reduction M2 M3›
  shows ‹valid-trail-reduction M1 M3›
proof −
  consider
    (same) ‹map lit-of M2 = map lit-of M3› and
      ‹map is-decided M2 = map is-decided M3› ‹length M2 = length M3› |
    (decomp-M1) K M'' M2' where
      ‹(Decided K # M'', M2') ∈ set (get-all-ann-decomposition M2)› and
      ‹map lit-of M'' = map lit-of M3› and
      ‹map is-decided M'' = map is-decided M3› and
      ‹length M3 = length M''›
    using M2-M3 unfolding valid-trail-reduction-simps
    by auto
  note decomp-M2 = this
  consider
```

```
  (same) ‹map lit-of M1 = map lit-of M2› and
    ‹map is-decided M1 = map is-decided M2› ‹length M1 = length M2› |
  (decomp-M1) K M'' M2' where
    ‹(Decided K # M'', M2') ∈ set (get-all-ann-decomposition M1)› and
    ‹map lit-of M'' = map lit-of M2› and
    ‹map is-decided M'' = map is-decided M2› and
    ‹length M2 = length M''›
  using M1-M2 unfolding valid-trail-reduction-simps
  by auto
then show ?thesis
proof cases
  case same
  from decomp-M2
  show ?thesis
  proof cases
    case same': same
    then show ?thesis
      using same by (auto simp: valid-trail-reduction-simps)
  next
    case decomp-M1 note decomp = this(1) and eq = this(2,3) and [simp] = this(4)
    obtain M4 M5 where
      decomp45: ‹(Decided K # M4, M5) ∈ set (get-all-ann-decomposition M1)› and
      M4-lit: ‹map lit-of M4 = map lit-of M''› and
      M4-dec: ‹map is-decided M4 = map is-decided M''›
      using get-all-ann-decomposition-change-annotation-exists[OF decomp, of M1] eq same
      by (auto simp: valid-trail-reduction-simps)
    show ?thesis
      by (rule valid-trail-reduction.backtrack-red[OF decomp45])
        (use M4-lit M4-dec eq same in auto)
  qed
next
  case decomp-M1 note decomp = this(1) and eq = this(2,3) and [simp] = this(4)
  from decomp-M2
  show ?thesis
  proof cases
    case same
    obtain M4 M5 where
      decomp45: ‹(Decided K # M4, M5) ∈ set (get-all-ann-decomposition M1)› and
      M4-lit: ‹map lit-of M4 = map lit-of M''› and
      M4-dec: ‹map is-decided M4 = map is-decided M''›
      using get-all-ann-decomposition-change-annotation-exists[OF decomp, of M1] eq same
      by (auto simp: valid-trail-reduction-simps)
    show ?thesis
      by (rule valid-trail-reduction.backtrack-red[OF decomp45])
        (use M4-lit M4-dec eq same in auto)
  next
   case (decomp-M1 K' M''' M2''') note decomp' = this(1) and eq' = this(2,3) and [simp] = this(4)
    obtain M4 M5 where
      decomp45: ‹(Decided K' # M4, M5) ∈ set (get-all-ann-decomposition M'')› and
      M4-lit: ‹map lit-of M4 = map lit-of M'''› and
      M4-dec: ‹map is-decided M4 = map is-decided M'''›
      using get-all-ann-decomposition-change-annotation-exists[OF decomp', of M''] eq
      by (auto simp: valid-trail-reduction-simps)
    obtain M6 where
      decomp45: ‹(Decided K' # M4, M6) ∈ set (get-all-ann-decomposition M1)›
      using get-all-ann-decomposition-exists-prepend[OF decomp45]
```

```
        get-all-ann-decomposition-exists-prepend[OF decomp]
        get-all-ann-decomposition-ex[of K' M4 ‹- @ M2' @ Decided K # - @ M5›]
      by (auto simp: valid-trail-reduction-simps)
    show ?thesis
      by (rule valid-trail-reduction.backtrack-red[OF decomp45])
        (use M4-lit M4-dec eq decomp-M1 in auto)
  qed
 qed
qed

lemma valid-trail-reduction-length-leD: ‹valid-trail-reduction M M' ⟹ length M' ≤ length M›
 by (auto simp: valid-trail-reduction-simps)

lemma valid-trail-reduction-level0-iff:
 assumes valid: ‹valid-trail-reduction M M'› and n-d: ‹no-dup M›
 shows ‹(L ∈ lits-of-l M ∧ get-level M L = 0) ⟷ (L ∈ lits-of-l M' ∧ get-level M' L = 0)›
proof −
 have H[intro]: ‹map lit-of M = map lit-of M' ⟹ L ∈ lits-of-l M ⟹ L ∈ lits-of-l M'› for M M'
  by (metis lits-of-def set-map)
 have [dest]: ‹undefined-lit c L ⟹ L ∈ lits-of-l c ⟹ False› for c
  by (auto dest: in-lits-of-l-defined-litD)

 show ?thesis
  using valid
 proof cases
  case keep-red
  then show ?thesis
   by (metis H trail-renumber-get-level)
 next
  case (backtrack-red K M'' M2) note decomp = this(1) and eq = this(2,3)
  obtain M3 where M: ‹M = M3 @ Decided K # M''›
   using decomp by auto
  have ‹(L ∈ lits-of-l M ∧ get-level M L = 0) ⟷
   (L ∈ lits-of-l M'' ∧ get-level M'' L = 0)›
   using n-d unfolding M
   by (auto 4 4 simp: valid-trail-reduction-simps get-level-append-if get-level-cons-if
      atm-of-eq-atm-of
     dest: in-lits-of-l-defined-litD no-dup-append-in-atm-notin
     split: if-splits)
  also have ‹... ⟷ (L ∈ lits-of-l M' ∧ get-level M' L = 0)›
   using eq by (metis local.H trail-renumber-get-level)
  finally show ?thesis
   by blast
 qed
qed

lemma map-lit-of-eq-defined-litD: ‹map lit-of M = map lit-of M' ⟹ defined-lit M = defined-lit M'›
 apply (induction M arbitrary: M')
 subgoal by auto
 subgoal for L M M'
  by (cases M'; cases L; cases hd M')
   (auto simp: defined-lit-cons)
 done


lemma map-lit-of-eq-no-dupD: ‹map lit-of M = map lit-of M' ⟹ no-dup M = no-dup M'›
```

apply (*induction M arbitrary*: *M'*)
subgoal by *auto*
subgoal for *L M M'*
by (*cases M'*; *cases L*; *cases hd M'*)
(*auto dest*: *map-lit-of-eq-defined-litD*)
done

Remarks about the predicate:

- The cases $\forall\, L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow E = (0::'b) \longrightarrow E' \neq (0::'c) \longrightarrow P$ are already covered by the invariants (where P means that there is clause which was already present before).
- There is no simple way to express that a reason is in *UE* or not in it. This was not a problem as long we did not empty it, but we had to include that. The only solution is to split the components in two parts: the one in the trail (that stay there forever and count toward the number of clauses) and the authorers that can be deleted.

inductive *cdcl-twl-restart-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
restart-trail:
‹*cdcl-twl-restart-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M'*, *N'*, *None*, *NE* + *mset* ‘# *NE'*, *UE''*, *NEk* + *mset* ‘# *NEk'*, *UEk* + *mset* ‘# *UEk'*, *NS*, *US'*, *N0*, *U0'*, {#}, *Q'*)›
if
‹*valid-trail-reduction M M'*› **and**
‹*init-clss-lf N* = *init-clss-lf N'* + *NE'* + *NEk'*› **and**
‹*learned-clss-lf N'* + *UE'* + *UEk'* ⊆# *learned-clss-lf N*› **and**
‹∀ *E*∈# (*NE'*+*NEk'*+*UE'*+*UEk'*). ∃ *L*∈*set E*. *L* ∈ *lits-of-l M* ∧ *get-level M L* = 0› **and**
‹∀ *L E E'* . *Propagated L E* ∈ *set M'* ⟶ *Propagated L E'* ∈ *set M* ⟶ *E* > 0 ⟶ *E'* > 0 ⟶
E ∈# *dom-m N'* ∧ *N'* ∝ *E* = *N* ∝ *E'*› **and**
‹∀ *L E E'*. *Propagated L E* ∈ *set M'* ⟶ *Propagated L E'* ∈ *set M* ⟶ *E* = 0 ⟶ *E'* ≠ 0 ⟶
mset (*N* ∝ *E'*) ∈# *NEk* + *mset* ‘# *NEk'* + *UEk* + *mset* ‘# *UEk'*› **and**
‹∀ *L E E'*. *Propagated L E* ∈ *set M'* ⟶ *Propagated L E'* ∈ *set M* ⟶ *E'* = 0 ⟶ *E* = 0› **and**
‹0 ∉# *dom-m N'*› **and**
‹*if length M* = *length M' then Q* = *Q' else Q'* = {#}› **and**
‹*US'* = {#}› **and**
‹*UE''* ⊆# *UE* + *mset* ‘# *UE'*› **and**
‹*U0'* = {#}›

lemma *cdcl-twl-restart-l-refl*:
‹*get-conflict-l S* = *None* ⟹ *get-subsumed-learned-clauses-l S* = {#} ⟹
get-learned-clauses0-l S = {#} ⟹
clauses-to-update-l S = {#} ⟹ *twl-list-invs S* ⟹ *no-dup* (*get-trail-l S*) ⟹
cdcl-twl-restart-l S S›
by (*cases S*)
(*auto simp*: *cdcl-twl-restart-l.simps twl-list-invs-def dest*: *no-dup-same-annotD*)

lemma *cdcl-twl-restart-l-list-invs*:
assumes
‹*cdcl-twl-restart-l S T*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs T*›
using *assms*

proof (*induction rule*: *cdcl-twl-restart-l.induct*)
case (*restart-trail M M′ N N′ NE′ NEk′ UE′ UEk′ NEk UEk Q Q′ US′ UE″ UE U0′ NE NS US N0 U0*) **note** *red* = *this*(*1*) **and**
init = *this*(*2*) **and**
learned = *this*(*3*) **and** *NUE* = *this*(*4*) **and** *tr-ge0* = *this*(*5*) **and** *tr-new0* = *this*(*6*) **and**
tr-still0 = *this*(*7*) **and** *dom0* = *this*(*8*) **and** *QQ′* = *this*(*9*) **and** *US* = *this*(*10*) **and** *incl* = *this*(*11*)
and *U0′* = *this*(*12*) **and** *list-invs* = *this*(*13*)
let *?S* = ‹(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0′*, {#}, *Q*)›
let *?T* = ‹(*M′*, *N′*, *None*, *NE* + *mset* ‘# *NE′*, *UE″*, *NEk* + *mset* ‘# *NEk′*, *UEk* + *mset* ‘# *UEk′*, *NS*, *US′*, *N0*, *U0′*, {#}, *Q′*)›
show *?case*
unfolding *twl-list-invs-def*
proof (*intro conjI impI allI ballI*)
fix *C*
assume ‹*C* ∈# *clauses-to-update-l ?T*›
then show ‹*C* ∈# *dom-m* (*get-clauses-l ?T*)›
by *simp*
next
show ‹*0* ∉# *dom-m* (*get-clauses-l ?T*)›
using *dom0* **by** *simp*
next
fix *L C*
assume *LC*: ‹*Propagated L C* ∈ *set* (*get-trail-l ?T*)› **and** *C0*: ‹*0* < *C*›
then obtain *C′* **where** *LC′*: ‹*Propagated L C′* ∈ *set* (*get-trail-l ?S*)›
using *red* **by** (*auto dest*!: *valid-trail-reduction-Propagated-inD*)
moreover have *C′0*: ‹*C′* ≠ *0*›
apply (*rule ccontr*)
using *C0 tr-still0 LC LC′*
by (*auto simp*: *twl-list-invs-def*
dest!: *valid-trail-reduction-Propagated-inD*)
ultimately have *C-dom*: ‹*C* ∈# *dom-m* (*get-clauses-l ?T*)› **and** *NCC′*: ‹*N′* ∝ *C* = *N* ∝ *C′*›
using *tr-ge0 C0 LC* **by** *auto*
show ‹*C* ∈# *dom-m* (*get-clauses-l ?T*)›
using *C-dom* **.**

have
L-watched: ‹*L* ∈ *set* (*watched-l* (*get-clauses-l ?S* ∝ *C′*))› **and**
L-C′0: ‹*length* (*get-clauses-l ?S* ∝ *C′*) > *2* ⟹ *L* = *get-clauses-l ?S* ∝ *C′* ! *0*›
using *list-invs C′0 LC′* **unfolding** *twl-list-invs-def*
by *auto*
show ‹*L* ∈ *set* (*watched-l* (*get-clauses-l ?T* ∝ *C*))›
using *L-watched NCC′* **by** *simp*

show ‹*length* (*get-clauses-l ?T* ∝ *C*) > *2* ⟹ *L* = *get-clauses-l ?T* ∝ *C* ! *0*›
using *L-C′0 NCC′* **by** *simp*
next
show ‹*distinct-mset* (*clauses-to-update-l ?T*)›
by *auto*
next
fix *C*
assume ‹*C* ∈# *ran-mf* (*get-clauses-l ?T*)›
then have ‹*C* ∈# *ran-mf* (*get-clauses-l ?S*)›
by (*smt* (*z3*) *all-clss-lf-ran-m get-clauses-l.simps learned local.init*
mset-subset-eq-exists-conv union-iff)
moreover have ‹∀ *C*∈#*ran-m N*. ¬ *tautology* (*mset* (*fst C*))›
using *list-invs* **unfolding** *twl-list-invs-def*

by *simp*
ultimately show ‹¬*tautology* (*mset C*)›
by *auto*
qed
qed

lemma *rtranclp-cdcl-twl-restart-l-list-invs*:
assumes
‹*cdcl-twl-restart-l*** *S T*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs T*›
using *assms* **by** *induction* (*auto intro*: *cdcl-twl-restart-l-list-invs*)

inductive *cdcl-twl-restart-only-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
restart-trail:
‹*cdcl-twl-restart-only-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M''*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
if
‹(*Decided K* # *M''*, *M'*) ∈ *set* (*get-all-ann-decomposition M*)› |
no-restart:
‹*cdcl-twl-restart-only-l* (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›

lemma *cdcl-twl-restart-only-l-list-invs*:
assumes
‹*cdcl-twl-restart-only-l S T*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs T*›
using *assms*
by (*induction rule*: *cdcl-twl-restart-only-l.induct*)
(*auto simp*: *twl-list-invs-def valid-trail-reduction.simps*)

lemma *cdcl-twl-restart-only-l-cdcl-twl-restart-only*:
assumes
‹*cdcl-twl-restart-only-l S T*› **and**
ST: ‹(*S*, *S'*) ∈ *twl-st-l None*›
shows ‹∃ *T'*. (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-restart-only S' T'*›
apply (*rule-tac x* = ‹(λ(*M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*).
(*drop* (*length* (*get-trail S'*) − *length* (*get-trail-l T*)) *M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*,
literals-to-update-l T)) *S'*›
in *exI*)
using *assms*
apply (*induction rule*: *cdcl-twl-restart-only-l.induct*)
apply (*cases S'*)
subgoal for *K M'' M' M N NE UE NEk UEk NS US N0 U0 Q a b c d e f g h i j k l*
using *convert-lits-l-decomp-ex*(*1*)[*of K M'' M' M* ‹*get-trail S'*› *N* ‹*NEk* + *UEk*›]
by (*auto simp*: *twl-st-l-def convert-lits-l-decomp-ex*
intro: *cdcl-twl-restart-only.intros*(*1*))
subgoal
by (*auto simp*: *twl-st-l-def convert-lits-l-imp-same-length*
intro: *cdcl-twl-restart-only.intros*(*2*))

done

lemma *cdcl-twl-restart-only-l-cdcl-twl-restart-only-spec*:
assumes *ST*: ‹$(S,\ T) \in$ *twl-st-l None*› ‹*twl-list-invs S*›
shows ‹*SPEC* (*cdcl-twl-restart-only-l S*) $\leq \Downarrow \{(S,\ S').\ (S,\ S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$
clauses-to-update-l S $= \{\#\}$ $\wedge$ *get-conflict-l S* $=$ *None*$\}$
(*SPEC* (*cdcl-twl-restart-only T*))›
apply (*rule RES-refine*)
subgoal premises *p* **for** *s*
using *cdcl-twl-restart-only-l-cdcl-twl-restart-only*[*of S s*, *OF p*(*1*)[*simplified*] *assms*(*1*)]
cdcl-twl-restart-only-l-list-invs[*of S s*, *OF p*(*1*)[*simplified*]] *p assms*(*2*)
apply − **apply** *normalize-goal+*
apply (*rule-tac x*= *x* **in** *bexI*)
apply (*auto simp*: *cdcl-twl-restart-only-l.simps*)
done
done

lemma *cdcl-twl-restart-l-cdcl-twl-restart*:
assumes *ST*: ‹$(S,\ T) \in$ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*›
shows ‹*SPEC* (*cdcl-twl-restart-l S*) $\leq \Downarrow \{(S,\ S').\ (S,\ S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$
clauses-to-update-l S $= \{\#\}$ $\wedge$ *get-conflict-l S* $=$ *None*$\}$
(*SPEC* (*cdcl-twl-restart T*))›
proof −
have [*simp*]: ‹*set* (*watched-l x*) $\cup$ *set* (*unwatched-l x*) $=$ *set x*› **for** *x*
by (*metis append-take-drop-id set-append*)
have ‹$\exists\ T'.$ *cdcl-twl-restart T T'* $\wedge$ $(S',\ T') \in$ *twl-st-l None*›
if ‹*cdcl-twl-restart-l S S'*› **for** *S'*
using *that ST struct-invs*
proof (*induction rule*: *cdcl-twl-restart-l.induct*)
case (*restart-trail M M' N N' NE' NEk' UE' UEk' NEk UEk Q Q' US' UE'' UE U0' NE NS US N0 U0*) **note** *red* $=$ *this*(*1*) **and**
init $=$ *this*(*2*) **and**
learned $=$ *this*(*3*) **and** *NUE* $=$ *this*(*4*) **and** *tr-ge0* $=$ *this*(*5*) **and** *tr-new0* $=$ *this*(*6*) **and**
tr-still0 $=$ *this*(*7*) **and** *dom0* $=$ *this*(*8*) **and** *QQ'* $=$ *this*(*9*) **and** *US* $=$ *this*(*10*) **and** *incl* $=$ *this*(*11*)
and *U0* $=$ *this*(*12*) **and** *ST* $=$ *this*(*13*) **and** *struct-invs* $=$ *this*(*14*)
let *?T'* $=$ ‹(*drop* (*length M* − *length M'*) (*get-trail T*), *twl-clause-of* ‘# *init-clss-lf N'*,
twl-clause-of ‘# *learned-clss-lf N'*, *None*, *NE*+*mset* ‘# *NE'*+(*NEk*+*mset*‘#*NEk'*), *UE''*+(*UEk*+*mset*‘#*UEk'*), *NS*, *US'*, *N0*,
U0', $\{\#\}$, *Q'*)›
have [*intro*]: ‹$Q \neq Q' \Longrightarrow Q' = \{\#\}$›
using *QQ'* **by** (*auto split*: *if-splits*)
obtain *TM* **where**
T: ‹*T* $=$ (*TM*, *twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*, *None*,
NE+*NEk*, *UE*+*UEk*, *NS*, *US*, *N0*, *U0*, $\{\#\}$, *Q*)› **and**
M-TM: ‹$(M,\ TM) \in$ *convert-lits-l N* (*NEk*+*UEk*)›
using *ST*
by (*cases T*) (*auto simp*: *twl-st-l-def*)
have ‹*no-dup TM*›
using *struct-invs* **unfolding** *T twl-struct-invs-def pcdcl-all-struct-invs-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: *trail.simps*)
then have *n-d*: ‹*no-dup M*›
using *M-TM* **by** *auto*

```
  have ‹cdcl-twl-restart T ?T'›
    using red
  proof (induction)
    case keep-red
    from arg-cong[OF this(1), of length] have [simp]: ‹length M = length M'› by simp
    have [simp]: ‹Q = Q'›
      using QQ' by simp
    have annot-in-clauses: ‹∀ L E. Propagated L E ∈ set TM ⟶
      E ∈# clauses
            (twl-clause-of `# init-clss-lf N +
              twl-clause-of `# learned-clss-lf N')  +
            (NE + NEk + clauses (twl-clause-of `# (NE' + NEk'))) +
            (UE'' + (UEk + mset `# UEk'))›
    proof (intro allI impI conjI)
      fix L E
      assume ‹Propagated L E ∈ set TM›
      then obtain E' where LE'-M: ‹Propagated L E' ∈ set M› and
        E-E': ‹convert-lit N (NEk+UEk) (Propagated L E') (Propagated L E)›
        using in-convert-lits-lD[OF - M-TM, of ‹Propagated L E›]
        by (auto simp: convert-lit.simps)
      then obtain E'' where LE''-M: ‹Propagated L E'' ∈ set M'›
        using valid-trail-reduction-Propagated-inD2[OF red, of L E'] by auto

      consider
        ‹E' = 0› and ‹E'' = 0› |
        ‹E' > 0› and ‹E'' = 0› and ‹mset (N ∝ E') ∈# NEk + mset `# NEk' + UEk + mset `#
UEk'› |
        ‹E' > 0› and ‹E'' > 0› and ‹E'' ∈# dom-m N'› and ‹N ∝ E' = N' ∝ E''›
        using tr-ge0 tr-new0 tr-still0 LE'-M LE''-M E-E'
        by (cases ‹E''>0›; cases ‹E' > 0›) auto
      then show ‹E ∈# clauses
            (twl-clause-of `# init-clss-lf N +
              twl-clause-of `# learned-clss-lf N') +
            (NE + NEk + clauses (twl-clause-of `# (NE' + NEk'))) +
            (UE'' + (UEk + mset `# UEk'))›
        apply cases
        subgoal
          using E-E' incl tr-still0
          by (auto simp: mset-take-mset-drop-mset' convert-lit.simps)
        subgoal
          using E-E' init
          by (auto simp: mset-take-mset-drop-mset' convert-lit.simps)
        subgoal
          using E-E' init
          by (auto simp: mset-take-mset-drop-mset' convert-lit.simps)
        done
    qed
    have ‹cdcl-twl-restart
      (TM, twl-clause-of `# init-clss-lf N, twl-clause-of `# learned-clss-lf N, None,
        NE + NEk, UE + UEk, NS, US, N0, U0, {#}, Q)
      (TM, twl-clause-of `# init-clss-lf N', twl-clause-of `# learned-clss-lf N', None,
        NE + NEk + (clauses (twl-clause-of `# (NE' + NEk'))), UE'' + (UEk + mset `# UEk'),
        NS, {#}, N0, {#}, {#}, Q)› (is ‹cdcl-twl-restart ?A ?B›)
      apply (rule cdcl-twl-restart.restart-clauses[where UE' = ‹(twl-clause-of `# (UE' + UEk'))›])
      subgoal
        using image-mset-subseteq-mono[OF learned, of twl-clause-of] by (auto simp: ac-simps)
```

```
    subgoal unfolding init image-mset-union by auto
    subgoal using NUE M-TM by auto
    subgoal by (rule annot-in-clauses)
    subgoal using US by (auto split: if-splits)
    subgoal using incl by (auto simp: mset-take-mset-drop-mset')
    done
  moreover have ‹?A = T›
    unfolding T by simp
  moreover have ‹?B = ?T'›
    using US U0
    by (auto simp: T mset-take-mset-drop-mset')
  ultimately show ?case
    by argo
next
  case (backtrack-red K M2 M'') note decomp = this(1)
  have [simp]: ‹length M2 = length M'›
    using arg-cong[OF backtrack-red(2), of length] by simp
  have M-TM: ‹(drop (length M − length M') M, drop (length M − length M') TM) ∈
     convert-lits-l N (NEk+UEk)›
    using M-TM unfolding convert-lits-l-def list-rel-def by auto
  have red: ‹valid-trail-reduction (drop (length M − length M') M) M'›
    using red backtrack-red by (auto simp: valid-trail-reduction.simps)
  have annot-in-clauses: ‹∀ L E. Propagated L E ∈ set (drop (length M − length M') TM) ⟶
    E ∈# clauses
        (twl-clause-of `# init-clss-lf N +
          twl-clause-of `# learned-clss-lf N') +
        (NE + NEk + clauses (twl-clause-of `# (NE' + NEk'))) +
        (UE'' + (UEk + mset `# UEk'))›
  proof (intro allI impI conjI)
    fix L E
    assume ‹Propagated L E ∈ set (drop (length M − length M') TM)›
    then obtain E' where LE'-M: ‹Propagated L E' ∈ set (drop (length M − length M') M)› and
      E-E': ‹convert-lit N (NEk+UEk) (Propagated L E') (Propagated L E)›
      using in-convert-lits-lD[OF - M-TM, of ‹Propagated L E›]
      by (auto simp: convert-lit.simps)
    then have ‹Propagated L E' ∈ set M2›
      using decomp by (auto dest!: get-all-ann-decomposition-exists-prepend)
    then obtain E'' where LE''-M: ‹Propagated L E'' ∈ set M'›
      using valid-trail-reduction-Propagated-inD2[OF red, of L E'] decomp
      by (auto dest!: get-all-ann-decomposition-exists-prepend)
    consider
      ‹E' = 0› and ‹E'' = 0› |
       ‹E' > 0› and ‹E'' = 0› and ‹mset (N ∝ E') ∈# NEk + mset `# NEk' + UEk + mset `#
UEk'› |
      ‹E' > 0› and ‹E'' > 0› and ‹E'' ∈# dom-m N'› and ‹N ∝ E' = N' ∝ E''›
      using tr-ge0 tr-new0 tr-still0 LE'-M LE''-M E-E' decomp
      by (cases ‹E''>0›; cases ‹E' > 0›)
        (auto 5 5 dest!: get-all-ann-decomposition-exists-prepend
        simp: convert-lit.simps)
    then show ‹E ∈# clauses
        (twl-clause-of `# init-clss-lf N +
          twl-clause-of `# learned-clss-lf N') +
        (NE + NEk + clauses (twl-clause-of `# (NE' + NEk'))) +
        (UE'' + (UEk + mset `# UEk'))›
      apply cases
      subgoal
```

using *E-E′ incl*
by (*auto simp*: *mset-take-mset-drop-mset′ convert-lit.simps*)
subgoal
using *E-E′ init*
by (*auto simp*: *mset-take-mset-drop-mset′ convert-lit.simps*)
subgoal
using *E-E′ init*
by (*auto simp*: *mset-take-mset-drop-mset′ convert-lit.simps*)
done
qed
have *lits-of-M2-M′*: ‹*lits-of-l M2* = *lits-of-l M′*›
using *arg-cong*[*OF backtrack-red*(*2*), *of set*] **by** (*auto simp*: *lits-of-def*)
have *lev-M2-M′*: ‹*get-level M2 L* = *get-level M′ L*› **for** *L*
using *trail-renumber-get-level*[*OF backtrack-red*(*2*−*3*)] **by** (*auto simp*:)
have *drop-M-M2*: ‹*drop* (*length M* − *length M′*) *M* = *M2*›
using *backtrack-red*(*1*) **by** *auto*
have *H*: ‹*L* ∈ *lits-of-l* (*drop* (*length M* − *length M′*) *TM*) ∧
get-level (*drop* (*length M* − *length M′*) *TM*) *L* = *0*›
if ‹*L* ∈ *lits-of-l M* ∧ *get-level M L* = *0*› **for** *L*
proof −
have ‹*L* ∈ *lits-of-l M2* ∧ *get-level M2 L* = *0*›
using *decomp that n-d*
by (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
dest: *in-lits-of-l-defined-litD*
simp: *get-level-append-if get-level-cons-if split*: *if-splits*)
then show *?thesis*
using *M-TM*
by (*auto dest*!: *simp*: *drop-M-M2*)
qed

have
‹∃ *M2*. (*Decided K* # *drop* (*length M* − *length M′*) *TM*, *M2*) ∈ *set* (*get-all-ann-decomposition TM*)›
using *convert-lits-l-decomp-ex*[*OF decomp* ‹(*M*, *TM*) ∈ *convert-lits-l N* (*NEk* + *UEk*)›]
‹(*M*, *TM*) ∈ *convert-lits-l N* (*NEk* + *UEk*)›
by (*simp add*: *convert-lits-l-imp-same-length*)
then obtain *TM2* **where** *decomp-TM*:
‹(*Decided K* # *drop* (*length M* − *length M′*) *TM*, *TM2*) ∈ *set* (*get-all-ann-decomposition TM*)›
by *blast*
have ‹*cdcl-twl-restart*
(*TM*, *twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*, *None*,
NE + *NEk*, *UE* + *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)
(*drop* (*length M* − *length M′*) *TM*, *twl-clause-of* ‘# *init-clss-lf N′*,
twl-clause-of ‘# *learned-clss-lf N′*, *None*,
NE + *NEk* + *clauses* (*twl-clause-of* ‘# (*NE′* + *NEk′*)), *UE″* + (*UEk* + *mset*‘#*UEk′*), *NS*, {#}, *N0*, {#},
{#}, {#})› (**is** ‹*cdcl-twl-restart ?A ?B*›)
apply (*rule cdcl-twl-restart.restart-trail*[**where** *UE′* = ‹(*twl-clause-of* ‘# (*UE′*+*UEk′*))›])
apply (*rule decomp-TM*)
subgoal
using *image-mset-subseteq-mono*[*OF learned*, *of twl-clause-of*] **by** (*auto simp*: *ac-simps*)
subgoal unfolding *init image-mset-union* **by** *auto*
subgoal using *NUE M-TM H* **by** *fastforce*
subgoal by (*rule annot-in-clauses*)
subgoal using *incl* **by** (*auto simp*: *mset-take-mset-drop-mset′*)
done

```
    moreover have ‹?A = T›
      unfolding T by auto
    moreover have ‹?B = ?T'›
      using QQ' decomp US U0 unfolding T by (auto simp: mset-take-mset-drop-mset')
    ultimately show ?case
      by argo
  qed
  moreover {
     have ‹(M', drop (length M − length M') TM) ∈ convert-lits-l N' ((NEk+mset‘#NEk') +
(UEk+mset‘#UEk'))›
    proof (rule convert-lits-lI)
      show ‹length M' = length (drop (length M − length M') TM)›
        using M-TM red by (auto simp: valid-trail-reduction.simps T
          dest: convert-lits-l-imp-same-length
          dest!: arg-cong[of ‹map lit-of -› - length] get-all-ann-decomposition-exists-prepend)
      fix i
      assume i-M': ‹i < length M'›
       then have MM'-IM: ‹length M − length M' + i < length M› ‹length M − length M' + i <
length TM›
        using M-TM red by (auto simp: valid-trail-reduction.simps T
          dest: convert-lits-l-imp-same-length
          dest!: arg-cong[of ‹map lit-of -› - length] get-all-ann-decomposition-exists-prepend)
      then have ‹convert-lit N (NEk + UEk) (drop (length M − length M') M ! i)
          (drop (length M − length M') TM ! i)›
        using M-TM list-all2-nthD[of ‹convert-lit N (NEk + UEk)› M TM ‹length M − length M' +
i›] i-M'
        unfolding convert-lits-l-def list-rel-def p2rel-def
        by auto
      moreover
        have ‹lit-of (drop (length M − length M') M!i) = lit-of (M'!i)› and
          ‹is-decided (drop (length M − length M') M!i) = is-decided (M'!i)›
        using red i-M' MM'-IM
        by (auto 5 5 simp:valid-trail-reduction-simps nth-append
          dest: map-eq-nth-eq[of - - - i]
          dest!: get-all-ann-decomposition-exists-prepend)
      moreover have ‹M'!i ∈ set M'›
        using i-M' by auto
      moreover have ‹drop (length M − length M') M!i ∈ set M›
        using MM'-IM by auto
      ultimately show ‹convert-lit N' ((NEk+mset‘#NEk') + (UEk+mset‘#UEk')) (M' ! i)
          (drop (length M − length M') TM ! i)›
        using tr-new0 tr-still0 tr-ge0
        by (cases ‹M'!i›)
         (fastforce simp: convert-lit.simps)+
    qed
     then have ‹((M', N', None, NE + mset ‘# NE', UE'', NEk+mset‘#NEk', UEk+mset‘#UEk',
NS, US', N0, U0', {#}, Q'), ?T')
      ∈ twl-st-l None›
      using M-TM by (auto simp: twl-st-l-def T)
  }
  ultimately show ?case
    by fast
qed
moreover have ‹cdcl-twl-restart-l S S' ⟹ twl-list-invs S'› for S'
  by (rule cdcl-twl-restart-l-list-invs) (use list-invs in fast)+
moreover have ‹cdcl-twl-restart-l S S' ⟹ clauses-to-update-l S' = {#} ∧ get-conflict-l S' = None›
```

for S'
  by (*auto simp*: *cdcl-twl-restart-l.simps*)
 ultimately show *?thesis*
  by (*blast intro*!: *RES-refine*)
qed

We here start the refinement towards an executable version of the restarts. The idea of the restart is the following:

1. We backtrack to level 0. This simplifies further steps (even if it would be better not to do that).
2. We first move all clause annotating a literal to *NE* or *UE*.
3. Now we can safely deleting any remaining learned clauses.
4. Once all that is done, we have to recalculate the watch lists (and can on the way GC the set of clauses).

The key idea of our approach is that each transformation is a proper restart. As restarts can be composed to obtain a single restart, we get a single restart. The modular approach is much nicer to prove, but it also makes it easier to have several different restart paths (with and without GC).

Handle true clauses from the trail

lemma *in-set-mset-eq-in*:
 ‹$i \in set\ A \Longrightarrow mset\ A = B \Longrightarrow i \in\# B$›
 by *fastforce*

Our transformation will be chains of a weaker version of restarts, that don't update the watch lists and only keep partial correctness of it.

lemma *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*:
 assumes
  ST: ‹*cdcl-twl-restart-l S T*› **and** *TU*: ‹*cdcl-twl-restart-l T U*› **and**
  n-d: ‹*no-dup* (*get-trail-l S*)›
 shows ‹*cdcl-twl-restart-l S U*›
 using *assms*
proof −
 obtain $M\ M'\ N\ N'\ NE'\ UE'\ NEk\ UEk\ NEk'\ UEk'\ NE\ UE\ UE''\ NS\ US\ N0\ U0\ Q\ Q'\ W'\ W$ **where**
  S: ‹$S = (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, W, Q)$› **and**
  T: ‹$T = (M', N', None, NE + mset\ `\#\ NE', UE'', NEk + mset`\#NEk', UEk + mset`\#UEk', NS, \{\#\}, N0, \{\#\}, W', Q')$› **and**
  tr-red: ‹*valid-trail-reduction M M'*› **and**
  init: ‹*init-clss-lf N* = *init-clss-lf N'* + *NE'* + *NEk'*› **and**
  learned: ‹*learned-clss-lf N'* + *UE'* + *UEk'* ⊆# *learned-clss-lf N*› **and**
  NUE: ‹$\forall E \in \#\, NE' + NEk' + UE' + UEk'.\ \exists L \in set\ E.\ L \in lits\text{-}of\text{-}l\ M \wedge get\text{-}level\ M\ L = 0$› **and**
  ge0: ‹$\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow 0 < E \longrightarrow 0 < E' \longrightarrow E \in \#\ dom\text{-}m\ N' \wedge N' \propto E = N \propto E'$› **and**
  new0: ‹$\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow E = 0 \longrightarrow E' \neq 0 \longrightarrow mset\ (N \propto E') \in \#\ NEk + mset\ `\#\ NEk' + UEk + mset\ `\#\ UEk'$› **and**
  still0: ‹$\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow E' = 0 \longrightarrow E = 0$› **and**
  dom0: ‹$0 \notin \#\ dom\text{-}m\ N'$› **and**

QQ': ‹*if length M = length M' then Q = Q' else Q' = {#}*› **and**
W: ‹*W = {#}*› **and**
incl: ‹*UE'' ⊆# UE + mset '# UE'*›
using *ST* **unfolding** *cdcl-twl-restart-l.simps*
apply −
apply *normalize-goal+*
by *force*
obtain *M'' N'' NE'' U2E'' Q'' W'' NEk'' UEk'' UE'''* **where**
U: ‹*U = (M'', N'', None, (NE + mset '# NE'') + mset '# NE'', UE''',*
(NEk + mset'#NEk'')+ mset'#NEk'', (UEk + mset'#UEk'')+ mset'#UEk'',NS,
{#}, N0, {#}, W'', Q'')› **and**
tr-red': ‹*valid-trail-reduction M' M''*› **and**
init': ‹*init-clss-lf N' = init-clss-lf N'' + NE'' + NEk''*› **and**
learned': ‹*learned-clss-lf N'' + U2E'' + UEk'' ⊆# learned-clss-lf N'*› **and**
NUE': ‹∀ *E*∈#*NE'' + NEk'' + U2E'' + UEk''*.
∃ *L*∈*set E*.
L ∈ lits-of-l M' ∧
get-level M' L = 0› **and**
ge0': ‹∀ *L E E'*.
Propagated L E ∈ set M'' ⟶
Propagated L E' ∈ set M' ⟶
0 < E ⟶
0 < E' ⟶
E ∈# dom-m N'' ∧ N'' ∝ E = N' ∝ E'› **and**
new0': ‹∀ *L E E'*.
Propagated L E ∈ set M'' ⟶
Propagated L E' ∈ set M' ⟶
E = 0 ⟶
E' ≠ 0 ⟶
mset (N' ∝ E')
∈# (NEk + mset '# NEk') + mset'#NEk'' + (UEk + mset '# UEk')+ mset'#UEk''› **and**
still0': ‹∀ *L E E'*.
Propagated L E ∈ set M'' ⟶
Propagated L E' ∈ set M' ⟶
E' = 0 ⟶ E = 0› **and**
dom0': ‹*0 ∉# dom-m N''*› **and**
Q'Q'': ‹*if length M' = length M'' then Q' = Q'' else Q'' = {#}*› **and**
W': ‹*W' = {#}*› **and**
W'': ‹*W'' = {#}*›
and
incl': ‹*UE''' ⊆# UE'' + mset '# U2E''*›
using *TU* **unfolding** *cdcl-twl-restart-l.simps T* **apply** −
apply *normalize-goal+*
by *blast*
have *U'*: ‹*U = (M'', N'', None, NE + mset '# (NE' + NE''), UE''',*
NEk + mset'#(NEk'+ NEk''), *UEk + mset'#(UEk'+UEk'')*, *NS*, {#},
N0, {#}, *W''*, *Q''*)›
unfolding *U* **by** *simp*
show *?thesis*
unfolding *S U' W W' W''*
apply (*rule cdcl-twl-restart-l.restart-trail*[**where** *UE'*=‹*UE'+U2E''*›])
subgoal using *valid-trail-reduction-trans*[*OF tr-red tr-red'*] **.**
subgoal using *init init'* **by** *auto*
subgoal using *learned learned' subset-mset.dual-order.trans*
by (*smt* (*verit*, *ccfv-threshold*) *add.assoc mset-subset-eq-mono-add-right-cancel union-commute*)
subgoal using *NUE NUE' valid-trail-reduction-level0-iff*[*OF tr-red*] *n-d* **unfolding** *S* **by** *auto*

```
    subgoal using ge0 ge0' tr-red' init learned NUE ge0  still0'
      apply (auto dest: valid-trail-reduction-Propagated-inD)
      apply (blast dest: valid-trail-reduction-Propagated-inD)+
       apply (metis neq0-conv still0' valid-trail-reduction-Propagated-inD)+
      done
    subgoal using new0 new0' tr-red' init learned NUE ge0
      apply (auto dest: valid-trail-reduction-Propagated-inD)
      by (smt neq0-conv valid-trail-reduction-Propagated-inD)
    subgoal using still0 still0' tr-red' by (fastforce dest: valid-trail-reduction-Propagated-inD)
    subgoal using dom0' .
    subgoal using QQ' Q'Q'' valid-trail-reduction-length-leD[OF tr-red]
        valid-trail-reduction-length-leD[OF tr-red']
      by (auto split: if-splits)
    subgoal by auto
    subgoal using incl incl' subset-mset.order-trans by fastforce
    subgoal by (rule refl)
    done
qed

lemma rtranclp-cdcl-twl-restart-l-no-dup:
  assumes
    ST: ‹cdcl-twl-restart-l** S T› and
    n-d: ‹no-dup (get-trail-l S)›
  shows ‹no-dup (get-trail-l T)›
  using assms
  apply (induction rule: rtranclp-induct)
  subgoal by auto
  subgoal
    by (auto simp: cdcl-twl-restart-l.simps valid-trail-reduction-simps
      dest: map-lit-of-eq-no-dupD dest!: no-dup-appendD get-all-ann-decomposition-exists-prepend)
  done

lemma tranclp-cdcl-twl-restart-l-cdcl-is-cdcl-twl-restart-l:
  assumes
    ST: ‹cdcl-twl-restart-l++ S T› and
    n-d: ‹no-dup (get-trail-l S)›
  shows ‹cdcl-twl-restart-l S T›
  using assms
  apply (induction rule: tranclp-induct)
  subgoal by auto
  subgoal
    using cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l
    rtranclp-cdcl-twl-restart-l-no-dup by blast
  done
```

Auxilary definition This definition states that the domain of the clauses is reduced, but the remaining clauses are not changed.

```
definition reduce-dom-clauses where
  ‹reduce-dom-clauses N N' ⟷
     (∀ C. C ∈# dom-m N' ⟶ C ∈# dom-m N ∧ fmlookup N C = fmlookup N' C)›

lemma reduce-dom-clauses-fdrop[simp]: ‹reduce-dom-clauses N (fmdrop C N)›
  using distinct-mset-dom[of N]
  by (auto simp: reduce-dom-clauses-def dest: in-diffD multi-member-split
    distinct-mem-diff-mset)
```

lemma *reduce-dom-clauses-refl*[*simp*]: ‹*reduce-dom-clauses N N*›
by (*auto simp*: *reduce-dom-clauses-def*)

lemma *reduce-dom-clauses-trans*:
‹*reduce-dom-clauses N N'* ⟹ *reduce-dom-clauses N' N'a* ⟹ *reduce-dom-clauses N N'a*›
by (*auto simp*: *reduce-dom-clauses-def*)

definition *valid-trail-reduction-eq* **where**
‹*valid-trail-reduction-eq M M'* ⟷ *valid-trail-reduction M M'* ∧ *length M* = *length M'*›

lemma *valid-trail-reduction-eq-alt-def*:
‹*valid-trail-reduction-eq M M'* ⟷ *map lit-of M* = *map lit-of M'* ∧
map is-decided M = *map is-decided M'*›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*
dest!: *get-all-ann-decomposition-exists-prepend*
dest: *map-eq-imp-length-eq trail-renumber-get-level*)

lemma *valid-trail-reduction-change-annot*:
‹*valid-trail-reduction* (*M* @ *Propagated L C* # *M'*)
(*M* @ *Propagated L 0* # *M'*)›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*)

lemma *valid-trail-reduction-eq-change-annot*:
‹*valid-trail-reduction-eq* (*M* @ *Propagated L C* # *M'*)
(*M* @ *Propagated L 0* # *M'*)›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*)

lemma *valid-trail-reduction-eq-refl*: ‹*valid-trail-reduction-eq M M*›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction-refl*)

lemma *valid-trail-reduction-eq-get-level*:
‹*valid-trail-reduction-eq M M'* ⟹ *get-level M* = *get-level M'*›
by (*intro ext*)
(*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*
dest!: *get-all-ann-decomposition-exists-prepend*
dest: *map-eq-imp-length-eq trail-renumber-get-level*)

lemma *valid-trail-reduction-eq-lits-of-l*:
‹*valid-trail-reduction-eq M M'* ⟹ *lits-of-l M* = *lits-of-l M'*›
apply (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*
dest!: *get-all-ann-decomposition-exists-prepend*
dest: *map-eq-imp-length-eq trail-renumber-get-level*)
apply (*metis image-set lits-of-def*)+
done

lemma *valid-trail-reduction-eq-trans*:
‹*valid-trail-reduction-eq M M'* ⟹ *valid-trail-reduction-eq M' M''* ⟹
valid-trail-reduction-eq M M''›
unfolding *valid-trail-reduction-eq-alt-def*
by *auto*

definition *no-dup-reasons-invs-wl* **where**
‹*no-dup-reasons-invs-wl S* ⟷
(*distinct-mset* (*mark-of* '# *filter-mset* (λ*C*. *is-proped C* ∧ *mark-of C* > *0*) (*mset* (*get-trail-l S*)))›

inductive *different-annot-all-killed* **where**
propa-changed:
‹*different-annot-all-killed N NUE* (*Propagated L C*) (*Propagated L C'*)›
if ‹$C \neq C'$› **and** ‹$C' = 0$› **and**
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{mset}\ (N{\propto}C) \in\# \mathit{NUE}$› |
propa-not-changed:
‹*different-annot-all-killed N NUE* (*Propagated L C*) (*Propagated L C*)› |
decided-not-changed:
‹*different-annot-all-killed N NUE* (*Decided L*) (*Decided L*)›

lemma *different-annot-all-killed-refl*[*iff*]:
‹*different-annot-all-killed N NUE z z* $\longleftrightarrow$ *is-proped z* $\vee$ *is-decided z*›
by (*cases z*) (*auto simp*: *different-annot-all-killed.simps*)

abbreviation *different-annots-all-killed* **where**
‹*different-annots-all-killed N NUE* $\equiv$ *list-all2* (*different-annot-all-killed N NUE*)›

lemma *different-annots-all-killed-refl*:
‹*different-annots-all-killed N NUE M M*›
by (*auto intro*!: *list.rel-refl-strong simp*: *count-decided-0-iff is-decided-no-proped-iff*)

Refinement towards code Once of the first thing we do, is removing clauses we know to be true. We do in two ways:

- along the trail (at level 0); this makes sure that annotations are kept;
- then along the watch list.

This is (obviously) not complete but is faster by avoiding iterating over all clauses. Here are the rules we want to apply for our very limited inprocessing:

inductive *remove-one-annot-true-clause* :: ‹'*v twl-st-l* $\Rightarrow$ '*v twl-st-l* $\Rightarrow$ *bool*› **where**
remove-irred-trail:
‹*remove-one-annot-true-clause* (*M* @ *Propagated L C* # *M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)
(*M* @ *Propagated L 0* # *M'*, *fmdrop C N*, *D*, *NE*, {#}, *add-mset* (*mset* (*N*$\propto$*C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)›
if
‹*get-level* (*M* @ *Propagated L C* # *M'*) *L* = *0*› **and**
‹*C* > *0*› **and**
‹*C* $\in\#$ *dom-m N*› **and**
‹*irred N C*› |
remove-red-trail:
‹*remove-one-annot-true-clause* (*M* @ *Propagated L C* # *M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)
(*M* @ *Propagated L 0* # *M'*, *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *add-mset* (*mset* (*N*$\propto$*C*)) *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)›
if
‹*get-level* (*M* @ *Propagated L C* # *M'*) *L* = *0*› **and**
‹*C* > *0*› **and**
‹*C* $\in\#$ *dom-m N*› **and**
‹$\neg$*irred N C*› |
remove-irred:
‹*remove-one-annot-true-clause* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)
(*M*, *fmdrop C N*, *D*, *add-mset* (*mset* (*N*$\propto$*C*))*NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)›

if
‹$L \in$ *lits-of-l* M› **and**
‹*get-level* $M\ L = 0$› **and**
‹$C \in\#$ *dom-m* N› **and**
‹$L \in$ *set* $(N \propto C)$› **and**
‹*irred* $N\ C$› **and**
‹$\forall\, L.$ *Propagated* $L\ C \notin$ *set* M› |
delete:
‹*remove-one-annot-true-clause* $(M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ W,\ Q)$
$(M,$ *fmdrop* $C\ N,\ D,\ NE,\ \{\#\},\ NEk,\ UEk,\ NS,\ \{\#\},\ N0,\ \{\#\},\ W,\ Q)$›
if
‹$C \in\#$ *dom-m* N› **and**
‹$\neg$*irred* $N\ C$› **and**
‹$\forall\, L.$ *Propagated* $L\ C \notin$ *set* M› |
delete-subsumed:
‹*remove-one-annot-true-clause* $(M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ W,\ Q)$
$(M,\ N,\ D,\ NE,\ \{\#\},\ NEk,\ UEk,\ NS,\ \{\#\},\ N0,\ \{\#\},\ W,\ Q)$›

Remarks:

1. $\forall\, L.$ *Propagated* $L\ C \notin$ *set* M is overkill. However, I am currently unsure how I want to handle it (either as *Propagated* $(N \propto C\ !\ 0)\ C \notin$ *set* M or as "the trail contains only zero anyway").

lemma *Ex-ex-eq-Ex*: ‹$(\exists\, NE'.\ (\exists\, b.\ NE' = \{\#b\#\} \wedge P\ b\ NE') \wedge Q\ NE') \longleftrightarrow$
$(\exists\, b.\ P\ b\ \{\#b\#\} \wedge Q\ \{\#b\#\})$›
by *auto*

lemma *in-set-definedD*:
‹*Propagated* $L'\ C \in$ *set* $M' \Longrightarrow$ *defined-lit* $M'\ L'$›
‹*Decided* $L' \in$ *set* $M' \Longrightarrow$ *defined-lit* $M'\ L'$›
by (*auto simp*: *defined-lit-def*)

lemma (**in** *conflict-driven-clause-learning*$_W$) *trail-no-annotation-reuse*:
assumes
struct-invs: ‹*cdcl*$_W$*-all-struct-inv* S› **and**
LC: ‹*Propagated* $L\ C \in$ *set* (*trail* S)› **and**
LC': ‹*Propagated* $L'\ C \in$ *set* (*trail* S)›
shows $L = L'$
proof −
have
confl: ‹*cdcl*$_W$*-conflicting* S› **and**
n-d: ‹*no-dup* (*trail* S)›
using *struct-invs* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def*
by *fast+*
have *H*: ‹$L = L'$› **if** ‹*trail* S = *ysa* @ *Propagated* $L'\ C$ # *c21* @ *Propagated* $L\ C$ # *zs*›
for *ysa xsa c21 zs L* L'
proof −
have *1*: ‹*c21* @ *Propagated* $L\ C$ # *zs* $\models$*as CNot* (*remove1-mset* $L'\ C$) $\wedge\ L' \in\#\ C$›
using *confl* **unfolding** *cdcl*$_W$*-conflicting-def that*
by (*auto*)
have *that'*: ‹*trail* S = (*ysa* @ *Propagated* $L'\ C$ # *c21*) @ *Propagated* $L\ C$ # *zs*›
unfolding *that* **by** *auto*
have *2*: ‹*zs* $\models$*as CNot* (*remove1-mset* $L\ C$) $\wedge\ L \in\#\ C$›
using *confl* **unfolding** *cdcl*$_W$*-conflicting-def that'*

```
    by blast
  show ‹L = L'›
    using 1 2 n-d unfolding that
    by (auto dest!: multi-member-split
      simp: true-annots-true-cls-def-iff-negation-in-model add-mset-eq-add-mset
      Decided-Propagated-in-iff-in-lits-of-l)
 qed
 show ?thesis
  using H[of - L - L'] H[of - L' - L]
  using split-list[OF LC] split-list[OF LC']
  by (force elim!: list-match-lel-lel)
qed

lemma remove-one-annot-true-clause-cdcl-twl-restart-l:
 assumes
  rem: ‹remove-one-annot-true-clause S T› and
  lst-invs: ‹twl-list-invs S› and
  SS': ‹(S, S') ∈ twl-st-l None› and
  struct-invs: ‹twl-struct-invs S'› and
  confl: ‹get-conflict-l S = None› and
  upd: ‹clauses-to-update-l S = {#}› and
  n-d: ‹no-dup (get-trail-l S)›
 shows ‹cdcl-twl-restart-l S T›
 using assms
proof −
 have dist-N: ‹distinct-mset (dom-m (get-clauses-l S))›
  by (rule distinct-mset-dom)
 then have C-notin-rem: ‹C ∉# remove1-mset C (dom-m (get-clauses-l S))› for C
  by (simp add: distinct-mset-remove1-All)
 have
  ‹∀ C∈#clauses-to-update-l S. C ∈# dom-m (get-clauses-l S)› and
  dom0: ‹0 ∉# dom-m (get-clauses-l S)› and
  annot: ‹⋀L C. Propagated L C ∈ set (get-trail-l S) ⟹
      0 < C ⟹
       (C ∈# dom-m (get-clauses-l S) ∧
       L ∈ set (watched-l (get-clauses-l S ∝ C)) ∧
       (length (get-clauses-l S ∝ C) > 2 ⟶ L = get-clauses-l S ∝ C ! 0))› and
  ‹distinct-mset (clauses-to-update-l S)›
  using lst-invs unfolding twl-list-invs-def apply −
  by fast+

 have struct-S': ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of S')›
  using struct-invs unfolding twl-struct-invs-def pcdcl-all-struct-invs-def state_W-of-def by fast
 show ?thesis
  using rem
 proof (cases rule: remove-one-annot-true-clause.cases)
  case (remove-irred-trail M L C M' N D NE UE NEk UEk NS US N0 U0 W Q) note S = this(1)
and T = this(2) and
   lev-L = this(3) and C0 = this(4) and C-dom = this(5) and irred = this(6)
  have D: ‹D = None› and W: ‹W = {#}›
   using confl upd unfolding S by auto
  have NE: ‹add-mset (mset (N ∝ C)) NE = NE + mset `# {#N∝C#}›
   by simp
  have UE: ‹UE = UE + mset `# {#}›
   by simp
  have new-NUE: ‹∀ E∈#{#} + {#N ∝ C#} + {#} + {#}.
```

```
    ∃ La∈set E.
      La ∈ lits-of-l (M @ Propagated L C # M') ∧
      get-level (M @ Propagated L C # M') La = 0›
   apply (intro ballI impI)
   apply (rule-tac x=L in bexI)
   using lev-L annot[of L -] C0 by (auto simp: S dest: in-set-takeD[of - 2])
  have [simp]: ‹Propagated L E' ∉ set M'› ‹Propagated L E' ∉ set M› for E'
   using n-d lst-invs
   by (auto simp: S twl-list-invs-def
     dest!: split-list[of ‹Propagated L E'› M]
       split-list[of ‹Propagated L E'› M'])
  have [simp]: ‹Propagated L' C ∉ set M'› ‹Propagated L' C ∉ set M› for L'
   using SS' n-d C0 struct-S'
   cdcl_W-restart-mset.trail-no-annotation-reuse[of ‹state_W-of S'› L ‹(mset (N ∝ C))› L']
   apply (auto simp: S twl-st-l-def convert-lits-l-imp-same-length trail.simps
     )
   apply (auto simp: list-rel-append1 list-rel-split-right-iff convert-lits-l-def
     p2rel-def)
   apply (case-tac y)
   apply (auto simp: list-rel-append1 list-rel-split-right-iff defined-lit-convert-lits-l
     simp flip: p2rel-def convert-lits-l-def dest: in-set-definedD(1)[of - - M'])
   apply (auto simp: list-rel-append1 list-rel-split-right-iff convert-lits-l-def
     p2rel-def convert-lit.simps
     dest!: split-list[of ‹Propagated L' C› M']
       split-list[of ‹Propagated L' C› M])
   done
  have propa-MM: ‹Propagated L E ∈ set M ⟹ Propagated L E' ∈ set M ⟹ E=E'› for L E E'
   using n-d
   by (auto simp: S twl-list-invs-def
     dest!: split-list[of ‹Propagated L E› M]
       split-list[of ‹Propagated L E'› M]
       elim!: list-match-lel-lel)
  have propa-M'M': ‹Propagated L E ∈ set M' ⟹ Propagated L E' ∈ set M' ⟹ E=E'› for L E E'
   using n-d
   by (auto simp: S twl-list-invs-def
     dest!: split-list[of ‹Propagated L E› M']
       split-list[of ‹Propagated L E'› M']
       elim!: list-match-lel-lel)
  have propa-MM': ‹Propagated L E ∈ set M ⟹ Propagated L E' ∈ set M' ⟹ False› for L E E'
   using n-d
   by (auto simp: S twl-list-invs-def
     dest!: split-list[of ‹Propagated L E› M]
       split-list[of ‹Propagated L E'› M']
       elim!: list-match-lel-lel)
  have propa-M'-nC-dom: ‹Propagated La E ∈ set M' ⟹ E ≠ C ∧ (E > 0 ⟶ E ∈# dom-m N)›
for La E
   using annot[of La E] unfolding S by auto
  have propa-M-nC-dom: ‹Propagated La E ∈ set M ⟹ E ≠ C ∧ (E > 0 ⟶ E ∈# dom-m N)›
for La E
   using annot[of La E] unfolding S by auto
  have H: ‹add-mset (mset (N ∝ C)) NEk = NEk + mset `# {#N ∝ C#}›
   ‹UEk = UEk + mset `# {#}›
   ‹NE = NE + mset `# {#}›
   by auto
  show ?thesis
   unfolding S T D W NE
```

apply (*subst H*)
apply (*subst*(*2*) *H*(*2*))
apply (*subst*(*2*) *H*(*3*))
apply (*rule cdcl-twl-restart-l.intros*[**where** *UE'*=‹{#}›])
subgoal by (*auto simp*: *valid-trail-reduction-change-annot*)
subgoal using *C-dom irred* **by** *auto*
subgoal using *irred* **by** *auto*
subgoal using *new-NUE* **.**
subgoal
apply (*intro conjI allI impI*)
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*]
unfolding *S* **by** *auto*
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*] *propa-MM'*[*of La E' E*] *propa-M'M'*[*of La E' E*]
unfolding *S* **by** *auto*
done
subgoal
apply (*intro allI impI*)
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*] *propa-MM'*[*of La E' E*] *propa-M'M'*[*of La E' E*]
by *auto*
done
subgoal
apply (*intro allI impI*)
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*] *propa-MM'*[*of La E' E*] *propa-M'M'*[*of La E' E*]
by *auto*
done
subgoal using *dom0* **unfolding** *S* **by** (*auto dest*: *in-diffD*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
next
case (*remove-red-trail M L C M' N D NE UE NEk UEk NS US N0 U0 W Q*) **note** *S* =*this*(*1*) **and** *T* = *this*(*2*) **and**
lev-L = *this*(*3*) **and** *C0* = *this*(*4*) **and** *C-dom* = *this*(*5*) **and** *irred* = *this*(*6*)
have *D*: ‹*D* = *None*› **and** *W*: ‹*W* = {#}›
using *confl upd* **unfolding** *S* **by** *auto*
have *UE*: ‹*add-mset* (*mset* (*N* ∝ *C*)) *UE* = *UE* + *mset* ‘# {#*N*∝*C*#}›
by *simp*
have *NE*: ‹*NE* = *NE* + *mset* ‘# {#}›
by *simp*
have *new-NUE*: ‹∀ *E*∈#{#} + {#} + {#} + {#*N* ∝ *C*#}.
∃ *La*∈*set E*.
La ∈ *lits-of-l* (*M* @ *Propagated L C* # *M'*) ∧
get-level (*M* @ *Propagated L C* # *M'*) *La* = *0*›
apply (*intro ballI impI*)
apply (*rule-tac x*=*L* **in** *bexI*)
using *lev-L annot*[*of L* -] *C0* **by** (*auto simp*: *S dest*: *in-set-takeD*[*of* - *2*])

have [*simp*]: ‹*Propagated L E′ ∉ set M′*› ‹*Propagated L E′ ∉ set M*› **for** *E′*
using *n-d lst-invs*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E′*› *M*]
split-list[*of* ‹*Propagated L E′*› *M′*])
have [*simp*]: ‹*Propagated L′ C ∉ set M′*› ‹*Propagated L′ C ∉ set M*› **for** *L′*
using *SS′ n-d C0 struct-S′*
cdcl$_W$*-restart-mset.trail-no-annotation-reuse*[*of* ‹*state*$_W$*-of S′*› *L* ‹(*mset* (*N* ∝ *C*))› *L′*]
apply (*auto simp*: *S twl-st-l-def convert-lits-l-imp-same-length trail.simps*
)
apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff convert-lits-l-def*
p2rel-def)
apply (*case-tac y*)
apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff defined-lit-convert-lits-l*
simp flip: *p2rel-def convert-lits-l-def dest*: *in-set-definedD*(*1*)[*of - - M′*])
apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff convert-lits-l-def*
p2rel-def convert-lit.simps
dest!: *split-list*[*of* ‹*Propagated L′ C*› *M′*]
split-list[*of* ‹*Propagated L′ C*› *M*])
done
have *propa-MM*: ‹*Propagated L E ∈ set M* ⟹ *Propagated L E′ ∈ set M* ⟹ *E*=*E′*› **for** *L E E′*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E′*› *M*]
elim!: *list-match-lel-lel*)
have *propa-M′M′*: ‹*Propagated L E ∈ set M′* ⟹ *Propagated L E′ ∈ set M′* ⟹ *E*=*E′*› **for** *L E E′*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M′*]
split-list[*of* ‹*Propagated L E′*› *M′*]
elim!: *list-match-lel-lel*)
have *propa-MM′*: ‹*Propagated L E ∈ set M* ⟹ *Propagated L E′ ∈ set M′* ⟹ *False*› **for** *L E E′*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E′*› *M′*]
elim!: *list-match-lel-lel*)
have *propa-M′-nC-dom*: ‹*Propagated La E ∈ set M′* ⟹ *E* ≠ *C* ∧ (*E* > *0* ⟶ *E* ∈# *dom-m N*)›
for *La E*
using *annot*[*of La E*] **unfolding** *S* **by** *auto*
have *propa-M-nC-dom*: ‹*Propagated La E ∈ set M* ⟹ *E* ≠ *C* ∧ (*E* > *0* ⟶ *E* ∈# *dom-m N*)›
for *La E*
using *annot*[*of La E*] **unfolding** *S* **by** *auto*
have *H*: ‹*add-mset* (*mset* (*N* ∝ *C*)) *UEk* = *UEk* + *mset* ‘# {#*N* ∝ *C*#}›
‹*NEk* = *NEk* + *mset* ‘# {#}›
‹*NE* = *NE* + *mset* ‘# {#}›
by *auto*
show *?thesis*
unfolding *S T D W UE*
apply (*subst H*)
apply (*subst*(*2*) *H*(*2*))
apply (*subst*(*2*) *H*(*3*))
apply (*rule cdcl-twl-restart-l.intros*[**where** *UE′*=‹{#}›])
subgoal by (*auto simp*: *valid-trail-reduction-change-annot*)
subgoal using *C-dom irred* **by** *auto*

```
    subgoal using C-dom irred by auto
    subgoal using new-NUE .
    subgoal
      apply (intro conjI allI impI)
      subgoal for La E E'
        using C-notin-rem propa-MM[of La E E'] propa-MM'[of La E E'] propa-M'-nC-dom[of La E]
          propa-M-nC-dom[of La E]
        unfolding S by auto
      subgoal for La E E'
        using C-notin-rem propa-MM[of La E E'] propa-MM'[of La E E'] propa-M'-nC-dom[of La E]
          propa-M-nC-dom[of La E] propa-MM'[of La E' E] propa-M'M'[of La E' E]
        unfolding S by auto
      done
    subgoal
      apply (intro allI impI)
      subgoal for La E E'
        using C-notin-rem propa-MM[of La E E'] propa-MM'[of La E E'] propa-M'-nC-dom[of La E]
          propa-M-nC-dom[of La E] propa-MM'[of La E' E] propa-M'M'[of La E' E]
        by auto
      done
    subgoal
      apply (intro allI impI)
      subgoal for La E E'
        using C-notin-rem propa-MM[of La E E'] propa-MM'[of La E E'] propa-M'-nC-dom[of La E]
          propa-M-nC-dom[of La E] propa-MM'[of La E' E] propa-M'M'[of La E' E]
        by auto
      done
    subgoal using dom0 unfolding S by (auto dest: in-diffD)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
  next
    case (remove-irred L M C N D NE UE NEk UEk NS US N0 U0 W Q) note S =this(1) and T =
this(2) and
      L-M = this(3) and lev-L = this(4) and C-dom = this(5) and watched-L = this(6) and
      irred = this(7) and L-notin-M = this(8)
    have NE: ‹add-mset (mset (N ∝ C)) NE = NE + mset `# {#N∝C#}›
      by simp
    have UE: ‹UE = UE + mset `# {#}›
      by simp
    have D: ‹D = None› and W: ‹W = {#}›
      using confl upd unfolding S by auto
    have new-NUE: ‹∀ E∈#{#N ∝ C#} + {#} + {#} + {#}.
       ∃ La∈set E.
          La ∈ lits-of-l M ∧
          get-level M La = 0›
      apply (intro ballI impI)
      apply (rule-tac x=L in bexI)
      using lev-L annot[of L -] L-M watched-L by (auto simp: S dest: in-set-takeD[of - 2])
    have C0: ‹C > 0›
      using dom0 C-dom unfolding S by (auto dest!: multi-member-split)
    have [simp]: ‹Propagated La C ∉ set M› for La
      using annot[of La C] dom0 n-d L-notin-M C0 unfolding S
      by auto
```

```
    have propa-MM: ‹Propagated L E ∈ set M ⟹ Propagated L E' ∈ set M ⟹ E=E'› for L E E'
      using n-d
      by (auto simp: S twl-list-invs-def
        dest!: split-list[of ‹Propagated L E› M]
          split-list[of ‹Propagated L E'› M]
          elim!: list-match-lel-lel)
    have H: ‹UEk = UEk + mset `# {#}›
      ‹NEk = NEk + mset `# {#}›
      ‹NE = NE + mset `# {#}›
      by auto
    have H: ‹UEk = UEk + mset `# {#}›
      ‹NEk = NEk + mset `# {#}›
      by auto
    show ?thesis
      unfolding S T D W NE
      apply (subst(2) H(1))
      apply (subst(2)H(2))
      apply (rule cdcl-twl-restart-l.intros[where UE'=‹{#}›])
      subgoal by (auto simp: valid-trail-reduction-refl)
      subgoal using C-dom irred by auto
      subgoal using C-dom irred by auto
      subgoal using new-NUE .
      subgoal
        using n-d L-notin-M C-notin-rem annot propa-MM unfolding S by force
      subgoal
        using propa-MM by auto
      subgoal
        using propa-MM by auto
      subgoal using dom0 C-dom unfolding S by (auto dest: in-diffD)
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      done
  next
    case (delete C N M D NE UE NEk UEk NS US N0 U0 W Q) note S = this(1) and T = this(2)
and C-dom = this(3) and
      irred = this(4) and L-notin-M = this(5)
    have D: ‹D = None› and W: ‹W = {#}›
      using confl upd unfolding S by auto
    have NE: ‹NE = NE + mset `# {#}›
      by simp
    have H: ‹UEk = UEk + mset `# {#}›
      ‹NEk = NEk + mset `# {#}›
      by auto
    have propa-MM: ‹Propagated L E ∈ set M ⟹ Propagated L E' ∈ set M ⟹ E=E'› for L E E'
      using n-d
      by (auto simp: S twl-list-invs-def
        dest!: split-list[of ‹Propagated L E› M]
          split-list[of ‹Propagated L E'› M]
          elim!: list-match-lel-lel)
    show ?thesis
      unfolding S T D W
      apply (subst (2) NE)
      apply (subst(2) H(1))
      apply (subst(2)H(2))
```

```
      apply (rule cdcl-twl-restart-l.intros[where UE'=‹{#}›])
      subgoal by (auto simp: valid-trail-reduction-refl)
      subgoal using C-dom irred by auto
      subgoal using C-dom irred by auto
      subgoal by simp
      subgoal
        apply (intro conjI impI allI)
        subgoal for L E E'
          using n-d L-notin-M C-notin-rem annot propa-MM[of L E E'] unfolding S
          by (metis dom-m-fmdrop get-clauses-l.simps get-trail-l.simps in-remove1-msetI)
        subgoal for L E E'
          using n-d L-notin-M C-notin-rem annot propa-MM[of L E E'] unfolding S
          by auto
        done
      subgoal
        using propa-MM by auto
      subgoal
        using propa-MM by auto
      subgoal using dom0 C-dom unfolding S by (auto dest: in-diffD)
      subgoal by auto
      subgoal by auto
      subgoal by auto
      subgoal by auto
      done
  next
    case (delete-subsumed M N D NE UE NEk UEk NS US N0 U0 W Q)
    have ‹cdcl-twl-restart-l (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q)
      (M, N, None, NE + mset `# {#}, {#}, NEk + mset `# {#}, UEk + mset `# {#},
      NS, {#}, N0, {#}, {#}, Q)›
      by (rule cdcl-twl-restart-l.intros)
        (use lst-invs n-d in ‹auto dest: no-dup-same-annotD simp: delete-subsumed twl-list-invs-def›)
    then show ?thesis
      using assms
      unfolding delete-subsumed
      by simp
  qed
qed


lemma is-annot-iff-annotates-first:
  assumes
    ST: ‹(S, T) ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs S› and
    struct-invs: ‹twl-struct-invs T› and
    C0: ‹C > 0›
  shows
    ‹(∃L. Propagated L C ∈ set (get-trail-l S)) ⟷
      ((length (get-clauses-l S ∝ C) > 2 ⟶
        Propagated (get-clauses-l S ∝ C ! 0) C ∈ set (get-trail-l S)) ∧
       ((length (get-clauses-l S ∝ C) ≤ 2 ⟶
    Propagated (get-clauses-l S ∝ C ! 0) C ∈ set (get-trail-l S) ∨
    Propagated (get-clauses-l S ∝ C ! 1) C ∈ set (get-trail-l S))))›
    (is ‹?A ⟷ ?B›)
proof (rule iffI)
  assume ?B
  then show ?A by auto
```

```
next
  assume ?A
  then obtain L where
    LC: ‹Propagated L C ∈ set (get-trail-l S)›
    by blast
  then have
    C: ‹C ∈# dom-m (get-clauses-l S)› and
    L-w: ‹L ∈ set (watched-l (get-clauses-l S ∝ C))› and
    L: ‹length (get-clauses-l S ∝ C) > 2 ⟹ L = get-clauses-l S ∝ C ! 0›
    using list-invs C0 unfolding twl-list-invs-def by blast+
  have ‹twl-st-inv T›
    using struct-invs unfolding twl-struct-invs-def by fast
  then have le2: ‹length (get-clauses-l S ∝ C) ≥ 2›
    using C ST multi-member-split[OF C] unfolding twl-struct-invs-def
    by (cases S; cases T)
      (auto simp: twl-st-inv.simps twl-st-l-def
        image-Un[symmetric])
  show ?B
  proof (cases ‹length (get-clauses-l S ∝ C) > 2›)
    case True
    show ?thesis
      using L True LC by auto
  next
    case False
    then show ?thesis
      using LC le2 L-w
      by (cases ‹get-clauses-l S ∝ C›;
          cases ‹tl (get-clauses-l S ∝ C)›)
        auto
  qed
qed

lemma trail-length-ge2:
  assumes
    ST: ‹(S, T) ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs S› and
    struct-invs: ‹twl-struct-invs T› and
    LaC: ‹Propagated L C ∈ set (get-trail-l S)› and
    C0: ‹C > 0›
  shows
    ‹length (get-clauses-l S ∝ C) ≥ 2›
proof −
  have conv:
  ‹(get-trail-l S, get-trail T) ∈ convert-lits-l (get-clauses-l S) (get-kept-unit-clauses-l S)›
   using ST unfolding twl-st-l-def by auto

  have ‹cdcl_W-restart-mset.cdcl_W-conflicting (state_W-of T)› and
    lev-inv: ‹cdcl_W-restart-mset.cdcl_W-M-level-inv (state_W-of T)›
    using struct-invs unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      pcdcl-all-struct-invs-def state_W-of-def
    by fast+

  have n-d: ‹no-dup (get-trail-l S)›
    using ST lev-inv unfolding cdcl_W-restart-mset.cdcl_W-M-level-inv-def
    by (auto simp: twl-st-l twl-st)
  have
```

```
    C: ‹C ∈# dom-m (get-clauses-l S)›
    using list-invs C0 LaC by (auto simp: twl-list-invs-def all-conj-distrib)
  have ‹twl-st-inv T›
    using struct-invs unfolding twl-struct-invs-def by fast
  then show le2: ‹length (get-clauses-l S ∝ C) ≥ 2›
    using C ST multi-member-split[OF C] unfolding twl-struct-invs-def
    by (cases S; cases T)
      (auto simp: twl-st-inv.simps twl-st-l-def
        image-Un[symmetric])
qed

lemma is-annot-no-other-true-lit:
  assumes
    ST: ‹(S, T) ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs S› and
    struct-invs: ‹twl-struct-invs T› and
    C0: ‹C > 0› and
    LaC: ‹Propagated La C ∈ set (get-trail-l S)› and
    LC: ‹L ∈ set (get-clauses-l S ∝ C)› and
    L: ‹L ∈ lits-of-l (get-trail-l S)›
  shows
    ‹La = L› and
    ‹length (get-clauses-l S ∝ C) > 2 ⟹ L = get-clauses-l S ∝ C ! 0›
proof −
  have conv:
   ‹(get-trail-l S, get-trail T) ∈ convert-lits-l (get-clauses-l S) (get-kept-unit-clauses-l S)›
   using ST unfolding twl-st-l-def by auto

  obtain M2 M1 where
    tr-S: ‹get-trail-l S = M2 @ Propagated La C # M1›
    using split-list[OF LaC] by blast
  then obtain M2' M1' where
    tr-T: ‹get-trail T = M2' @ Propagated La (mset (get-clauses-l S ∝ C)) # M1'› and
    M2: ‹(M2, M2') ∈ convert-lits-l (get-clauses-l S) (get-kept-unit-clauses-l S)› and
    M1: ‹(M1, M1') ∈ convert-lits-l (get-clauses-l S) (get-kept-unit-clauses-l S)›
   using conv C0 by (auto simp: list-all2-append1 list-all2-append2 list-all2-Cons1 list-all2-Cons2
    convert-lits-l-def list-rel-def convert-lit.simps dest!: p2relD)
  have ‹cdcl_W-restart-mset.cdcl_W-conflicting (state_W-of T)› and
    lev-inv: ‹cdcl_W-restart-mset.cdcl_W-M-level-inv (state_W-of T)›
    using struct-invs unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      pcdcl-all-struct-invs-def state_W-of-def
    by fast+
  then have ‹La ∈# mset (get-clauses-l S ∝ C)› and
    ‹M1' ⊨as CNot (remove1-mset La (mset (get-clauses-l S ∝ C)))›
    using tr-T
    unfolding cdcl_W-restart-mset.cdcl_W-conflicting-def
    by (auto 5 5 simp: twl-st twl-st-l)
  then have
    ‹M1 ⊨as CNot (remove1-mset La (mset (get-clauses-l S ∝ C)))›
    using M1 convert-lits-l-true-clss-clss by blast
  then have all-false: ‹−K ∈ lits-of-l (get-trail-l S)›
    if ‹K ∈# remove1-mset La (mset (get-clauses-l S ∝ C))›
    for K
    using that tr-S unfolding true-annots-true-cls-def-iff-negation-in-model
    by (auto dest!: multi-member-split)
  have La0: ‹length (get-clauses-l S ∝ C) > 2 ⟹ La = get-clauses-l S ∝ C ! 0› and
```

```
    C: ‹C ∈# dom-m (get-clauses-l S)› and
    ‹La ∈ set (watched-l (get-clauses-l S ∝ C))›
    using list-invs tr-S C0 by (auto simp: twl-list-invs-def all-conj-distrib)
  have n-d: ‹no-dup (get-trail-l S)›
    using ST lev-inv unfolding cdcl_W-restart-mset.cdcl_W-M-level-inv-def
    by (auto simp: twl-st-l twl-st)
  show ‹La = L›
  proof (rule ccontr)
    assume ‹¬?thesis›
    then have ‹L ∈# remove1-mset La (mset (get-clauses-l S ∝ C))›
      using LC by auto
    from all-false[OF this] show False
      using L n-d by (auto dest: no-dup-consistentD)
  qed
  then show ‹length (get-clauses-l S ∝ C) > 2 ⟹ L = get-clauses-l S ∝ C ! 0›
    using La0 by simp
qed

lemma remove-one-annot-true-clause-cdcl-twl-restart-l2:
  assumes
    rem: ‹remove-one-annot-true-clause S T› and
    lst-invs: ‹twl-list-invs S› and
    confl: ‹get-conflict-l S = None› and
    upd: ‹clauses-to-update-l S = {#}› and
    n-d: ‹(S, T') ∈ twl-st-l None› ‹twl-struct-invs T'›
  shows ‹cdcl-twl-restart-l S T›
proof −
  have n-d: ‹no-dup (get-trail-l S)›
    using n-d unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def
      pcdcl-all-struct-invs-def state_W-of-def
    by (auto simp: twl-st twl-st-l)

  show ?thesis
    apply (rule remove-one-annot-true-clause-cdcl-twl-restart-l[OF - - ‹(S, T') ∈ twl-st-l None›])
    subgoal using rem .
    subgoal using lst-invs .
    subgoal using ‹twl-struct-invs T'› .
    subgoal using confl .
    subgoal using upd .
    subgoal using n-d .
    done
qed

lemma remove-one-annot-true-clause-get-conflict-l:
  ‹remove-one-annot-true-clause S T ⟹ get-conflict-l T = get-conflict-l S›
  by (auto simp: remove-one-annot-true-clause.simps)

lemma rtranclp-remove-one-annot-true-clause-get-conflict-l:
  ‹remove-one-annot-true-clause** S T ⟹ get-conflict-l T = get-conflict-l S›
  by (induction rule: rtranclp-induct) (auto simp: remove-one-annot-true-clause-get-conflict-l)

lemma remove-one-annot-true-clause-clauses-to-update-l:
  ‹remove-one-annot-true-clause S T ⟹ clauses-to-update-l T = clauses-to-update-l S›
  by (auto simp: remove-one-annot-true-clause.simps)
```

lemma *remove-one-annot-true-clause-get-all-mark-of-propagated*:
 ‹*remove-one-annot-true-clause S T* ⟹ *set* (*get-all-mark-of-propagated* (*get-trail-l T*)) ⊆ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∪ {*0*}›
 by (*induction rule*: *remove-one-annot-true-clause.induct*) *auto*

lemma *rtranclp-remove-one-annot-true-clause-get-all-mark-of-propagated*:
 ‹*remove-one-annot-true-clause*** *S T* ⟹ *set* (*get-all-mark-of-propagated* (*get-trail-l T*)) ⊆ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∪ {*0*}›
 by (*induction rule*: *rtranclp-induct*)
  (*blast dest*: *remove-one-annot-true-clause-get-all-mark-of-propagated*)+

lemma *rtranclp-remove-one-annot-true-clause-clauses-to-update-l*:
 ‹*remove-one-annot-true-clause*** *S T* ⟹ *clauses-to-update-l T* = *clauses-to-update-l S*›
 by (*induction rule*: *rtranclp-induct*) (*auto simp*: *remove-one-annot-true-clause-clauses-to-update-l*)

lemma *cdcl-twl-restart-l-invs*:
 assumes *ST*: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
  list-invs: ‹*twl-list-invs S*› **and**
  struct-invs: ‹*twl-struct-invs T*› **and** ‹*cdcl-twl-restart-l S S'*›
 shows ‹∃ *T'*. (*S'*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs S'* ∧
      clauses-to-update-l S' = {#} ∧ *cdcl-twl-restart T T'* ∧ *twl-struct-invs T'*›
 using *cdcl-twl-restart-l-cdcl-twl-restart*[*OF ST list-invs struct-invs*]
 cdcl-twl-restart-twl-struct-invs[*OF - struct-invs*]
 by (*smt RETURN-ref-SPECD RETURN-rule assms*(*4*) *in-pair-collect-simp order-trans*)

lemma *rtranclp-cdcl-twl-restart-l-invs*:
 assumes
  ‹*cdcl-twl-restart-l*** *S S'*› **and**
  ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
  list-invs: ‹*twl-list-invs S*› **and**
  struct-invs: ‹*twl-struct-invs T*› **and**
  ‹*clauses-to-update-l S* = {#}›
 shows ‹∃ *T'*. (*S'*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs S'* ∧
      clauses-to-update-l S' = {#} ∧ *cdcl-twl-restart*** *T T'* ∧ *twl-struct-invs T'*›
 using *assms*(*1*)
 apply (*induction rule*: *rtranclp-induct*)
 subgoal
  using *assms*(*2*−) **apply** − **by** (*rule exI*[*of* - *T*]) *auto*
 subgoal for *T U*
  using *cdcl-twl-restart-l-invs*[*of T* - *U*] *assms*
  by (*meson rtranclp.rtrancl-into-rtrancl*)
 done

lemma *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*:
 assumes
  rem: ‹*remove-one-annot-true-clause*** *S T*› **and**
  lst-invs: ‹*twl-list-invs S*› **and**
  confl: ‹*get-conflict-l S* = *None*› **and**
  upd: ‹*clauses-to-update-l S* = {#}› **and**
  n-d: ‹(*S*, *S'*) ∈ *twl-st-l None*› ‹*twl-struct-invs S'*›
 shows ‹∃ *T'*. *cdcl-twl-restart-l*** *S T* ∧ (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-restart*** *S' T'* ∧
  twl-struct-invs T'›
 using *rem*
proof (*induction*)

```
  case base
  then show ?case
    using assms apply − by (rule-tac x=S' in exI) auto
next
  case (step U V) note st = this(1) and step = this(2) and IH = this(3)
  obtain U' where
    IH: ‹cdcl-twl-restart-l** S U› and
    UT': ‹(U, U') ∈ twl-st-l None› and
    S'U': ‹cdcl-twl-restart** S' U'›
    using IH by blast
  have ‹twl-list-invs U›
    using rtranclp-cdcl-twl-restart-l-list-invs[OF IH lst-invs] .
  have ‹get-conflict-l U = None›
    using rtranclp-remove-one-annot-true-clause-get-conflict-l[OF st] confl
    by auto
  have ‹clauses-to-update-l U = {#}›
    using rtranclp-remove-one-annot-true-clause-clauses-to-update-l[OF st] upd
    by auto
  have ‹twl-struct-invs U'›
      by (metis (no-types, opaque-lifting) ‹cdcl-twl-restart** S' U'›
          cdcl-twl-restart-twl-struct-invs n-d(2) rtranclp-induct)
  have ‹cdcl-twl-restart-l U V›
    apply (rule remove-one-annot-true-clause-cdcl-twl-restart-l2[of - - U'])
    subgoal using step .
    subgoal using ‹twl-list-invs U› .
    subgoal using ‹get-conflict-l U = None› .
    subgoal using ‹clauses-to-update-l U = {#}› .
    subgoal using UT' .
    subgoal using ‹twl-struct-invs U'› .
    done
  moreover obtain V' where
    UT': ‹(V, V') ∈ twl-st-l None› and
    ‹cdcl-twl-restart U' V'› and
    ‹twl-struct-invs V'›
    using cdcl-twl-restart-l-invs[OF UT' - -  ‹cdcl-twl-restart-l U V›] ‹twl-list-invs U›
      ‹twl-struct-invs U'›
    by blast
  ultimately show ?case
    using S'U' IH by fastforce
qed

definition drop-clause-add-move-init :: ‹'v twl-st-l ⇒ nat ⇒ 'v twl-st-l› where
  ‹drop-clause-add-move-init = (λ(M, N, D, NE0, UE, NS, US, N0, U0, Q, W) C.
     (M, fmdrop C N, D, add-mset (mset (N ∝ C)) NE0, UE, NS, {#}, N0, U0, Q, W))›

lemma [simp]:
  ‹get-trail-l (drop-clause-add-move-init V C) = get-trail-l V›
  by (cases V) (auto simp: drop-clause-add-move-init-def)

definition drop-clause :: ‹'v twl-st-l ⇒ nat ⇒ 'v twl-st-l› where
  ‹drop-clause = (λ(M, N, D, NE0, UE, NS, US, N0, U0, Q, W) C.
     (M, fmdrop C N, D, NE0, UE, NS, {#}, N0, U0, Q, W))›

lemma [simp]:
  ‹get-trail-l (drop-clause V C) = get-trail-l V›
  by (cases V) (auto simp: drop-clause-def)
```

definition *remove-all-annot-true-clause-one-imp*
where
‹*remove-all-annot-true-clause-one-imp* = (λ(*C*, *S*). *do* {
    if C ∈# *dom-m* (*get-clauses-l S*) *then*
      if irred (*get-clauses-l S*) *C*
      then RETURN (*drop-clause-add-move-init S C*)
      else RETURN (*drop-clause S C*)
    else do {
      RETURN S
    }
  })›

definition *remove-one-annot-true-clause-imp-inv* **where**
  ‹*remove-one-annot-true-clause-imp-inv S* =
    (λ(*i*, *T*). *remove-one-annot-true-clause*** *S T* ∧ *twl-list-invs S* ∧ *i* ≤ *length* (*get-trail-l S*) ∧
      twl-list-invs S ∧
      clauses-to-update-l S = *clauses-to-update-l T* ∧
      literals-to-update-l S = *literals-to-update-l T* ∧
      get-conflict-l T = *None* ∧
      (∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'*) ∧
      get-conflict-l S = *None* ∧ *clauses-to-update-l S* = {#} ∧
      length (*get-trail-l S*) = *length* (*get-trail-l T*) ∧
      (∀ *j*<*i*. *is-proped* (*rev* (*get-trail-l T*) ! *j*) ∧ *mark-of* (*rev* (*get-trail-l T*) ! *j*) = *0*))›

definition *remove-one-annot-true-clause-one-imp-pre* **where**
  ‹*remove-one-annot-true-clause-one-imp-pre i T* ⟷
    (*twl-list-invs T* ∧ *i* < *length* (*get-trail-l T*) ∧
        twl-list-invs T ∧
        (∃ *S'*. (*T*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'*) ∧
        get-conflict-l T = *None* ∧ *clauses-to-update-l T* = {#})›

definition *replace-annot-l-pre* :: ‹'*v literal* ⇒ *nat* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
‹*replace-annot-l-pre L C S* ⟷
  Propagated L C ∈ *set* (*get-trail-l S*) ∧ *C* > *0* ∧
  (∃ *i*. *remove-one-annot-true-clause-one-imp-pre i S*)›

lemma (**in** −)[*simp*]:
  ‹(*S*, *T*) ∈ *twl-st-l b* ⟹ (λ*x*. *atm-of* (*lit-of x*)) ‘ *set* (*get-trail T*) = (λ*x*. *atm-of* (*lit-of x*)) ‘ *set* (*get-trail-l S*)›
  apply (*cases S*; *cases T*; *cases b*)
  apply (*auto simp*: *twl-st-l-def dest*: *in-convert-lits-lD*)
  apply (*metis* (*mono-tags*, *lifting*) *defined-lit-convert-lits-l defined-lit-map rev-image-eqI*)
  by (*metis* (*mono-tags*, *lifting*) *defined-lit-convert-lits-l defined-lit-map rev-image-eqI*)

lemma [*twl-st-l*,*simp*]:
  ‹(*S*, *T*) ∈ *twl-st-l b* ⟹ *pget-all-init-clss* (*pstate*$_W$ *-of T*) = (*get-all-init-clss-l S*)›
  by (*cases S*; *cases T*; *cases b*)
   (*auto simp*: *twl-st-l-def mset-take-mset-drop-mset' get-init-clss-l-def*)

lemma [*twl-st-l*,*simp*]:
  ‹(*S*, *T*) ∈ *twl-st-l b* ⟹ *pget-all-learned-clss* (*pstate*$_W$ *-of T*) = (*get-all-learned-clss-l S*)›
  by (*cases S*; *cases T*; *cases b*)
   (*auto simp*: *twl-st-l-def mset-take-mset-drop-mset' get-learned-clss-l-def*)

lemma *replace-annot-l-pre-alt-def*:
  ‹*replace-annot-l-pre L C S* ⟷
   (*Propagated L C* ∈ *set* (*get-trail-l S*) ∧ *C* > *0* ∧
   (∃ *i*. *remove-one-annot-true-clause-one-imp-pre i S*)) ∧
   L ∈# *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses-l S* +
     get-subsumed-init-clauses-l S + *get-init-clauses0-l S*)›
   (**is** ‹*?A* ⟷ *?B*›)
proof −
  have ‹*L* ∈# *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses-l S* +
     get-subsumed-init-clauses-l S + *get-init-clauses0-l S*)›
    if *pre*: ‹*replace-annot-l-pre L C S*› **and** *LC*: ‹*Propagated L C* ∈ *set* (*get-trail-l S*)›
  proof −
    obtain *T* **where**
      ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
      struct: ‹*twl-struct-invs T*› **and**
      ‹*get-conflict-l S* = *None*›
      using *pre* **unfolding** *replace-annot-l-pre-def*
        remove-one-annot-true-clause-one-imp-pre-def
      by *fast*
    have *alien*: ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of T*)›
      using *struct* **unfolding** *twl-struct-invs-def*
        cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
      pcdcl-all-struct-invs-def state$_W$*-of-def*
      by *fast*
   moreover have ‹*atm-of L* ∈ *atms-of-ms* (*mset* ‘ *set-mset* (*get-init-clss-l S*)) ∪
      atms-of-mm (*get-unit-init-clauses-l S*) ∪
      atms-of-mm (*get-subsumed-init-clauses-l S*) ∪
      atms-of-mm (*get-init-clauses0-l S*)›
      using *ST LC alien* **unfolding** *cdcl*$_W$*-restart-mset.no-strange-atm-def*
      by (*auto simp*: *twl-st twl-st-l in-all-lits-of-mm-ain-atms-of-iff*
        lits-of-def image-image)
    ultimately show ‹*?thesis*›
      using *ST LC* **unfolding** *cdcl*$_W$*-restart-mset.no-strange-atm-def*
      by (*auto simp*: *twl-st twl-st-l in-all-lits-of-mm-ain-atms-of-iff*
        lits-of-def image-image get-init-clss-l-def)
  qed
  then show ‹*?thesis*›
    by (*auto simp*: *replace-annot-l-pre-def*)
qed

definition *replace-annot-l* :: ‹- ⇒ - ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
  ‹*replace-annot-l L C* =
    (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
      ASSERT(*replace-annot-l-pre L C* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
      RES {(*M'*, *N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*)| *M'*.
       (∃ *M2 M1 C*. *M* = *M2* @ *Propagated L C* # *M1* ∧ *M'* = *M2* @ *Propagated L 0* # *M1*)}
    })›

definition *remove-and-add-cls-l* :: ‹*nat* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
  ‹*remove-and-add-cls-l C* =
    (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
      ASSERT(*C* ∈# *dom-m N*);
      RETURN (*M*, *fmdrop C N*, *D*, *NE*, *UE*,
       (*if irred N C then add-mset* (*mset* (*N*∝*C*)) *else id*) *NEk*,
  (*if* ¬*irred N C then add-mset* (*mset* (*N*∝*C*)) *else id*) *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)

})›

The following progrom removes all clauses that are annotations. However, this is not compatible with special handling of binary clauses, since we want to make sure that they should not been deleted.

definition *remove-one-annot-true-clause-one-imp* :: ‹*nat* ⇒ *'v twl-st-l* ⇒ (*nat* × *'v twl-st-l*) *nres*›
where
‹*remove-one-annot-true-clause-one-imp* = (λ*i S*. *do* {
    ASSERT(*remove-one-annot-true-clause-one-imp-pre i S*);
    ASSERT(*is-proped* ((*rev* (*get-trail-l S*))!*i*));
    (*L*, *C*) ← *SPEC*(λ(*L*, *C*). (*rev* (*get-trail-l S*))!*i* = *Propagated L C*);
    ASSERT(*Propagated L C* ∈ *set* (*get-trail-l S*));
    if C = *0 then RETURN* (*i*+*1*, *S*)
    else do {
      ASSERT(*C* ∈# *dom-m* (*get-clauses-l S*));
 S ← *replace-annot-l L C S*;
 S ← *remove-and-add-cls-l C S*;
      RETURN (*i*+*1*, *S*)
    }
  })›

definition *remove-all-learned-subsumed-clauses* :: ‹*'v twl-st-l* ⇒ (*'v twl-st-l*) *nres*› **where**
‹*remove-all-learned-subsumed-clauses* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
  RETURN (*M*, *N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*))›

lemma *decomp-nth-eq-lit-eq*:
  assumes
    ‹*M* = *M2* @ *Propagated L C'* # *M1*› **and**
    ‹*rev M* ! *i* = *Propagated L C*› **and**
    ‹*no-dup M*› **and**
    ‹*i* < *length M*›
  shows ‹*length M1* = *i*› **and** ‹*C* = *C'*›
proof −
  have [*simp*]: ‹*defined-lit M1* (*lit-of* (*M1* ! *i*))› **if** ‹*i* < *length M1*› **for** *i*
    using *that* **by** (*simp add*: *in-lits-of-l-defined-litD lits-of-def*)
  have[*simp*]: ‹*undefined-lit M2 L* ⟹
      k < *length M2* ⟹
      M2 ! *k* ≠ *Propagated L C*› **for** *k*
    using *defined-lit-def nth-mem* **by** *fastforce*
  have[*simp*]: ‹*undefined-lit M1 L* ⟹
      k < *length M1* ⟹
      M1 ! *k* ≠ *Propagated L C*› **for** *k*
    using *defined-lit-def nth-mem* **by** *fastforce*
  have ‹*M* ! (*length M* − *Suc i*) ∈ *set M*›
    apply (*rule nth-mem*)
    using *assms* **by** *auto*
  from *split-list*[*OF this*] **show** ‹*length M1* = *i*› **and** ‹*C* = *C'*›
    using *assms*
    by (*auto simp*: *nth-append nth-Cons nth-rev split*: *if-splits nat.splits*
      elim!: *list-match-lel-lel*)
qed

lemma
  assumes ‹*remove-one-annot-true-clause-imp-inv S s*› **and**

```
    s[simp]: ‹s = (i, U)›
  shows
    remove-all-learned-subsumed-clauses-cdcl-twl-restart-l:
      ‹remove-all-learned-subsumed-clauses U ≤ SPEC(λU'. cdcl-twl-restart-l U U' ∧
        get-trail-l U = get-trail-l U')› (is ?A) and
    remove-one-annot-true-clause-imp-inv-no-dupD:
      ‹no-dup (get-trail-l U)› and
    remove-one-annot-true-clause-imp-inv-no-dupD2:
      ‹no-dup (get-trail-l S)›
proof −
  obtain x where
    SU: ‹remove-one-annot-true-clause** S U› and
    list-invs: ‹twl-list-invs S› and
    ‹i ≤ length (get-trail-l S)› and
    ‹twl-list-invs S› and
    clss-upd-U: ‹clauses-to-update-l S = clauses-to-update-l U› and
    ‹literals-to-update-l S = literals-to-update-l U› and
    conflU: ‹get-conflict-l U = None› and
    conflS: ‹get-conflict-l S = None› and
    Sx: ‹(S, x) ∈ twl-st-l None› and
    struct-invs: ‹twl-struct-invs x› and
    clss-upd-S: ‹clauses-to-update-l S = {#}› and
    ‹length (get-trail-l S) = length (get-trail-l U)› and
    ‹∀j<i. is-proped (rev (get-trail-l U) ! j) ∧
          mark-of (rev (get-trail-l U) ! j) = 0›
    using assms
    unfolding remove-all-learned-subsumed-clauses-def
      remove-one-annot-true-clause-imp-inv-def prod.case
    by blast
  obtain T' where
    list-invs-U: ‹twl-list-invs U› and
    UT': ‹(U, T') ∈ twl-st-l None› and
    xT': ‹cdcl-twl-restart** x T'›
    using rtranclp-cdcl-twl-restart-l-list-invs[of S U, OF - list-invs]
      rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2[OF SU list-invs
        conflS clss-upd-S Sx struct-invs]
    by auto
  have 1: ‹Propagated L E ∈ set (get-trail-l U) ⟹ 0 < E ⟹ E ∈# dom-m (get-clauses-l U)›
    ‹0 ∉# dom-m (get-clauses-l U)› for E L
    using list-invs-U
    unfolding twl-list-invs-def
    by auto
  have ‹twl-struct-invs T'›
    using rtranclp-cdcl-twl-restart-twl-struct-invs struct-invs xT' by blast
  then show ‹no-dup (get-trail-l U)›
    unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def pcdcl-all-struct-invs-def state_W-of-def
    using UT' by (simp add: twl-st)
  from no-dup-same-annotD[OF this]
  show ?A
    using clss-upd-U conflU 1 unfolding clss-upd-S
    unfolding remove-all-learned-subsumed-clauses-def
    by (refine-rcg)
      (auto simp: cdcl-twl-restart-l.simps)
 show ‹no-dup (get-trail-l S)›
   using Sx struct-invs
```

unfolding *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def pcdcl-all-struct-invs-def state$_W$-of-def
by (*simp add*: *twl-st*)
qed

definition *remove-one-annot-true-clause-imp* :: ‹*'v twl-st-l* ⇒ (*'v twl-st-l*) *nres*›
where
‹*remove-one-annot-true-clause-imp* = (λ*S*. *do* {
k ← *SPEC*(λ*k*. (∃ *M1 M2 K*. (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*get-trail-l* *S*)) ∧
count-decided M1 = *0* ∧ *k* = *length M1*)
∨ (*count-decided* (*get-trail-l S*) = *0* ∧ *k* = *length* (*get-trail-l S*)));
start ← *SPEC* (λ*i*. *i* ≤ *k* ∧ (∀ *j* < *i*. *is-proped* (*rev* (*get-trail-l S*) ! *j*) ∧ *mark-of* (*rev* (*get-trail-l S*) ! *j*) = *0*));
(*i*, *T*) ← *WHILE*$_T$*remove-one-annot-true-clause-imp-inv S*
(λ(*i*, *S*). *i* < *k*)
(λ(*i*, *S*). *remove-one-annot-true-clause-one-imp i S*)
(*start*, *S*);
ASSERT (*remove-one-annot-true-clause-imp-inv S* (*i*, *T*));
remove-all-learned-subsumed-clauses T
})›

lemma *remove-one-annot-true-clause-imp-same-length*:
‹*remove-one-annot-true-clause S T* ⟹ *length* (*get-trail-l S*) = *length* (*get-trail-l T*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*) (*auto simp*:)

lemma *rtranclp-remove-one-annot-true-clause-imp-same-length*:
‹*remove-one-annot-true-clause*** *S T* ⟹ *length* (*get-trail-l S*) = *length* (*get-trail-l T*)›
by (*induction rule*: *rtranclp-induct*) (*auto simp*: *remove-one-annot-true-clause-imp-same-length*)

lemma *remove-one-annot-true-clause-map-is-decided-trail*:
‹*remove-one-annot-true-clause S U* ⟹
map is-decided (*get-trail-l S*) = *map is-decided* (*get-trail-l U*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
auto

lemma *remove-one-annot-true-clause-map-mark-of-same-or-0*:
‹*remove-one-annot-true-clause S U* ⟹
mark-of (*get-trail-l S* ! *i*) = *mark-of* (*get-trail-l U* ! *i*) ∨ *mark-of* (*get-trail-l U* ! *i*) = *0*›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
(*auto simp*: *nth-append nth-Cons split*: *nat.split*)

lemma *remove-one-annot-true-clause-imp-inv-trans*:
‹*remove-one-annot-true-clause-imp-inv S* (*i*, *T*) ⟹ *remove-one-annot-true-clause-imp-inv T U* ⟹
remove-one-annot-true-clause-imp-inv S U›
using *rtranclp-remove-one-annot-true-clause-imp-same-length*[*of S T*]
by (*auto simp*: *remove-one-annot-true-clause-imp-inv-def*)

lemma *rtranclp-remove-one-annot-true-clause-map-is-decided-trail*:
‹*remove-one-annot-true-clause*** *S U* ⟹
map is-decided (*get-trail-l S*) = *map is-decided* (*get-trail-l U*)›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *remove-one-annot-true-clause-map-is-decided-trail*)

lemma *rtranclp-remove-one-annot-true-clause-map-mark-of-same-or-0*:

‹*remove-one-annot-true-clause*** *S U* ⟹
mark-of (*get-trail-l S* ! *i*) = *mark-of* (*get-trail-l U* ! *i*) ∨ *mark-of* (*get-trail-l U* ! *i*) = *0*›
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *remove-one-annot-true-clause-map-mark-of-same-or-0*)

lemma *remove-one-annot-true-clause-map-lit-of-trail*:
‹*remove-one-annot-true-clause S U* ⟹
map lit-of (*get-trail-l S*) = *map lit-of* (*get-trail-l U*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
auto

lemma *rtranclp-remove-one-annot-true-clause-map-lit-of-trail*:
‹*remove-one-annot-true-clause*** *S U* ⟹
map lit-of (*get-trail-l S*) = *map lit-of* (*get-trail-l U*)›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *remove-one-annot-true-clause-map-lit-of-trail*)

lemma *remove-one-annot-true-clause-reduce-dom-clauses*:
‹*remove-one-annot-true-clause S U* ⟹
reduce-dom-clauses (*get-clauses-l S*) (*get-clauses-l U*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
auto

lemma *rtranclp-remove-one-annot-true-clause-reduce-dom-clauses*:
‹*remove-one-annot-true-clause*** *S U* ⟹
reduce-dom-clauses (*get-clauses-l S*) (*get-clauses-l U*)›
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *remove-one-annot-true-clause-reduce-dom-clauses intro*: *reduce-dom-clauses-trans*)

lemma *RETURN-le-RES-no-return*:
‹*f* ≤ *SPEC* (λ*S*. *g S* ∈ Φ) ⟹ *do* {*S* ← *f*; *RETURN* (*g S*)} ≤ *RES* Φ›
by (*cases f*) (*auto simp*: *RES-RETURN-RES*)

lemma *remove-one-annot-true-clause-one-imp-spec*:
assumes
I: ‹*remove-one-annot-true-clause-imp-inv S iT*› **and**
cond: ‹*case iT of* (*i*, *S*) ⇒ *i* < *length* (*get-trail-l S*)› **and**
iT: ‹*iT* = (*i*, *T*)› **and**
proped: ‹*is-proped* (*rev* (*get-trail-l S*) ! *i*)›
shows ‹*remove-one-annot-true-clause-one-imp i T*
≤ *SPEC* (λ*s*′. *remove-one-annot-true-clause-imp-inv S s*′ ∧
(*s*′, *iT*) ∈ *measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*))›
proof −
obtain *M N D NE UE NEk UEk NS US N0 U0 WS Q* **where**
T: ‹*T* = (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases T*)

obtain *x* **where**
ST: ‹*remove-one-annot-true-clause*** *S T*› **and**
‹*twl-list-invs S*› **and**
‹*i* ≤ *length* (*get-trail-l S*)› **and**
‹*twl-list-invs S*› **and**
‹(*S*, *x*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs x*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**

level0: ‹∀ *j*<*i*. *is-proped* (*rev* (*get-trail-l T*) ! *j*)› **and**
mark0: ‹∀ *j*<*i*. *mark-of* (*rev* (*get-trail-l T*) ! *j*) = *0*› **and**
le: ‹*length* (*get-trail-l S*) = *length* (*get-trail-l T*)› **and**
clss-upd: ‹*clauses-to-update-l S* = *clauses-to-update-l T*› **and**
lits-upd: ‹*literals-to-update-l S* = *literals-to-update-l T*›
using *I* **unfolding** *remove-one-annot-true-clause-imp-inv-def iT prod.case* **by** *blast*
then have *list-invs-T*: ‹*twl-list-invs T*›
by (*meson rtranclp-cdcl-twl-restart-l-list-invs*
rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2)
obtain *x'* **where**
Tx': ‹(*T*, *x'*) ∈ *twl-st-l None*› **and**
struct-invs-T: ‹*twl-struct-invs x'*›
using ‹(*S*, *x*) ∈ *twl-st-l None*› ‹*twl-list-invs S*› ‹*twl-struct-invs x*› *confl*
rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2 ST upd **by** *blast*
then have *n-d*: ‹*no-dup* (*get-trail-l T*)›
unfolding *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def
pcdcl-all-struct-invs-def state$_W$-of-def
by (*auto simp*: *twl-st twl-st-l*)

have *D*: ‹*D* = *None*› **and** *WS*: ‹*WS* = {#}›
using *confl upd rtranclp-remove-one-annot-true-clause-clauses-to-update-l*[*OF ST*]
using *rtranclp-remove-one-annot-true-clause-get-conflict-l*[*OF ST*] **unfolding** *T* **by** *auto*
have *lits-of-ST*: ‹*lits-of-l* (*get-trail-l S*) = *lits-of-l* (*get-trail-l T*)›
using *arg-cong*[*OF rtranclp-remove-one-annot-true-clause-map-lit-of-trail*[*OF ST*], *of set*]
by (*simp add*: *lits-of-def*)

have *rem-one-annot-i-T*: ‹*remove-one-annot-true-clause-one-imp-pre i T*›
using *Tx' struct-invs-T level0 cond list-invs-T D WS*
unfolding *remove-one-annot-true-clause-one-imp-pre-def iT T prod.case*
by *fastforce*
have
annot-in-dom: ‹*C* ∈# *dom-m* (*get-clauses-l T*)› (**is** *?annot*)
if
‹*case LC of* (*L*, *C*) ⇒ *rev* (*get-trail-l T*) ! *i* = *Propagated L C*› **and**
‹*LC* = (*L*, *C*)› **and**
‹¬(*C* = *0*)›
for *LC L C*
proof −
have ‹*rev* (*get-trail-l T*)!*i* ∈ *set* (*get-trail-l T*)›
using *list-invs-T assms le* **unfolding** *T*
by (*auto simp*: *twl-list-invs-def rev-nth*)
then show *?annot*
using *list-invs-T that le* **unfolding** *T*
by (*auto simp*: *twl-list-invs-def simp del*: *nth-mem*)
qed
have *replace-annot-l*:
‹*replace-annot-l L C T*
≤ *SPEC*
(λ*Sa*. *do* {
S ← *remove-and-add-cls-l C Sa*;
RETURN (*i* + *1*, *S*)
} ≤ *SPEC*
(λ*s'*. *remove-one-annot-true-clause-imp-inv S s'* ∧
(*s'*, *iT*)
∈ *measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*)))›

```
  if
    rem-one: ‹remove-one-annot-true-clause-one-imp-pre i T› and
    ‹is-proped (rev (get-trail-l T) ! i)› and
    LC-d: ‹case LC of (L, C) ⇒ rev (get-trail-l T) ! i = Propagated L C› and
    LC: ‹LC = (L, C)› and
    LC-T: ‹Propagated L C ∈ set (get-trail-l T)› and
    ‹C ≠ 0› and
    dom: ‹C ∈# dom-m (get-clauses-l T)›
  for LC C L
proof −
  have ‹i < length M›
    using rem-one unfolding remove-one-annot-true-clause-one-imp-pre-def T by auto

  {
    fix M2 Ca M1
    assume M: ‹M = M2 @ Propagated L Ca # M1› and ‹irred N Ca›
    have n-d: ‹no-dup M›
      using n-d unfolding T by auto
    then have [simp]: ‹Ca = C›
      using LC-T
      by (auto simp: T M dest!: in-set-definedD)
    have ‹Ca > 0›
      using that(6) by auto
    let ?U = ‹(M2 @ Propagated L 0 # M1, fmdrop Ca N, D, NE, {#},
       add-mset (mset (N ∝ Ca)) NEk, UEk, NS, {#}, N0, {#}, WS, Q)›

    have lev: ‹get-level (M2 @ Propagated L C # M1) L = 0› and
      M1: ‹length M1 = i›
      using n-d level0 LC-d decomp-nth-eq-lit-eq(1)[OF M
  LC-d[unfolded T get-trail-l.simps LC prod.case]
  n-d ‹i < length M›]
unfolding M T
    apply (auto simp: count-decided-0-iff nth-append nth-Cons is-decided-no-proped-iff
      in-set-conv-nth rev-nth
     split: if-splits)
     by (metis diff-less gr-implies-not0 linorder-neqE-nat nth-rev-alt rev-nth zero-less-Suc)

    have TU: ‹remove-one-annot-true-clause T ?U›
      unfolding T M
apply (rule remove-one-annot-true-clause.remove-irred-trail)
using ‹irred N Ca› ‹Ca > 0› dom lev
by (auto simp: T M)
    moreover {
have ‹length (get-trail-l ?U) = length (get-trail-l T)›
  using TU by (auto simp: remove-one-annot-true-clause.simps T M)
then have ‹j<i ⟹ is-proped (rev (get-trail-l ?U) ! j)› for j
  using arg-cong[OF remove-one-annot-true-clause-map-is-decided-trail[OF TU],
   of ‹λxs. xs ! (length (get-trail-l ?U) − Suc j)›] level0 ‹i < length M›
  by (auto simp: rev-nth T is-decided-no-proped-iff M
    nth-append nth-Cons split: nat.splits)
    }
    moreover {
have ‹length (get-trail-l ?U) = length (get-trail-l T)›
  using TU by (auto simp: remove-one-annot-true-clause.simps T M)
then have ‹j<i ⟹ mark-of (rev (get-trail-l ?U) ! j) = 0› for j
  using remove-one-annot-true-clause-map-mark-of-same-or-0[OF TU,
```

```
    of ‹(length (get-trail-l ?U) − Suc j)›] mark0 ‹i < length M›
  by (auto simp: rev-nth T is-decided-no-proped-iff M
    nth-append nth-Cons split: nat.splits)
    }
    moreover have ‹length (get-trail-l S) = length (get-trail-l ?U)›
using le TU by (auto simp: T M split: if-splits)
    moreover have ‹∃S'. (S, S') ∈ twl-st-l None ∧ twl-struct-invs S'›
      by (rule exI[of - x])
  (use ‹(S, x) ∈ twl-st-l None› ‹twl-struct-invs x› in blast)
    ultimately have 1: ‹remove-one-annot-true-clause-imp-inv S (Suc i, ?U)›
using ‹twl-list-invs S› ‹i ≤ length (get-trail-l S)›
‹(S, x) ∈ twl-st-l None› and
‹twl-struct-invs x› and
‹get-conflict-l S = None› and
‹clauses-to-update-l S = {#}› and
‹∀j<i. is-proped (rev (get-trail-l T) ! j)› and
‹∀j<i. mark-of (rev (get-trail-l T) ! j) = 0› and
le T clss-upd lits-upd ST TU D M1 ‹i < length M›
unfolding remove-one-annot-true-clause-imp-inv-def prod.case
by (auto simp: less-Suc-eq nth-append)
    have 2: ‹length (get-trail-l S) − Suc i < length (get-trail-l S) − i›
      by (simp add: T ‹i < length M› diff-less-mono2 le)
    note 1 2
  }
  moreover {
    fix M2 Ca M1
    assume M: ‹M = M2 @ Propagated L Ca # M1› and ‹¬irred N Ca›
    have n-d: ‹no-dup M›
      using n-d unfolding T by auto
    then have [simp]: ‹Ca = C›
      using LC-T
      by (auto simp: T M dest!: in-set-definedD)
    have ‹Ca > 0›
      using that(6) by auto
    let ?U = ‹(M2 @ Propagated L 0 # M1, fmdrop Ca N, D, NE,
      {#}, NEk, add-mset (mset (N ∝ Ca)) UEk, NS, {#}, N0, {#}, WS, Q)›

    have lev: ‹get-level (M2 @ Propagated L C # M1) L = 0› and
      M1: ‹length M1 = i›
      using n-d level0 LC-d decomp-nth-eq-lit-eq(1)[OF M
  LC-d[unfolded T get-trail-l.simps LC prod.case]
  n-d ‹i < length M›]
unfolding M T
    apply (auto simp: count-decided-0-iff nth-append nth-Cons is-decided-no-proped-iff
      in-set-conv-nth rev-nth
     split: if-splits)
     by (metis diff-less gr-implies-not0 linorder-neqE-nat nth-rev-alt rev-nth zero-less-Suc)

    have TU: ‹remove-one-annot-true-clause T ?U›
      unfolding T M
apply (rule remove-one-annot-true-clause.remove-red-trail)
using ‹¬irred N Ca› ‹Ca > 0› dom lev
by (auto simp: T M)
    moreover {
have ‹length (get-trail-l ?U) = length (get-trail-l T)›
  using TU by (auto simp: remove-one-annot-true-clause.simps T M)
```

then have ‹*j*<*i* ⟹ *is-proped* (*rev* (*get-trail-l ?U*) ! *j*)› **for** *j*
using *arg-cong*[*OF remove-one-annot-true-clause-map-is-decided-trail*[*OF TU*],
of ‹λ*xs*. *xs* ! (*length* (*get-trail-l ?U*) − *Suc j*)›] *level0* ‹*i* < *length M*›
by (*auto simp*: *rev-nth T is-decided-no-proped-iff M*
nth-append nth-Cons split: *nat.splits*)
}
moreover {
have ‹*length* (*get-trail-l ?U*) = *length* (*get-trail-l T*)›
using *TU* **by** (*auto simp*: *remove-one-annot-true-clause.simps T M*)
then have ‹*j*<*i* ⟹ *mark-of* (*rev* (*get-trail-l ?U*) ! *j*) = *0*› **for** *j*
using *remove-one-annot-true-clause-map-mark-of-same-or-0*[*OF TU*,
of ‹(*length* (*get-trail-l ?U*) − *Suc j*)›] *mark0* ‹*i* < *length M*›
by (*auto simp*: *rev-nth T is-decided-no-proped-iff M*
nth-append nth-Cons split: *nat.splits*)
}
moreover have ‹*length* (*get-trail-l S*) = *length* (*get-trail-l ?U*)›
using *le TU* **by** (*auto simp*: *T M split*: *if-splits*)
moreover have ‹∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'*›
by (*rule exI*[*of* - *x*])
(*use* ‹(*S*, *x*) ∈ *twl-st-l None*› ‹*twl-struct-invs x*› **in** *blast*)
ultimately have *1*: ‹*remove-one-annot-true-clause-imp-inv S* (*Suc i*, *?U*)›
using ‹*twl-list-invs S*› ‹*i* ≤ *length* (*get-trail-l S*)›
‹(*S*, *x*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs x*› **and**
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}› **and**
‹∀ *j*<*i*. *is-proped* (*rev* (*get-trail-l T*) ! *j*)› **and**
‹∀ *j*<*i*. *mark-of* (*rev* (*get-trail-l T*) ! *j*) = *0*› **and**
le T clss-upd lits-upd ST TU D cond ‹*i* < *length M*› *M1*
unfolding *remove-one-annot-true-clause-imp-inv-def prod.case*
by (*auto simp*: *less-Suc-eq nth-append*)
have *2*: ‹*length* (*get-trail-l S*) − *Suc i* < *length* (*get-trail-l S*) − *i*› **by** (*simp add*: *T* ‹*i* < *length M*› *diff-less-mono2 le*)
note *1 2*
}
moreover have ‹*C* = *Ca*› **if** ‹*M* = *M2* @ *Propagated L Ca* # *M1*› **for** *M1 M2 Ca*
using *LC-T n-d*
by (*auto simp*: *T that dest*!: *in-set-definedD*)
moreover have ‹*replace-annot-l-pre L C* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
using *LC-T that* **unfolding** *replace-annot-l-pre-def*
by (*auto simp*: *T*)
ultimately show *?thesis*
using *dom cond*
by (*auto simp*: *remove-and-add-cls-l-def*
replace-annot-l-def T iT
intro!: *RETURN-le-RES-no-return ASSERT-leI*)
qed

have *rev-set*: ‹*rev* (*get-trail-l T*) ! *i* ∈ *set* (*get-trail-l T*)›
using *assms*
by (*metis length-rev nth-mem rem-one-annot-i-T*
remove-one-annot-true-clause-one-imp-pre-def set-rev)
show *?thesis*
unfolding *remove-one-annot-true-clause-one-imp-def*
apply *refine-vcg*

subgoal using *rem-one-annot-i-T* **unfolding** *iT T* **by** *simp*
subgoal using *proped I le*
rtranclp-remove-one-annot-true-clause-map-is-decided-trail[*of S T*,
THEN arg-cong, *of* ‹λ*xs*. (*rev xs*) ! *i*›]
unfolding *iT T remove-one-annot-true-clause-imp-inv-def*
remove-one-annot-true-clause-one-imp-pre-def
by (*auto simp add*: *All-less-Suc rev-map is-decided-no-proped-iff*)
subgoal
using *rev-set* **unfolding** *T*
by *auto*
subgoal using *I le* **unfolding** *iT T remove-one-annot-true-clause-imp-inv-def*
remove-one-annot-true-clause-one-imp-pre-def
by (*auto simp add*: *All-less-Suc*)
subgoal using *cond le* **unfolding** *iT T remove-one-annot-true-clause-one-imp-pre-def* **by** *auto*
subgoal by (*rule annot-in-dom*)
subgoal for *LC L C*
by (*rule replace-annot-l*)
done

qed

lemma *remove-one-annot-true-clause-count-dec*: ‹*remove-one-annot-true-clause S b* ⟹
count-decided (*get-trail-l S*) = *count-decided* (*get-trail-l b*)›
by (*auto simp*: *remove-one-annot-true-clause.simps*)

lemma *rtranclp-remove-one-annot-true-clause-count-dec*:
‹*remove-one-annot-true-clause*** *S b* ⟹
count-decided (*get-trail-l S*) = *count-decided* (*get-trail-l b*)›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *remove-one-annot-true-clause-count-dec*)

lemma *valid-trail-reduction-count-dec-ge*:
‹*valid-trail-reduction M M'* ⟹
count-decided M' ≤ *count-decided M*›
apply (*induction rule*: *valid-trail-reduction.induct*)
by (*auto simp*: *dest*!: *get-all-ann-decomposition-exists-prepend*
dest: *trail-renumber-count-dec*)

lemma (**in** −)*cdcl-twl-restart-l-count-dec*:
‹*cdcl-twl-restart-l S b* ⟹
count-decided (*get-trail-l S*) ≥ *count-decided* (*get-trail-l b*)›
by (*induction rule*: *cdcl-twl-restart-l.induct*)
(*auto simp*: *remove-one-annot-true-clause-count-dec*
dest: *valid-trail-reduction-count-dec-ge*)

lemma *remove-one-annot-true-clause-imp-spec*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}›
shows ‹*remove-one-annot-true-clause-imp S* ≤ *SPEC*(λ*T*. *cdcl-twl-restart-l S T*)›
unfolding *remove-one-annot-true-clause-imp-def*
apply (*refine-vcg WHILEIT-rule*[**where** *R*=‹*measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*)› **and**

```
    I=‹remove-one-annot-true-clause-imp-inv S›]
   remove-all-learned-subsumed-clauses-cdcl-twl-restart-l[THEN order-trans])
  subgoal by auto
  subgoal using assms unfolding remove-one-annot-true-clause-imp-inv-def prod.simps apply −
   apply (intro conjI)
   apply (solves auto)+
   apply fast+
   done
  apply (rule remove-one-annot-true-clause-one-imp-spec[of - - ])
  subgoal unfolding remove-one-annot-true-clause-imp-inv-def by auto
  subgoal unfolding remove-one-annot-true-clause-imp-inv-def by auto
  subgoal
   by (auto dest!: get-all-ann-decomposition-exists-prepend
     simp: count-decided-0-iff rev-nth is-decided-no-proped-iff)
  subgoal
   by (auto dest!: get-all-ann-decomposition-exists-prepend
     simp: count-decided-0-iff rev-nth is-decided-no-proped-iff)
  subgoal by auto
  apply assumption
  apply assumption
  subgoal for x start s a b xa
   using tranclp-cdcl-twl-restart-l-cdcl-is-cdcl-twl-restart-l[of S xa]
    rtranclp-into-tranclp1[of cdcl-twl-restart-l S b xa]
    remove-one-annot-true-clause-imp-inv-no-dupD2[of S s ‹fst s› ‹snd s›]
   by (auto simp: remove-one-annot-true-clause-imp-inv-def
     dest!: rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2)
  done

lemma remove-one-annot-true-clause-imp-spec-lev0:
  assumes
   ST: ‹(S, T) ∈ twl-st-l None› and
   list-invs: ‹twl-list-invs S› and
   struct-invs: ‹twl-struct-invs T› and
   ‹get-conflict-l S = None› and
   ‹clauses-to-update-l S = {#}› and
   ‹count-decided (get-trail-l S) = 0›
  shows ‹remove-one-annot-true-clause-imp S ≤ SPEC(λT. cdcl-twl-restart-l S T ∧
    count-decided (get-trail-l T) = 0 ∧ (∀ L ∈ set (get-trail-l T). mark-of L = 0) ∧
    length (get-trail-l S) = length (get-trail-l T)) ›
proof −
  have H: ‹∀ j<a. is-proped (rev (get-trail-l b) ! j) ∧
        mark-of (rev (get-trail-l b) ! j) = 0 ⟹ ¬ a < length (get-trail-l b) ⟹
    ∀ x ∈ set (get-trail-l b). is-proped x ∧ mark-of x = 0› for a b
   apply (rule ballI)
   apply (subst (asm) set-rev[symmetric])
   apply (subst (asm) in-set-conv-nth)
   apply auto
   done
  have K: ‹a < length (get-trail-l b) ⟹ is-decided (get-trail-l b ! a) ⟹
    count-decided (get-trail-l b) ≠ 0› for a b
   using count-decided-0-iff nth-mem by blast
  show ?thesis
   apply (rule SPEC-rule-conjI)
   apply (rule remove-one-annot-true-clause-imp-spec[OF assms(1−5)])
   unfolding remove-one-annot-true-clause-imp-def
   apply (refine-vcg WHILEIT-rule[where
```

```
      R=‹measure (λ(i, -::'a twl-st-l). length (get-trail-l S) − i)› and
      I=‹remove-one-annot-true-clause-imp-inv S›]
    remove-one-annot-true-clause-one-imp-spec
    remove-all-learned-subsumed-clauses-cdcl-twl-restart-l[THEN order-trans])
  subgoal by auto
  subgoal using assms unfolding remove-one-annot-true-clause-imp-inv-def prod.simps apply − by
(intro conjI) ((solves auto)+, fast, (solves auto)+)
  subgoal using assms unfolding remove-one-annot-true-clause-imp-inv-def by auto
  subgoal using assms by (auto simp: count-decided-0-iff is-decided-no-proped-iff
    rev-nth)
  subgoal by auto
  apply assumption
  apply assumption
  subgoal using assms unfolding remove-one-annot-true-clause-imp-inv-def
    apply (auto simp: rtranclp-remove-one-annot-true-clause-count-dec
      dest: cdcl-twl-restart-l-count-dec)
    done
  subgoal
    using assms(6) unfolding remove-one-annot-true-clause-imp-inv-def
    by (auto dest: H K)
  subgoal
    using assms(6) unfolding remove-one-annot-true-clause-imp-inv-def
    by (auto dest: H K)
 done
qed


definition collect-valid-indices :: ‹- ⇒ nat list nres› where
 ‹collect-valid-indices S = SPEC (λN. True)›

definition mark-to-delete-clauses-l-inv
 :: ‹'v twl-st-l ⇒ nat list ⇒ nat × 'v twl-st-l × nat list ⇒ bool›
where
 ‹mark-to-delete-clauses-l-inv = (λS xs0 (i, T, xs).
    remove-one-annot-true-clause** S T ∧
    get-trail-l S = get-trail-l T ∧
    (∃S'. (S, S') ∈ twl-st-l None ∧ twl-struct-invs S') ∧
    twl-list-invs S ∧
    get-conflict-l S = None ∧
    clauses-to-update-l S = {#})›

definition mark-to-delete-clauses-l-pre
 :: ‹'v twl-st-l ⇒ bool›
where
 ‹mark-to-delete-clauses-l-pre S ⟷
  (∃T. (S, T) ∈ twl-st-l None ∧ twl-struct-invs T ∧ twl-list-invs S)›

definition mark-garbage-l:: ‹nat ⇒ 'v twl-st-l ⇒ 'v twl-st-l› where
 ‹mark-garbage-l = (λC (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q).
 (M, fmdrop C N, D, NE, {#}, NEk, UEk, NS, {#}, N0, {#}, WS, Q))›

definition can-delete where
 ‹can-delete S C b = (b ⟶
   (length (get-clauses-l S ∝ C) = 2 ⟶
     (Propagated (get-clauses-l S ∝ C ! 0) C ∉ set (get-trail-l S)) ∧
     (Propagated (get-clauses-l S ∝ C ! 1) C ∉ set (get-trail-l S))) ∧
```

```
    (length (get-clauses-l S ∝ C) > 2 ⟶
      (Propagated (get-clauses-l S ∝ C ! 0) C ∉ set (get-trail-l S))) ∧
    ¬irred (get-clauses-l S) C)›

definition mark-to-delete-clauses-l :: ‹'v twl-st-l ⇒ 'v twl-st-l nres› where
‹mark-to-delete-clauses-l = (λS. do {
    ASSERT(mark-to-delete-clauses-l-pre S);
    xs ← collect-valid-indices S;
    to-keep ← SPEC(λ-::nat. True); — the minimum number of clauses that should be kept.
    (-, S, -) ← WHILE_T^mark-to-delete-clauses-l-inv S xs
      (λ(i, S, xs). i < length xs)
      (λ(i, S, xs). do {
        if(xs!i ∉# dom-m (get-clauses-l S)) then RETURN (i, S, delete-index-and-swap xs i)
        else do {
          ASSERT(0 < length (get-clauses-l S∝(xs!i)));
          ASSERT (xs ! i ≠ 0);
          can-del ← SPEC (can-delete S (xs!i));
          ASSERT(i < length xs);
          if can-del
          then
            RETURN (i, mark-garbage-l (xs!i) S, delete-index-and-swap xs i)
          else
            RETURN (i+1, S, xs)
       }
      })
      (to-keep, S, xs);
    remove-all-learned-subsumed-clauses S
  })›


lemma mark-to-delete-clauses-l-spec:
  assumes
    ST: ‹(S, S') ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs S› and
    struct-invs: ‹twl-struct-invs S'› and
    confl: ‹get-conflict-l S = None› and
    upd: ‹clauses-to-update-l S = {#}›
  shows ‹mark-to-delete-clauses-l S ≤ ⇓ Id (SPEC(λT. remove-one-annot-true-clause^++ S T ∧
    get-trail-l S = get-trail-l T))›
proof −

  define I where
    ‹I (xs :: nat list) ≡ (λ(i :: nat, T, xs :: nat list). remove-one-annot-true-clause** S T)› for xs

  have mark0: ‹mark-to-delete-clauses-l-pre S›
    using ST list-invs struct-invs unfolding mark-to-delete-clauses-l-pre-def
    by blast
  have I0: ‹I xs (l, S, xs')›
    for xs xs' :: ‹nat list› and l :: nat
  proof −
    show ?thesis
      unfolding I-def
      by auto
  qed
  have mark-to-delete-clauses-l-inv-notin:
    ‹mark-to-delete-clauses-l-inv S xs0 (a, aa, delete-index-and-swap xs' a)›
```

```
  if
    ‹mark-to-delete-clauses-l-pre S› and
    ‹xs0 ∈ {N. True}› and
    ‹mark-to-delete-clauses-l-inv S xs0 s› and
    ‹I xs0 s› and
    ‹case s of (i, S, xs) ⇒ i < length xs› and
    ‹aa' = (aa, xs')› and
    ‹s = (a, aa')› and
    ‹ba ! a ∉# dom-m (get-clauses-l aa)›
  for s a aa ba xs0 aa' xs'
proof −
  show ?thesis
    using that
    unfolding mark-to-delete-clauses-l-inv-def
    by auto
qed
have I-notin: ‹I xs0 (a, aa, delete-index-and-swap xs' a)›
  if
    ‹mark-to-delete-clauses-l-pre S› and
    ‹xs0 ∈ {N. True}› and
    ‹mark-to-delete-clauses-l-inv S xs0 s› and
    ‹I xs0 s› and
    ‹case s of (i, S, xs) ⇒ i < length xs› and
    ‹aa' = (aa, xs')› and
    ‹s = (a, aa')› and
    ‹ba ! a ∉# dom-m (get-clauses-l aa)›
  for s a aa ba xs0 aa' xs'
proof −
  show ?thesis
    using that
    unfolding I-def
    by auto
qed

have length-ge0: ‹0 < length (get-clauses-l aa ∝ (xs ! a))›
  if
    inv: ‹mark-to-delete-clauses-l-inv S xs0 s› and
    I: ‹I xs0 s› and
    cond: ‹case s of (i, S, xs0) ⇒ i < length xs0› and
    st:
      ‹aa' = (aa, xs)›
      ‹s = (a, aa')› and
    xs: ‹¬ xs ! a ∉# dom-m (get-clauses-l aa)›
  for s a b aa xs0 aa' xs
proof −
  have
    rem: ‹remove-one-annot-true-clause** S aa›
    using I unfolding I-def st prod.case by blast+
  then obtain T' where
    T': ‹(aa, T') ∈ twl-st-l None› and
    ‹twl-struct-invs T'›
    using rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2[OF rem list-invs confl upd
     ST struct-invs] by blast
  then have ‹Multiset.Ball (get-clauses T') struct-wf-twl-cls›
    unfolding twl-struct-invs-def twl-st-inv-alt-def
    by fast
```

```
  then have ‹∀ x∈#ran-m (get-clauses-l aa). 2 ≤ length (fst x)›
    using xs T' by (auto simp: twl-st-l)
  then show ?thesis
    using xs by (auto simp: ran-m-def)
qed

have mark-to-delete-clauses-l-inv-del:
    ‹mark-to-delete-clauses-l-inv S xs0 (i, mark-garbage-l (xs ! i) T, delete-index-and-swap xs i)› and
  I-del: ‹I xs0 (i, mark-garbage-l (xs ! i) T, delete-index-and-swap xs i)›
  if
    ‹mark-to-delete-clauses-l-pre S› and
    ‹xs0 ∈ {N. True}› and
    inv: ‹mark-to-delete-clauses-l-inv S xs0 s› and
    I: ‹I xs0 s› and
    i-le: ‹case s of (i, S, xs) ⇒ i < length xs› and
    st: ‹sT = (T, xs)›
      ‹s = (i, sT)› and
    in-dom: ‹¬ xs ! i ∉# dom-m (get-clauses-l T)› and
    ‹0 < length (get-clauses-l T ∝ (xs ! i))› and
    can-del: ‹can-delete T (xs ! i) b› and
    ‹i < length xs› and
    [simp]: ‹b›
   for x s i T b xs0 sT xs
proof −
  obtain M N D NE UE NEk UEk NS US N0 U0 WS Q where
    S: ‹S = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q)›
    by (cases S)
  obtain M' N' D' NE' UE' NEk' UEk' NS' US' N0' U0' WS' Q' where
    T: ‹T = (M', N', D', NE', UE', NEk', UEk', NS', US', N0', U0', WS', Q')›
    by (cases T)
  have
    rem: ‹remove-one-annot-true-clause** S T›
    using I unfolding I-def st prod.case by blast+

  obtain V where
    SU: ‹cdcl-twl-restart-l** S T› and
    UV: ‹(T, V) ∈ twl-st-l None› and
    TV: ‹cdcl-twl-restart** S' V› and
    struct-invs-V: ‹twl-struct-invs V›
    using rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2[OF rem list-invs confl upd
      ST struct-invs]
    by auto
  have list-invs-U': ‹twl-list-invs T›
    using SU list-invs rtranclp-cdcl-twl-restart-l-list-invs by blast

  have ‹xs ! i > 0›
    apply (rule ccontr)
    using in-dom list-invs-U' unfolding twl-list-invs-def by (auto dest: multi-member-split)
  have ‹N' ∝ (xs ! i) ! 0 ∈ lits-of-l M'›
    if ‹Propagated (N' ∝ (xs ! i) ! 0) (xs0 ! i) ∈ set M'›
    using that by (auto dest!: split-list)
  then have not-annot: ‹Propagated Laa (xs ! i) ∈ set M' ⟹ False› for Laa
    using is-annot-iff-annotates-first[OF UV list-invs-U' struct-invs-V ‹xs ! i > 0›]
    is-annot-no-other-true-lit[OF UV list-invs-U' struct-invs-V ‹xs ! i > 0›, of Laa ‹
      N' ∝ (xs !i) ! 0›] can-del
    trail-length-ge2[OF UV list-invs-U' struct-invs-V - ‹xs ! i > 0›, of Laa]
```

```
    unfolding S T can-delete-def
    by (auto dest: no-dup-same-annotD)

  have star: ‹remove-one-annot-true-clause T (mark-garbage-l (xs ! i) T)›
    unfolding st T mark-garbage-l-def prod.simps
    apply (rule remove-one-annot-true-clause.delete)
    subgoal using in-dom i-le unfolding st prod.case T by auto
    subgoal using can-del unfolding T can-delete-def by auto
    subgoal using not-annot unfolding T by auto
    done
  moreover have ‹get-trail-l (mark-garbage-l (xs ! i) T) = get-trail-l T›
    by (cases T) (auto simp: mark-garbage-l-def)
  ultimately show ‹mark-to-delete-clauses-l-inv S xs0
      (i, mark-garbage-l (xs ! i) T, delete-index-and-swap xs i)›
    using inv apply −
    unfolding mark-to-delete-clauses-l-inv-def prod.simps st
    apply normalize-goal+
    by (intro conjI; (rule-tac x=x in exI)?)
     auto

  show ‹I xs0 (i, mark-garbage-l (xs ! i) T, delete-index-and-swap xs i)›
    using rem star unfolding st I-def by simp
qed
have
  mark-to-delete-clauses-l-inv-keep:
    ‹mark-to-delete-clauses-l-inv S xs0 (i + 1, T, xs)› and
  I-keep: ‹I xs0 (i + 1, T, xs)›
  if
    ‹mark-to-delete-clauses-l-pre S› and
    inv: ‹mark-to-delete-clauses-l-inv S xs0 s› and
    I: ‹I xs0 s› and
    cond: ‹case s of (i, S, xs) ⇒ i < length xs› and
    st: ‹sT = (T, xs)›
       ‹s = (i, sT)› and
    dom: ‹¬ xs ! i ∉# dom-m (get-clauses-l T)› and
    ‹0 < length (get-clauses-l T ∝ (xs ! i))› and
    ‹can-delete T (xs ! i) b› and
    ‹i < length xs› and
    ‹¬ b›
  for x s i T b xs0 sT xs
proof −
  show ‹mark-to-delete-clauses-l-inv S xs0 (i + 1, T, xs)›
    using inv
    unfolding mark-to-delete-clauses-l-inv-def prod.simps st
    by fast
  show  ‹I xs0 (i + 1, T, xs)›
    using I unfolding I-def st prod.simps .
qed
have remove-all-learned-subsumed-clauses: ‹remove-all-learned-subsumed-clauses aa
     ≤ SPEC
       (λT. remove-one-annot-true-clause^++ S T ∧
           get-trail-l S = get-trail-l T)›
  if
    ‹mark-to-delete-clauses-l-pre S› and
    ‹xs0 ∈ {N. True}› and
    ‹True› and
```

```
    ‹mark-to-delete-clauses-l-inv S xs0 s› and
    ‹I xs0 s› and
    ‹¬ (case s of (i, S, xs) ⇒ i < length xs)› and
    ‹s = (a, b)› and
    ‹b = (aa, ba)›
  for x s a b aa ba xs0
proof −
  have 1: ‹remove-all-learned-subsumed-clauses aa
     ≤ SPEC
       (λT. remove-one-annot-true-clause aa T ∧
           get-trail-l aa = get-trail-l T)›
    unfolding remove-all-learned-subsumed-clauses-def
    by refine-rcg
      (auto intro!: remove-one-annot-true-clause.delete-subsumed)
  show ?thesis
    by (rule 1[THEN order-trans])
      (use that in ‹auto simp: mark-to-delete-clauses-l-inv-def›)
qed
show ?thesis
  unfolding mark-to-delete-clauses-l-def collect-valid-indices-def
  apply (rule ASSERT-refine-left)
   apply (rule mark0)
  apply (subst intro-spec-iff)
  apply (intro ballI)
  subgoal for xs0
    apply (refine-vcg
      WHILEIT-rule-stronger-inv[where I'=‹I xs0› and
        R= ‹measure (λ(i :: nat, N, xs0). Suc (length xs0) − i)›])
    subgoal by auto
    subgoal using list-invs confl upd ST struct-invs unfolding mark-to-delete-clauses-l-inv-def
      by (cases S') force
    subgoal by (rule I0)
    subgoal
      by (rule mark-to-delete-clauses-l-inv-notin)
    subgoal
      by (rule I-notin)
    subgoal
      by auto
    subgoal
      by (rule length-ge0)
    subgoal
      apply (auto simp: mark-to-delete-clauses-l-inv-def )
      by (metis gr0I rtranclp-cdcl-twl-restart-l-list-invs
        rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2 twl-list-invs-def)
    subgoal
      by auto
    subgoal — delete clause
      by (rule mark-to-delete-clauses-l-inv-del)
    subgoal
      by (rule I-del)
    subgoal
      by auto
    subgoal — Keep clause
      by (rule mark-to-delete-clauses-l-inv-keep)
    subgoal
      by (rule I-keep)
```

```
      subgoal
        by auto
      subgoal for x s a b aa ba
        by (rule remove-all-learned-subsumed-clauses)
      done
    done
qed

definition GC-clauses :: ‹nat clauses-l ⇒ nat clauses-l ⇒ (nat clauses-l × (nat ⇒ nat option)) nres›
where
‹GC-clauses N N' = do {
  xs ← SPEC(λxs. set-mset (dom-m N) ⊆ set xs);
  (N, N', m) ← nfoldli
    xs
    (λ(N, N', m). True)
    (λC (N, N', m).
       if C ∈# dom-m N
       then do {
         C' ← SPEC(λi. i ∉# dom-m N' ∧ i ≠ 0);
  RETURN (fmdrop C N, fmupd C' (N ∝ C, irred N C) N', m(C ↦ C'))
       }
       else
         RETURN (N, N', m))
    (N, N', (λ-. None));
  RETURN (N', m)
}›


inductive GC-remap
  :: ‹('a, 'b) fmap × ('a ⇒ 'c option) × ('c, 'b) fmap ⇒  ('a, 'b) fmap × ('a ⇒ 'c option) × ('c, 'b)
fmap ⇒ bool›
where
remap-cons:
  ‹GC-remap (N, m, new) (fmdrop C N, m(C ↦ C'), fmupd C' (the (fmlookup N C)) new)›
   if ‹C' ∉# dom-m new› and
      ‹C ∈# dom-m N› and
      ‹C ∉ dom m› and
      ‹C' ∉ ran m›

lemma GC-remap-ran-m-old-new:
  ‹GC-remap (old, m, new) (old', m', new') ⟹ ran-m old + ran-m new = ran-m old' + ran-m new'›
  by (induction (old, m, new) (old', m', new') rule: GC-remap.induct)
    (auto simp: ran-m-lf-fmdrop ran-m-mapsto-upd-notin)

lemma GC-remap-init-clss-l-old-new:
  ‹GC-remap (old, m, new) (old', m', new') ⟹
    init-clss-l old + init-clss-l new = init-clss-l old' + init-clss-l new'›
  by (induction (old, m, new) (old', m', new') rule: GC-remap.induct)
    (auto simp: ran-m-lf-fmdrop ran-m-mapsto-upd-notin split: if-splits)

lemma GC-remap-learned-clss-l-old-new:
  ‹GC-remap (old, m, new) (old', m', new') ⟹
    learned-clss-l old + learned-clss-l new = learned-clss-l old' + learned-clss-l new'›
  by (induction (old, m, new) (old', m', new') rule: GC-remap.induct)
    (auto simp: ran-m-lf-fmdrop ran-m-mapsto-upd-notin split: if-splits)
```

lemma *GC-remap-ran-m-remap*:
‹*GC-remap* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) ⟹ *C* ∈# *dom-m old* ⟹ *C* ∉# *dom-m old'* ⟹
m' C ≠ *None* ∧
fmlookup new' (the (m' C)) = fmlookup old C›
by (*induction* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*)

lemma *GC-remap-ran-m-no-rewrite-map*:
‹*GC-remap* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) ⟹ *C* ∉# *dom-m old* ⟹ *m' C* = *m C*›
by (*induction* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*)

lemma *GC-remap-ran-m-no-rewrite-fmap*:
‹*GC-remap* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) ⟹ *C* ∈# *dom-m new* ⟹
C ∈# *dom-m new'* ∧ *fmlookup new C* = *fmlookup new' C*›
by (*induction* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*)

lemma *rtranclp-GC-remap-init-clss-l-old-new*:
‹*GC-remap*** *S S'* ⟹
init-clss-l (*fst S*) + *init-clss-l* (*snd* (*snd S*)) = *init-clss-l* (*fst S'*) + *init-clss-l* (*snd* (*snd S'*))›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*
dest: *GC-remap-init-clss-l-old-new*)

lemma *rtranclp-GC-remap-learned-clss-l-old-new*:
‹*GC-remap*** *S S'* ⟹
learned-clss-l (*fst S*) + *learned-clss-l* (*snd* (*snd S*)) =
learned-clss-l (*fst S'*) + *learned-clss-l* (*snd* (*snd S'*))›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*
dest: *GC-remap-learned-clss-l-old-new*)

lemma *rtranclp-GC-remap-ran-m-no-rewrite-fmap*:
‹*GC-remap*** *S S'* ⟹ *C* ∈# *dom-m* (*snd* (*snd S*)) ⟹
C ∈# *dom-m* (*snd* (*snd S'*)) ∧ *fmlookup* (*snd* (*snd S*)) *C* = *fmlookup* (*snd* (*snd S'*)) *C*›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin dest*: *GC-remap-ran-m-no-rewrite-fmap*)

lemma *GC-remap-ran-m-no-rewrite*:
‹*GC-remap S S'* ⟹ *C* ∈# *dom-m* (*fst S*) ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹
fmlookup (*fst S*) *C* = *fmlookup* (*fst S'*) *C*›
by (*induction rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin distinct-mset-dom*
distinct-mset-set-mset-remove1-mset
dest: *GC-remap-ran-m-remap*)

lemma *GC-remap-ran-m-lookup-kept*:
assumes
‹*GC-remap*** *S y*› **and**
‹*GC-remap y z*› **and**
‹*C* ∈# *dom-m* (*fst S*)› **and**
‹*C* ∈# *dom-m* (*fst z*)› **and**
‹*C* ∉# *dom-m* (*fst y*)›

shows ‹*fmlookup* (*fst S*) *C* = *fmlookup* (*fst z*) *C*›
using *assms* **by** (*smt GC-remap.cases fmlookup-drop fst-conv in-dom-m-lookup-iff*)

lemma *rtranclp-GC-remap-ran-m-no-rewrite*:
‹*GC-remap*** *S S'* ⟹ *C* ∈# *dom-m* (*fst S*) ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹
fmlookup (*fst S*) *C* = *fmlookup* (*fst S'*) *C*›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
by (*cases* ‹*C* ∈# *dom-m* (*fst y*)›)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin dest*: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept*)
done

lemma *GC-remap-ran-m-no-lost*:
‹*GC-remap S S'* ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹ *C* ∈# *dom-m* (*fst S*)›
by (*induction rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin distinct-mset-dom distinct-mset-set-mset-remove1-mset*
dest: *GC-remap-ran-m-remap*)

lemma *rtranclp-GC-remap-ran-m-no-lost*:
‹*GC-remap*** *S S'* ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹ *C* ∈# *dom-m* (*fst S*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
by (*cases* ‹*C* ∈# *dom-m* (*fst y*)›)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*
dest: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost*)
done

lemma *GC-remap-ran-m-no-new-lost*:
‹*GC-remap S S'* ⟹ *dom* (*fst* (*snd S*)) ⊆ *set-mset* (*dom-m* (*fst S*)) ⟹
dom (*fst* (*snd S'*)) ⊆ *set-mset* (*dom-m* (*fst S*))›
by (*induction rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin distinct-mset-dom*
distinct-mset-set-mset-remove1-mset
dest: *GC-remap-ran-m-remap*)

lemma *rtranclp-GC-remap-ran-m-no-new-lost*:
‹*GC-remap*** *S S'* ⟹ *dom* (*fst* (*snd S*)) ⊆ *set-mset* (*dom-m* (*fst S*)) ⟹
dom (*fst* (*snd S'*)) ⊆ *set-mset* (*dom-m* (*fst S*))›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
using *GC-remap-ran-m-no-lost*[*of y z*] *GC-remap-ran-m-remap*[*of* ‹*fst y*› ‹*fst* (*snd y*)› ‹*snd* (*snd y*)›
‹*fst z*› ‹*fst* (*snd z*)› ‹*snd* (*snd z*)›] *rtranclp-GC-remap-ran-m-no-lost*[*of x y*]
GC-remap-ran-m-no-rewrite-map[*of* ‹*fst y*› ‹*fst* (*snd y*)› ‹*snd* (*snd y*)›
‹*fst z*› ‹*fst* (*snd z*)› ‹*snd* (*snd z*)›]
apply (*auto simp flip*: *in-dom-m-lookup-iff simp del*: *lookup-None-notin-dom-m*)
apply (*smt* (*verit*, *best*) *domIff option.collapse option.discI rtranclp-GC-remap-ran-m-no-lost subset-eq*)
done
done

lemma *rtranclp-GC-remap-map-ran*:
assumes
‹*GC-remap*** *S S'*› **and**
‹(*the* ∘∘ *fst*) (*snd S*) '# *mset-set* (*dom* (*fst* (*snd S*))) = *dom-m* (*snd* (*snd S*))› **and**
‹*finite* (*dom* (*fst* (*snd S*)))›
shows ‹*finite* (*dom* (*fst* (*snd S'*))) ∧
(*the* ∘∘ *fst*) (*snd S'*) '# *mset-set* (*dom* (*fst* (*snd S'*))) = *dom-m* (*snd* (*snd S'*))›
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step y z*) **note** *star* = *this*(*1*) **and** *st* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *H* = *this*(*4*−)
from *st*
show *?case*
proof *cases*
case (*remap-cons C' new C N m*)
have ‹*C* ∉ *dom m*›
using *step remap-cons* **by** *auto*
then have [*simp*]: ‹{#*the* (*if x* = *C then Some C' else m x*). *x* ∈# *mset-set* (*dom m*)#} =
{#*the* (*m x*). *x* ∈# *mset-set* (*dom m*)#}›
apply (*auto intro*!: *image-mset-cong split*: *if-splits*)
by (*metis empty-iff finite-set-mset-mset-set local.remap-cons*(*5*) *mset-set.infinite set-mset-empty*)

show *?thesis*
using *step remap-cons*
by (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*
dest: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost dest*:)
qed
qed

lemma *rtranclp-GC-remap-ran-m-no-new-map*:
‹*GC-remap*** *S S'* ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹ *C* ∈# *dom-m* (*fst S*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
by (*cases* ‹*C* ∈# *dom-m* (*fst y*)›)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin dest*: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost*)
done

lemma *rtranclp-GC-remap-learned-clss-lD*:
‹*GC-remap*** (*N*, *x*, *m*) (*N'*, *x'*, *m'*) ⟹ *learned-clss-l N* + *learned-clss-l m* = *learned-clss-l N'* + *learned-clss-l m'*›
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*: *GC-remap-learned-clss-l-old-new*)

lemma *rtranclp-GC-remap-learned-clss-l*:
‹*GC-remap*** (*x1a*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *x1ad*) ⟹ *learned-clss-l x1ad* = *learned-clss-l x1a*›
by (*auto dest*!: *rtranclp-GC-remap-learned-clss-lD*[*of* - - - - - -])

lemma *remap-cons2*:

assumes
‹$C' \notin\# \ dom\text{-}m \ new$› **and**
‹$C \in\# \ dom\text{-}m \ N$› **and**
‹$(the \circ\circ fst) \ (snd \ (N, \ m, \ new)) \ `\# \ mset\text{-}set \ (dom \ (fst \ (snd \ (N, \ m, \ new)))) =$
$dom\text{-}m \ (snd \ (snd \ (N, \ m, \ new)))$› **and**
‹$\bigwedge x. \ x \in\# \ dom\text{-}m \ (fst \ (N, \ m, \ new)) \Longrightarrow x \notin dom \ (fst \ (snd \ (N, \ m, \ new)))$› **and**
‹$finite \ (dom \ m)$›
shows
‹$GC\text{-}remap \ (N, \ m, \ new) \ (fmdrop \ C \ N, \ m(C \mapsto C'), \ fmupd \ C' \ (the \ (fmlookup \ N \ C)) \ new)$›
proof −
have *3*: ‹$C \in dom \ m \Longrightarrow False$›
apply (*drule mk-disjoint-insert*)
using *assms*
apply (*auto 5 5 simp*: *ran-def*)
done

have *4*: ‹*False*› **if** *C'*: ‹$C' \in ran \ m$›
proof −
obtain *a* **where** *a*: ‹$a \in dom \ m$› **and** [*simp*]: ‹$m \ a = Some \ C'$›
using *that C'* **unfolding** *ran-def*
by *auto*
show *False*
using *mk-disjoint-insert*[*OF a*] *assms* **by** (*auto simp*: *union-single-eq-member*)
qed

show *?thesis*
apply (*rule remap-cons*)
apply (*rule assms*(*1*))
apply (*rule assms*(*2*))
apply (*use 3* **in** *fast*)
apply (*use 4* **in** *fast*)
done
qed

inductive-cases *GC-remapE*: ‹*GC-remap S T*›

lemma *rtranclp-GC-remap-finite-map*:
‹$GC\text{-}remap^{**} \ \ S \ S' \Longrightarrow finite \ (dom \ (fst \ (snd \ S))) \Longrightarrow finite \ (dom \ (fst \ (snd \ S')))$›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
by (*auto elim*: *GC-remapE*)
done

lemma *rtranclp-GC-remap-old-dom-map*:
‹$GC\text{-}remap^{**} \ \ R \ S \Longrightarrow (\bigwedge x. \ x \in\# \ dom\text{-}m \ (fst \ R) \Longrightarrow x \notin dom \ (fst \ (snd \ R))) \Longrightarrow$
$(\bigwedge x. \ x \in\# \ dom\text{-}m \ (fst \ S) \Longrightarrow x \notin dom \ (fst \ (snd \ S)))$›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z x*
by (*fastforce elim*!: *GC-remapE simp*: *distinct-mset-dom distinct-mset-set-mset-remove1-mset*)
done

lemma *remap-cons2-rtranclp*:

assumes
‹(*the* ∘∘ *fst*) (*snd R*) '# *mset-set* (*dom* (*fst* (*snd R*))) = *dom-m* (*snd* (*snd R*))› **and**
‹⋀*x*. *x* ∈# *dom-m* (*fst R*) ⟹ *x* ∉ *dom* (*fst* (*snd R*))› **and**
‹*finite* (*dom* (*fst* (*snd R*)))› **and**
st: ‹*GC-remap*** *R S› **and**
C': ‹*C'* ∉# *dom-m* (*snd* (*snd S*))› **and**
C: ‹*C* ∈# *dom-m* (*fst S*)›
shows
‹*GC-remap*** *R* (*fmdrop C* (*fst S*), (*fst* (*snd S*))(*C* ↦ *C'*), *fmupd C'* (*the* (*fmlookup* (*fst S*) *C*)) (*snd* (*snd S*)))›
proof −
have
1: ‹(*the* ∘∘ *fst*) (*snd S*) '# *mset-set* (*dom* (*fst* (*snd S*))) = *dom-m* (*snd* (*snd S*))› **and**
2: ‹⋀*x*. *x* ∈# *dom-m* (*fst S*) ⟹ *x* ∉ *dom* (*fst* (*snd S*))› **and**
3: ‹*finite* (*dom* (*fst* (*snd S*)))›
using *assms*(*1*) *assms*(*3*) *assms*(*4*) *rtranclp-GC-remap-map-ran* **apply** *blast*
apply (*meson assms*(*2*) *assms*(*4*) *rtranclp-GC-remap-old-dom-map*)
using *assms*(*3*) *assms*(*4*) *rtranclp-GC-remap-finite-map* **by** *blast*
have *5*: ‹*GC-remap S*
(*fmdrop C* (*fst S*), (*fst* (*snd S*))(*C* ↦ *C'*), *fmupd C'* (*the* (*fmlookup* (*fst S*) *C*)) (*snd* (*snd S*)))›
using *remap-cons2*[*OF C' C*, *of* ‹(*fst* (*snd S*))›] *1 2 3* **by** (*cases S*) *auto*
show *?thesis*
using *5 st* **by** *simp*
qed

lemma (**in** −) *fmdom-fmrestrict-set*: ‹*fmdrop xa* (*fmrestrict-set s N*) = *fmrestrict-set* (*s* − {*xa*}) *N*›
by (*rule fmap-ext-fmdom*)
(*auto simp*: *fset-fmdom-fmrestrict-set fmember.rep-eq notin-fset*)

lemma (**in** −) *GC-clauses-GC-remap*:
‹*GC-clauses N fmempty* ≤ *SPEC*(λ(*N''*, *m*). *GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N''*) ∧
0 ∉# *dom-m N''*)›
proof −
let *?remap* = ‹(*GC-remap*)** (*N*, λ-. *None*, *fmempty*)›
note *remap* = *remap-cons2-rtranclp*[*of* ‹(*N*, λ-. *None*, *fmempty*)›, *of* ‹(*a*, *b*, *c*)› **for** *a b c*, *simplified*]
define *I* **where**
‹*I a b* ≡ (λ(*old* :: *nat clauses-l*, *new* :: *nat clauses-l*, *m* :: *nat* ⇒ *nat option*).
?remap (*old*, *m*, *new*) ∧ *0* ∉# *dom-m new* ∧
set-mset (*dom-m old*) ⊆ *set b*)›
for *a b* :: ‹*nat list*›
have *I0*: ‹*set-mset* (*dom-m N*) ⊆ *set x* ⟹ *I* [] *x* (*N*, *fmempty*, λ-. *None*)› **for** *x*
unfolding *I-def*
by (*auto intro*!: *fmap-ext-fmdom simp*: *fset-fmdom-fmrestrict-set fmember.rep-eq*
notin-fset dom-m-def)

have *I-drop*: ‹*I* (*l1* @ [*xa*]) *l2*
(*fmdrop xa a*, *fmupd xb* (*a* ∝ *xa*, *irred a xa*) *aa*, *ba*(*xa* ↦ *xb*))›
if
‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
‹*x* = *l1* @ *xa* # *l2*› **and**
‹*I l1* (*xa* # *l2*) *σ*› **and**
‹*case σ of* (*N*, *N'*, *m*) ⇒ *True*› **and**
‹*σ* = (*a*, *b*)› **and**
‹*b* = (*aa*, *ba*)› **and**
‹*xa* ∈# *dom-m a*› **and**

```
  ‹xb ∉# dom-m aa ∧ xb ≠ 0›
  for x xa l1 l2 σ a b aa ba xb
proof −
  have ‹insert xa (set l2) − set l1 − {xa} = set l2 − insert xa (set l1)›
    by auto
  have ‹GC-remap** (N, Map.empty, fmempty)
      (fmdrop xa a, ba(xa ↦ xb), fmupd xb (the (fmlookup a xa)) aa)›
    by (rule remap)
      (use that in ‹auto simp: I-def›)
  then show ?thesis
    using that distinct-mset-dom[of a] distinct-mset-dom[of aa] unfolding I-def prod.simps
    apply (auto dest!: mset-le-subtract[of ‹dom-m -› - ‹{#xa#}›] simp: mset-set.insert-remove)
    by (metis Diff-empty Diff-insert0 add-mset-remove-trivial finite-set-mset
      finite-set-mset-mset-set insert-subset-eq-iff mset-set.remove set-mset-mset subseteq-remove1)
qed

have I-notin: ‹I (l1 @ [xa]) l2 (a, aa, ba)›
  if
    ‹set-mset (dom-m N) ⊆ set x› and
    ‹x = l1 @ xa # l2› and
    ‹I l1 (xa # l2) σ› and
    ‹case σ of (N, N', m) ⇒ True› and
    ‹σ = (a, b)› and
    ‹b = (aa, ba)› and
    ‹xa ∉# dom-m a›
    for x xa l1 l2 σ a b aa ba
proof −
  show ?thesis
    using that unfolding I-def
    by (auto dest!: multi-member-split)
qed
have early-break: ‹GC-remap** (N, Map.empty, fmempty) (fmempty, x2, x1)›
   if
     ‹set-mset (dom-m N) ⊆ set x› and
     ‹x = l1 @ l2› and
     ‹I l1 l2 σ› and
     ‹¬ (case σ of (N, N', m) ⇒ True)› and
     ‹σ = (a, b)› and
     ‹b = (aa, ba)› and
     ‹(aa, ba) = (x1, x2)›
    for x l1 l2 σ a b aa ba x1 x2
 proof −
   show ?thesis using that by auto
 qed

have final-rel: ‹GC-remap** (N, Map.empty, fmempty) (fmempty, x2, x1)›
if
  ‹set-mset (dom-m N) ⊆ set x› and
  ‹I x [] σ› and
  ‹case σ of (N, N', m) ⇒ True› and
  ‹σ = (a, b)› and
  ‹b = (aa, ba)› and
  ‹(aa, ba) = (x1, x2)›
proof −
  show ‹GC-remap** (N, Map.empty, fmempty) (fmempty, x2, x1)›
    using that
```

by (*auto simp*: *I-def*)
qed
have *final-rel*: ‹*GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *x2*, *x1*)› ‹*0* ∉# *dom-m x1*›
if
‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
‹*I x* [] *σ*› **and**
‹*case σ of* (*N*, *N'*, *m*) ⇒ *True*› **and**
‹*σ* = (*a*, *b*)› **and**
‹*b* = (*aa*, *ba*)› **and**
‹(*aa*, *ba*) = (*x1*, *x2*)›
for *x σ a b aa ba x1 x2*
using *that*
by (*auto simp*: *I-def*)
show *?thesis*
unfolding *GC-clauses-def*
apply (*refine-vcg nfoldli-rule*[**where** *I* = *I*])
subgoal by (*rule I0*)
subgoal by (*rule I-drop*)
subgoal by (*rule I-notin*)
— Final properties:
subgoal for *x l1 l2 σ a b aa ba x1 x2*
by (*rule early-break*)
subgoal
by (*auto simp*: *I-def*)
subgoal
by (*rule final-rel*) *assumption*+
subgoal
by (*rule final-rel*) *assumption*+
done
qed

definition *cdcl-GC-clauses-pre* :: ‹*'v twl-st-l* ⇒ *bool*› **where**
‹*cdcl-GC-clauses-pre S* ⟷ (
∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧
twl-list-invs S ∧ *twl-struct-invs T* ∧
get-conflict-l S = *None* ∧ *clauses-to-update-l S* = {#} ∧
count-decided (*get-trail-l S*) = *0* ∧ (∀ *L*∈*set* (*get-trail-l S*). *mark-of L* = *0*)
) ›

definition *cdcl-GC-clauses* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*cdcl-GC-clauses* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*cdcl-GC-clauses-pre* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
b ← *SPEC*(λ*b*. *True*);
if b then do {
(*N'*, -) ← *SPEC* (λ(*N''*, *m*). *GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N''*) ∧
0 ∉# *dom-m N''*);
RETURN (*M*, *N'*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *WS*, *Q*)
}
else RETURN (*M*, *N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *WS*, *Q*)})›

lemma *cdcl-GC-clauses-cdcl-twl-restart-l*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**

```
    upd: ‹clauses-to-update-l S = {#}› and
    count-dec: ‹count-decided (get-trail-l S) = 0› and
    mark: ‹∀ L∈set (get-trail-l S). mark-of L = 0›
  shows ‹cdcl-GC-clauses S ≤ SPEC (λT. cdcl-twl-restart-l S T ∧
      get-trail-l S = get-trail-l T)›
proof −
  show ?thesis
    unfolding cdcl-GC-clauses-def
    apply refine-vcg
    subgoal using assms unfolding cdcl-GC-clauses-pre-def by blast
    subgoal using confl upd count-dec mark by (auto simp: cdcl-twl-restart-l.simps
        valid-trail-reduction-refl
      dest: rtranclp-GC-remap-init-clss-l-old-new rtranclp-GC-remap-learned-clss-l-old-new
      intro!: exI[of - ‹{#}›])
    subgoal by simp
    subgoal using confl upd count-dec mark by (auto simp: cdcl-twl-restart-l.simps
        valid-trail-reduction-refl cdcl-GC-clauses-pre-def twl-list-invs-def)
    subgoal by simp
    done
qed

lemma remove-one-annot-true-clause-cdcl-twl-restart-l-spec:
  assumes
    ST: ‹(S, T) ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs S› and
    struct-invs: ‹twl-struct-invs T› and
    confl: ‹get-conflict-l S = None› and
    upd: ‹clauses-to-update-l S = {#}›
  shows ‹SPEC(remove-one-annot-true-clause^++ S) ≤ SPEC(λT. cdcl-twl-restart-l S T)›
proof −
  have ‹cdcl-twl-restart-l S U'›
    if rem: ‹remove-one-annot-true-clause^++ S U'› for U'
  proof −
    have n-d: ‹no-dup (get-trail-l S)›
      using ST struct-invs unfolding twl-struct-invs-def
          cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
          cdcl_W-restart-mset.cdcl_W-M-level-inv-def
          pcdcl-all-struct-invs-def state_W-of-def
      by (simp add: twl-st twl-st-l)
    have subs-U': ‹get-subsumed-learned-clauses-l U' = {#} ∧ get-learned-clauses0-l U' = {#}›
      using rem unfolding tranclp-unfold-end
      by (cases U'; auto simp: remove-one-annot-true-clause.simps)
    have SU': ‹cdcl-twl-restart-l** S U'›
      using rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2[of S U' T, OF
        tranclp-into-rtranclp[OF rem] list-invs
        confl upd ST struct-invs]
      apply −
      apply normalize-goal+
      by auto
    moreover have ‹cdcl-twl-restart-l S U'› if ‹S = U'›
      using confl upd rem rtranclp-cdcl-twl-restart-l-no-dup[OF SU'] list-invs
       n-d subs-U'
      unfolding that[symmetric]
      by (cases S)
        (auto simp: cdcl-twl-restart-l.simps twl-list-invs-def
         dest: no-dup-same-annotD)
```

ultimately show ‹*cdcl-twl-restart-l S U'*›
using *tranclp-cdcl-twl-restart-l-cdcl-is-cdcl-twl-restart-l*[*of S U'*, *OF - n-d*]
by (*meson rtranclpD*)
qed
then show *?thesis*
by *force*
qed

definition (**in** −) *restart-abs-l-pre* :: ‹*'v twl-st-l* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-abs-l-pre S last-GC last-Restart brk* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *restart-prog-pre S' last-GC last-Restart brk*
∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#})›

definition (**in** −) *cdcl-twl-local-restart-l-spec* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*cdcl-twl-local-restart-l-spec* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*). *do* {
ASSERT(∃ *last-GC last-Restart*. *restart-abs-l-pre* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)
last-GC last-Restart False);
(*M*, *Q*) ← *SPEC*(λ(*M'*, *Q'*). (∃ *K M2*. (*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q' = {#}) ∨ (*M'* = *M* ∧ *Q'* = *Q*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)
})›

definition *cdcl-twl-restart-l-prog* **where**
‹*cdcl-twl-restart-l-prog S* = *do* {
cdcl-twl-local-restart-l-spec S
}›

lemma *cdcl-twl-local-restart-l-spec-cdcl-twl-restart-l*:
assumes *inv*: ‹*restart-abs-l-pre S last-GC last-Restart False*›
shows ‹*cdcl-twl-local-restart-l-spec S* ≤
⇓ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *twl-list-invs S*}(*SPEC* (*cdcl-twl-restart-only-l S*))›
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*›
using *inv* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st-l twl-st*)
have *S*: ‹*S* = (*get-trail-l S*, *snd S*)›
by (*cases S*) *auto*

obtain *M N D NE UE NEk UEk NS US N0 U0 W Q* **where**
S: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)›
by (*cases S*)
have *restart*: ‹*cdcl-twl-restart-only-l S* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q'*) ∧
twl-list-invs (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q'*)› (**is** ‹*?A* ∧ *?B*›)
if *decomp'*: ‹(∃ *K M2*. (*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q' = {#}) ∨ (*M'* = *M* ∧ *Q'* = *Q*)›
for *M' K M2 Q'*

```
  proof −
    consider
      (nope) ‹M = M'› and ‹Q' = Q› |
      (decomp) K M2 where ‹(Decided K # M', M2) ∈ set (get-all-ann-decomposition M)› and
        ‹Q' = {#}›
      using decomp' by auto
    then have ?A
    proof cases
      case [simp]: nope
      have valid: ‹valid-trail-reduction M M'›
        by (use valid-trail-reduction.keep-red[of M'] in ‹auto simp: S›)
      have
        S1: ‹S = (M', N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q)› and
        S2 : ‹(M', N, D, NE, UE, NEk, UEk, NS, US, N0, U0, W, Q') =
          (M', N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q)›
        using confl upd unfolding S
        by auto
      show ?thesis
        unfolding S[symmetric] S1 S2
        by (rule cdcl-twl-restart-only-l.intros(2))
    next
      case decomp note decomp = this(1) and Q = this(2)
      have valid: ‹valid-trail-reduction M M'›
        by (use valid-trail-reduction.backtrack-red[OF decomp, of M'] in ‹auto simp: S›)
      have
        S1: ‹S = (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q)› and
        S2 : ‹(M', N, D, NE, UE, NEk, UEk, NS, US, N0, U0, W, Q') = (M', N, None, NE, UE,
          NEk, UEk, NS, US, N0, U0, {#}, {#})›
        using confl upd unfolding S Q
        by auto

      show ?thesis
        unfolding S[symmetric] S1 S2
        by (rule cdcl-twl-restart-only-l.intros)
         (rule decomp)
    qed
    moreover have ?B
      using decomp' list-invs by (auto simp: twl-list-invs-def S
        dest!: get-all-ann-decomposition-exists-prepend)
    ultimately show ?thesis by fast
  qed
  show ?thesis
    apply (subst S)
    unfolding cdcl-twl-local-restart-l-spec-def prod.case RES-RETURN-RES2 less-eq-nres.simps
      uncurry-def
    apply (rule ASSERT-leI)
    subgoal using assms unfolding S by fast
    apply (rule RES-refine)
    apply clarsimp
    apply (rule restart)
    apply simp
    done
qed

definition (in −) cdcl-twl-local-restart-l-spec0 :: ‹'v twl-st-l ⇒ 'v twl-st-l nres› where
  ‹cdcl-twl-local-restart-l-spec0 = (λ(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, W, Q). do {
```

ASSERT(∃ *last-GC last-Restart*. *restart-abs-l-pre* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)
last-GC last-Restart False);
(*M*, *Q*) ← *SPEC*(λ(*M′*, *Q′*). (∃ *K M2*. (*Decided K* # *M′*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q′ = {#} ∧ *count-decided M′* = *0*) ∨ (*M′* = *M* ∧ *Q′* = *Q* ∧ *count-decided M′* = *0*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)
})›

lemma *cdcl-twl-local-restart-l-spec0-cdcl-twl-restart-l*:
assumes *inv*: ‹*restart-abs-l-pre S last-GC last-Restart False*›
shows ‹*cdcl-twl-local-restart-l-spec0 S* ≤ *SPEC* (λ*T*. *cdcl-twl-restart-l S T* ∧ *count-decided* (*get-trail-l T*) = *0*)›
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*›
using *inv* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st-l twl-st*)
have *S*: ‹*S* = (*get-trail-l S*, *snd S*)›
by (*cases S*) *auto*

obtain *M N D NE UE NEk UEk NS US N0 U0 W Q* **where**
S: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *W*, *Q*)›
by (*cases S*)
have *restart*: ‹*cdcl-twl-restart-l S* (*M′*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q′*)› (**is** *?A*)
if *decomp′*: ‹(∃ *K M2*. (*Decided K* # *M′*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q′ = {#}) ∨ (*M′* = *M* ∧ *Q′* = *Q*)›
for *M′ K M2 Q′*
proof −
consider
(*nope*) ‹*M* = *M′*› **and** ‹*Q′* = *Q*› |
(*decomp*) *K M2* **where** ‹(*Decided K* # *M′*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
‹*Q′* = {#}›
using *decomp′* **by** *auto*
then show *?A*
proof *cases*
case [*simp*]: *nope*
have *valid*: ‹*valid-trail-reduction M M′*›
by (*use valid-trail-reduction.keep-red*[*of M′*] **in** ‹*auto simp*: *S*›)
have
S1: ‹*S* = (*M′*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)› **and**
S2 : ‹(*M′*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q′*) =
(*M′*, *N*, *None*, *NE* + *mset* ‘# {#}, *UE* + *mset* ‘# {#}, *NEk* + *mset* ‘# {#}, *UEk* + *mset* ‘# {#},
NS, {#}, *N0*, {#}, {#}, *Q*)›
using *confl upd* **unfolding** *S*
by *auto*
have

```
    ‹∀ C∈#clauses-to-update-l S. C ∈# dom-m (get-clauses-l S)› and
    dom0: ‹0 ∉# dom-m (get-clauses-l S)› and
    annot: ‹⋀L C. Propagated L C ∈ set (get-trail-l S) ⟹
      0 < C ⟹
       (C ∈# dom-m (get-clauses-l S) ∧
       L ∈ set (watched-l (get-clauses-l S ∝ C)) ∧
       (length (get-clauses-l S ∝ C) > 2 ⟶ L = get-clauses-l S ∝ C ! 0))› and
    ‹distinct-mset (clauses-to-update-l S)›
    using list-invs unfolding twl-list-invs-def S[symmetric] by auto
  have n-d: ‹no-dup M›
    using struct-invs ST unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def pcdcl-all-struct-invs-def
    by (auto simp: twl-st-l twl-st S)
  have propa-MM: ‹Propagated L E ∈ set M ⟹ Propagated L E' ∈ set M' ⟹ E=E'› for L E E'
    using n-d
    by (auto simp: S twl-list-invs-def
        dest!: split-list[of ‹Propagated L E› M]
        split-list[of ‹Propagated L E'› M]
        dest: no-dup-same-annotD
        elim!: list-match-lel-lel)

  show ?thesis
    unfolding S[symmetric] S1 S2
    apply (rule cdcl-twl-restart-l.intros[where UE' = ‹{#}›])
    subgoal by (auto)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal using propa-MM annot unfolding S by fastforce
    subgoal using propa-MM annot unfolding S by fastforce
    subgoal using propa-MM annot unfolding S by fastforce
    subgoal using dom0 unfolding S by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
next
  case decomp note decomp = this(1) and Q = this(2)
  have valid: ‹valid-trail-reduction M M'›
    by (use valid-trail-reduction.backtrack-red[OF decomp, of M'] in ‹auto simp: S›)
  have
    S1: ‹S = (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q)› and
    S2 : ‹(M', N, D, NE, UE, NEk, UEk, NS, {#}, N0, {#}, W, Q') =
      (M', N, None, NE + mset `# {#}, UE + mset `# {#},
      NEk + mset `# {#}, UEk + mset `# {#}, NS, {#}, N0, {#}, {#}, {#})›
    using confl upd unfolding S Q
    by auto

 have
    ‹∀ C∈#clauses-to-update-l S. C ∈# dom-m (get-clauses-l S)› and
    dom0: ‹0 ∉# dom-m (get-clauses-l S)› and
    annot: ‹⋀L C. Propagated L C ∈ set (get-trail-l S) ⟹
      0 < C ⟹
       (C ∈# dom-m (get-clauses-l S) ∧
       L ∈ set (watched-l (get-clauses-l S ∝ C)) ∧
```

```
      (length (get-clauses-l S ∝ C) > 2 ⟶ L = get-clauses-l S ∝ C ! 0))› and
    ‹distinct-mset (clauses-to-update-l S)›
    using list-invs unfolding twl-list-invs-def S[symmetric] by auto
  have n-d: ‹no-dup M›
    using struct-invs ST unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def pcdcl-all-struct-invs-def
    by (auto simp: twl-st-l twl-st S)
  obtain M3 where M: ‹M = M3 @ Decided K # M'›
    using decomp by auto
  have propa-MM: ‹Propagated L E ∈ set M' ⟹ Propagated L E' ∈ set M ⟹ E=E'› for L E E'
    using n-d
    by (auto simp: S twl-list-invs-def M
        dest!: split-list[of ‹Propagated L E› M]
        split-list[of ‹Propagated L E'› M]
        dest: no-dup-same-annotD
        elim!: list-match-lel-lel)

  show ?thesis
    unfolding S[symmetric] S1 S2
    apply (rule cdcl-twl-restart-l.intros[where UE' = ‹{#}›])
    subgoal by (rule valid)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal using propa-MM annot unfolding S by fastforce
    subgoal using propa-MM annot unfolding S by fastforce
    subgoal using propa-MM annot unfolding S by fastforce
    subgoal using dom0 unfolding S by auto
    subgoal using decomp unfolding S by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    done
 qed
qed
show ?thesis
  apply (subst S)
  unfolding cdcl-twl-local-restart-l-spec0-def prod.case RES-RETURN-RES2 less-eq-nres.simps
    uncurry-def
  apply (rule ASSERT-leI)
  subgoal using assms[unfolded S] by fast
  apply clarsimp
  apply (rule conjI)
  apply (rule restart)
  apply auto
  done
qed

definition cdcl-twl-full-restart-l-GC-prog-pre
  :: ‹'v twl-st-l ⇒ bool›
where
  ‹cdcl-twl-full-restart-l-GC-prog-pre S ⟷
   (∃ T. (S, T) ∈ twl-st-l None ∧ twl-struct-invs T ∧ twl-list-invs S ∧
    get-conflict T = None ∧ clauses-to-update T = {#} ∧
     cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init ((state_W-of T)))›
```

lemma *valid-trail-reduction-lit-of-nth*:
‹*valid-trail-reduction M M'* ⟹ *length M* = *length M'* ⟹ *i* < *length M* ⟹
lit-of (*M* ! *i*) = *lit-of* (*M'* ! *i*)›
apply (*induction rule*: *valid-trail-reduction.induct*)
subgoal premises *p* **for** *K M'' M2*
using *arg-cong*[*OF p(2)*, *of length*] *p(1) arg-cong*[*OF p(2)*, *of* ‹λ*xs*. *xs* ! *i*›] *p(4)*
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
subgoal premises *p*
using *arg-cong*[*OF p(1)*, *of length*] *p(3) arg-cong*[*OF p(1)*, *of* ‹λ*xs*. *xs* ! *i*›] *p(4)*
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
done

lemma *cdcl-twl-restart-l-lit-of-nth*:
‹*cdcl-twl-restart-l S U* ⟹ *i* < *length* (*get-trail-l U*) ⟹ *is-proped* (*get-trail-l U* ! *i*) ⟹
length (*get-trail-l S*) = *length* (*get-trail-l U*) ⟹
lit-of (*get-trail-l S* ! *i*) = *lit-of* (*get-trail-l U* ! *i*)›
apply (*induction rule*: *cdcl-twl-restart-l.induct*)
subgoal for *M M' N N' NE' UE' NE UE Q Q'*
using *valid-trail-reduction-length-leD*[*of M M'*]
valid-trail-reduction-lit-of-nth[*of M M' i*]
by *auto*
done

lemma *valid-trail-reduction-is-decided-nth*:
‹*valid-trail-reduction M M'* ⟹ *length M* = *length M'* ⟹ *i* < *length M* ⟹
is-decided (*M* ! *i*) = *is-decided* (*M'* ! *i*)›
apply (*induction rule*: *valid-trail-reduction.induct*)
subgoal premises *p* **for** *K M'' M2*
using *arg-cong*[*OF p(2)*, *of length*] *p(1) arg-cong*[*OF p(3)*, *of* ‹λ*xs*. *xs* ! *i*›] *p(4)*
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
subgoal premises *p*
using *arg-cong*[*OF p(1)*, *of length*] *p(3) arg-cong*[*OF p(2)*, *of* ‹λ*xs*. *xs* ! *i*›] *p(4)*
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
done

lemma *cdcl-twl-restart-l-mark-of-same-or-0*:
‹*cdcl-twl-restart-l S U* ⟹ *i* < *length* (*get-trail-l U*) ⟹ *is-proped* (*get-trail-l U* ! *i*) ⟹
length (*get-trail-l S*) = *length* (*get-trail-l U*) ⟹
(*mark-of* (*get-trail-l U* ! *i*) > *0* ⟹ *mark-of* (*get-trail-l S* ! *i*) > *0* ⟹
mset (*get-clauses-l S* ∝ *mark-of* (*get-trail-l S* ! *i*))
= *mset* (*get-clauses-l U* ∝ *mark-of* (*get-trail-l U* ! *i*)) ⟹ *P*) ⟹
(*mark-of* (*get-trail-l U* ! *i*) = *0* ⟹ *P*) ⟹ *P*›
apply (*induction rule*: *cdcl-twl-restart-l.induct*)
subgoal for *M M' N N' NE' UE' NE UE Q Q'*
using *valid-trail-reduction-length-leD*[*of M M'*]
valid-trail-reduction-lit-of-nth[*of M M' i*]
valid-trail-reduction-is-decided-nth[*of M M' i*]
split-list[*of* ‹*M* ! *i*› *M*, *OF nth-mem*] *split-list*[*of* ‹*M'* ! *i*› *M'*, *OF nth-mem*]
by (*cases* ‹*M* ! *i*›; *cases* ‹*M'* ! *i*›)
(*force simp*: *all-conj-distrib*)+
done

end
theory *Watched-Literals-Watch-List*
imports *Watched-Literals-List Watched-Literals-All-Literals*
begin

no-notation *funcset* (**infixr** $\rightarrow$ *60*)

Chapter 6

Third Refinement: Remembering watched

6.1 Types

type-synonym *clauses-to-update-wl* = ‹*nat multiset*›
type-synonym *'v watcher* = ‹(*nat* × *'v literal* × *bool*)›
type-synonym *'v watched* = ‹*'v watcher list*›
type-synonym *'v lit-queue-wl* = ‹*'v literal multiset*›

type-synonym *'v twl-st-wl* =
‹(*'v*, *nat*) *ann-lits* × *'v clauses-l* ×
'v cconflict × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* ×
'v clauses × *'v clauses* × *'v lit-queue-wl* × (*'v literal* ⇒ *'v watched*)›

6.2 Access Functions

fun *clauses-to-update-wl* :: ‹*'v twl-st-wl* ⇒ *'v literal* ⇒ *nat* ⇒ *clauses-to-update-wl*› **where**
‹*clauses-to-update-wl* (-, *N*, -, -, -, -, - ,- , -, -, -, -, *W*) *L i* =
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*mset* (*drop i* (*map fst* (*W L*))))›

fun *get-trail-wl* :: ‹*'v twl-st-wl* ⇒ (*'v*, *nat*) *ann-lit list*› **where**
‹*get-trail-wl* (*M*, -, -, -, -, -, - ,- , -) = *M*›

fun *literals-to-update-wl* :: ‹*'v twl-st-wl* ⇒ *'v lit-queue-wl*› **where**
‹*literals-to-update-wl* (-, -, -, -, -, - ,- , -, -, -, -, *Q*, -) = *Q*›

fun *set-literals-to-update-wl* :: ‹*'v lit-queue-wl* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl*› **where**
‹*set-literals-to-update-wl Q* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, -, *W*) =
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›

fun *get-conflict-wl* :: ‹*'v twl-st-wl* ⇒ *'v cconflict*› **where**
‹*get-conflict-wl* (-, -, *D*, -, -, -, -) = *D*›

fun *get-clauses-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses-l*› **where**
‹*get-clauses-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *WS*, *Q*) = *N*›

fun *get-unit-learned-clss-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses*› **where**
‹*get-unit-learned-clss-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *Q*, *W*) = *UE* + *UEk*›

fun *get-unit-init-clss-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses*› **where**

‹get-unit-init-clss-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = NE + NEk›

fun *get-unit-clauses-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-unit-clauses-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = NE + NEk + UE + UEk›

fun *get-kept-unit-clauses-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-kept-unit-clauses-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = NEk + UEk›

fun *get-unkept-learned-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-unkept-learned-clss-wl (M, N, D, NE, UE, NEk, UEk, NS, US, WS, Q) = UE›

fun *get-subsumed-init-clauses-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-subsumed-init-clauses-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = NS›

fun *get-subsumed-learned-clauses-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-subsumed-learned-clauses-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = US›

abbreviation *get-subsumed-clauses-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-subsumed-clauses-wl S ≡ get-subsumed-init-clauses-wl S + get-subsumed-learned-clauses-wl S›

fun *get-init-clauses0-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-init-clauses0-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) = N0›

fun *get-learned-clauses0-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-learned-clauses0-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) = U0›

abbreviation *get-clauses0-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-clauses0-wl S ≡ get-init-clauses0-wl S + get-learned-clauses0-wl S›

definition *get-learned-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clause-l multiset› **where**
‹get-learned-clss-wl S = learned-clss-lf (get-clauses-wl S)›

abbreviation *get-all-learned-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clause multiset› **where**
‹get-all-learned-clss-wl S ≡ mset `# get-learned-clss-wl S + get-unit-learned-clss-wl S +
get-subsumed-learned-clauses-wl S + get-learned-clauses0-wl S›

lemma *get-unit-clauses-wl-alt-def*:
‹get-unit-clauses-wl S = get-unit-init-clss-wl S + get-unit-learned-clss-wl S›
by (*cases S*) *auto*

fun *get-watched-wl* :: ‹'v twl-st-wl ⇒ ('v literal ⇒ 'v watched)› **where**
‹get-watched-wl (-, -, -, -, -, -, -, -, -, -, -, -, W) = W›

fun *get-unkept-unit-init-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-unkept-unit-init-clss-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = NE›

fun *get-unkept-unit-learned-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-unkept-unit-learned-clss-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = UE›

fun *get-kept-unit-init-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-kept-unit-init-clss-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = NEk›

fun *get-kept-unit-learned-clss-wl* :: ‹'v twl-st-wl ⇒ 'v clauses› **where**
‹get-kept-unit-learned-clss-wl (M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) = UEk›

lemma *get-unit-init-clss-wl-alt-def*:

‹*get-unit-init-clss-wl T = get-unkept-unit-init-clss-wl T + get-kept-unit-init-clss-wl T*›
by (*cases T*) *auto*

lemma *get-unit-learned-clss-wl-alt-def*:
‹*get-unit-learned-clss-wl T = get-unkept-unit-learned-clss-wl T + get-kept-unit-learned-clss-wl T*›
by (*cases T*) *auto*

abbreviation *get-init-clss-wl* :: ‹*'v twl-st-wl ⇒ 'v clause-l multiset*› **where**
‹*get-init-clss-wl S ≡ init-clss-lf (get-clauses-wl S)*›

definition *all-lits-st* :: ‹*'v twl-st-wl ⇒ 'v literal multiset*› **where**
‹*all-lits-st S ≡ all-lits (get-clauses-wl S) (get-unit-clauses-wl S + get-subsumed-clauses-wl S + get-clauses0-wl S)*›

definition *all-init-lits-of-wl* :: ‹*'v twl-st-wl ⇒ 'v clause*› **where**
‹*all-init-lits-of-wl S' ≡ all-lits-of-mm (mset `# get-init-clss-wl S' + get-unit-init-clss-wl S' +*
get-subsumed-init-clauses-wl S' + get-init-clauses0-wl S')›

definition *all-learned-lits-of-wl* :: ‹*'v twl-st-wl ⇒ 'v clause*› **where**
‹*all-learned-lits-of-wl S' ≡ all-lits-of-mm (mset `# learned-clss-lf (get-clauses-wl S') + get-unit-learned-clss-wl S' +*
get-subsumed-learned-clauses-wl S' + get-learned-clauses0-wl S')›

lemma *all-lits-st-alt-def*:
‹*Watched-Literals-Watch-List.all-lits-st S = all-init-lits-of-wl S + all-learned-lits-of-wl S*›
apply (*auto simp*: *all-lits-st-def all-init-lits-of-wl-def all-learned-lits-of-wl-def*
ac-simps all-lits-def all-lits-of-mm-union)
by (*metis all-clss-l-ran-m all-lits-of-mm-union get-unit-clauses-wl-alt-def image-mset-union union-assoc*)

lemma *all-init-lits-of-wl-all-lits-st*:
‹*set-mset (all-init-lits-of-wl S) ⊆ set-mset (all-lits-st S)*›
unfolding *all-lits-st-def all-init-lits-of-wl-def all-lits-def*
apply (*subst* (*2*) *all-clss-l-ran-m*[*symmetric*])
unfolding *image-mset-union*
apply (*cases S*)
apply (*auto simp*: *all-lits-of-mm-union*)
done

lemma *in-all-lits-uminus-iff*[*simp*]: ‹*- L ∈# all-lits-st S ⟷ L ∈# all-lits-st S*›
by (*auto simp*: *all-lits-st-def in-all-lits-of-mm-uminus-iff all-lits-def*)

lemma *all-lits-ofI*[*intro*]:
‹*x = get-clauses-wl S ⟹ C ∈# dom-m x ⟹ n < length (x ∝ C) ⟹ x ∝ C ! n ∈# all-lits-st S*›
using *in-clause-in-all-lits-of-m*[*of ‹x ∝ C ! n›*] *nth-mem-mset*[*of n ‹x ∝ C›*]
by (*auto simp*: *all-lits-st-def all-lits-def all-lits-of-mm-union ran-m-def all-lits-of-mm-add-mset*
dest!: *multi-member-split*)

6.3 Watch List Function

definition *op-watch-list* :: ‹(*'v literal ⇒ 'v watched*) ⇒ *'v literal ⇒ nat ⇒ 'v watcher*› **where**
[*simp*]: ‹*op-watch-list W K i = W K ! i*›

definition *mop-watch-list* :: ‹(*'v literal ⇒ 'v watched*) ⇒ *'v literal ⇒ nat ⇒ 'v watcher nres*› **where**
‹*mop-watch-list W K i = do {*

ASSERT($i < \mathit{length}\ (W\ K)$);
RETURN $(W\ K\ !\ i)$
}›

definition *op-watch-list-append* :: ‹(*'v literal* ⇒ *'v watched*) ⇒ *'v literal* ⇒ *'v watcher* ⇒ (*'v literal* ⇒ *'v watched*)› **where**
[*simp*]: ‹*op-watch-list-append W K x* = *W* (*K* := *W K* @ [*x*])›

definition *mop-watch-list-append* :: ‹(*'v literal* ⇒ *'v watched*) ⇒ *'v literal* ⇒ *'v watcher* ⇒ (*'v literal* ⇒ *'v watched*) *nres*› **where**
‹*mop-watch-list-append W K x = do* {
RETURN (*W* (*K* := *W K* @ [*x*]))
}›

definition *op-watch-list-take* :: ‹(*'v literal* ⇒ *'v watched*) ⇒ *'v literal* ⇒ *nat* ⇒ (*'v literal* ⇒ *'v watched*)› **where**
[*simp*]: ‹*op-watch-list-take W K i* = *W* (*K* := *take i* (*W K*))›

definition *mop-watch-list-take* :: ‹(*'v literal* ⇒ *'v watched*) ⇒ *'v literal* ⇒ *nat* ⇒ (*'v literal* ⇒ *'v watched*) *nres*› **where**
‹*mop-watch-list-take W K i = do* {
ASSERT($i \leq \mathit{length}\ (W\ K)$);
RETURN (*W* (*K* := *take i* (*W K*)))
}›

lemma shows
op-watch-list:
‹$i < \mathit{length}\ (W\ K) \Longrightarrow$ *mop-watch-list W K i* ≤ *SPEC*(λ*c*. (*c*, *op-watch-list W K i*) ∈ *Id*)› **and**
op-watch-list-append:
‹*mop-watch-list-append W K x* ≤ *SPEC*(λ*c*. (*c*, *op-watch-list-append W K x*) ∈ *Id*)› **and**
op-watch-list-take:
‹$i \leq \mathit{length}\ (W\ K) \Longrightarrow$ *mop-watch-list-take W K i* ≤ *SPEC*(λ*c*. (*c*, *op-watch-list-take W K i*) ∈ *Id*)›
by (*auto simp*: *mop-watch-list-def mop-watch-list-append-def mop-watch-list-take-def*
intro!: *ASSERT-leI*)

6.4 Watch List Invariants

We cannot just extract the literals of the clauses: we cannot be sure that atoms appear *both* positively and negatively in the clauses. If we could ensure that there are no pure literals, the definition of *all-lits-of-mm* can be changed to *all-lits-of-mm Ls* = $\sum$ <^*sub*># *Ls*.

In this definition K is the blocking literal.

We have several different version of the watch-list invariants, either because we need a different version or to simplify proofs.

1. CDCL: This is the invariant that is the most important.
   - binary clauses cannot be deleted
   - blocking literals are in the clause
   - the watched literals belong to the clause and are at the beginning.
   - the set of all literals is the set of all literals (irred+red)
2. Inprocessing, deduplicating binary clauses

- binary clauses can be deleted
- blocking literals still are in the clause
- the watched literals belong to the clause and are at the beginning.
- the set of all literals is the set of all irredundant literals (irred)

3. Inprocessing, removing true/false literals

   - all clauses appear in the watch list
   - the set of all literals is the set of all irredundant literals (irred)

(We also have the version for all literals)

1. Reduction

   - all clauses appear in the watch list
   - the set of all literals is the set of all irredundant literals (irred)

   We use the set of irredundant literals because it is easier to handle removing literals – deleting a clause does not change the set of all irredundant literals. We then rely on the invariants to go back to the set of all literals.

One additional constraint is that the watch lists do not contain duplicates. This might seem like a consequence from the fact that we are correctly watching. However, the invariant talks only about non-deleted clauses. Technically we would not need the distinctiveness at this level, but during refinement we need it in order to bound the length of the watch lists.

fun *correctly-marked-as-binary* **where**
‹*correctly-marked-as-binary N* (*i*, *K*, *b*) ⟷ (*b* ⟷ (*length* (*N* ∝ *i*) = *2*))›

declare *correctly-marked-as-binary.simps*[*simp del*]

abbreviation *distinct-watched* :: ‹'*v watched* ⇒ *bool*› **where**
‹*distinct-watched xs* ≡ *distinct* (*map* (λ(*i*, *j*, *k*). *i*) *xs*)›

lemma *distinct-watched-alt-def*: ‹*distinct-watched xs* = *distinct* (*map fst xs*)›
by (*induction xs*; *auto*)

fun *correct-watching-except* :: ‹*nat* ⇒ *nat* ⇒ '*v literal* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*correct-watching-except i j K* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
(*L* = *K* ⟶
distinct-watched (*take i* (*W L*) @ *drop j* (*W L*)) ∧
((∀ (*i*, *K*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧
K ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)). *b* ⟶ *i* ∈# *dom-m N*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* '# *mset* (*take i* (*W L*) @ *drop j* (*W L*))) = *clause-to-update* *L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))) ∧
(*L* ≠ *K* ⟶
distinct-watched (*W L*) ∧
((∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧

$(\forall (i, K, b) \in \#\, mset\ (W\ L).\ b \longrightarrow i \in\#\ dom\text{-}m\ N) \wedge$
filter-mset $(\lambda i.\ i \in\#\ dom\text{-}m\ N)$ $(fst\ `\#\ mset\ (W\ L)) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))))\rangle$

fun *correct-watching* :: $\langle 'v\ twl\text{-}st\text{-}wl \Rightarrow bool\rangle$ **where**
$\langle correct\text{-}watching\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) \longleftrightarrow$
$(\forall L \in\#\ all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W).$
distinct-watched $(W\ L) \wedge$
$(\forall (i, K, b) \in \#\, mset\ (W\ L).\ i \in\#\ dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge K \neq L \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i, K, b)) \wedge$
$(\forall (i, K, b) \in \#\, mset\ (W\ L).\ b \longrightarrow i \in\#\ dom\text{-}m\ N) \wedge$
filter-mset $(\lambda i.\ i \in\#\ dom\text{-}m\ N)$ $(fst\ `\#\ mset\ (W\ L)) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))\rangle$

declare *correct-watching.simps*[*simp del*]

lemma *all-lits-st-simps*[*simp*]:
$\langle all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(K := WK)) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
$\langle all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, add\text{-}mset\ K\ Q, W) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$ — actually covered below, but still useful for 'unfolding' by hand
$\langle x1 \in\#\ dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow n' < length\ (x1aa \propto x1) \Longrightarrow$
all-lits-st $(x1b, x1aa(x1 \hookrightarrow WB\text{-}More\text{-}Refinement\text{-}List.swap\ (x1aa \propto x1)\ n\ n'), D, x1c, x1d, NEk, UEk, NS, US, N0, U0, x1e, x2e) =$
all-lits-st $(x1b, x1aa, D, x1c, x1d, NEk, UEk, NS, US, N0, U0, x1e, x2e)\rangle$
$\langle all\text{-}lits\text{-}st\ (L\ \#\ M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
$\langle NO\text{-}MATCH\ \{\#\}\ Q \Longrightarrow all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, W)\rangle$
$\langle NO\text{-}MATCH\ []\ M \Longrightarrow all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $([], N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
$\langle NO\text{-}MATCH\ None\ D \Longrightarrow all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
$all\text{-}lits\text{-}st\ (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
$\langle all\text{-}lits\text{-}st\ (set\text{-}literals\text{-}to\text{-}update\text{-}wl\ WS\ S) = all\text{-}lits\text{-}st\ S\rangle$
by (*cases S*; *auto simp*: *all-lits-st-def all-lits-def all-lits-of-mm-union ran-m-clause-upd image-mset-remove1-mset-if*; *fail*)+

lemma *in-clause-in-all-lits-of-mm*[*simp*]:
$\langle x1 \in\#\ dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow$
$x1aa \propto x1\ !\ n \in\#\ all\text{-}lits\text{-}st\ (x1b, x1aa, D, x1c, x1d, NS, US, N0, U0, x1e, x2e)\rangle$
by (*auto simp*: *all-lits-st-def all-lits-def all-lits-of-mm-union ran-m-clause-upd all-lits-of-mm-add-mset in-clause-in-all-lits-of-m image-mset-remove1-mset-if ran-m-def dest*!: *multi-member-split*)

lemma *correct-watching-except-correct-watching*:
assumes
j: $\langle j \geq length\ (W\ K)\rangle$ **and**
corr: $\langle correct\text{-}watching\text{-}except\ i\ j\ K\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
shows $\langle correct\text{-}watching\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(K := take\ i\ (W\ K)))\rangle$

proof −
have
H1: ‹⋀*L i' K' b*. *L* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟹
(*L* = *K* ⟹
distinct-watched (*take i* (*W L*) @ *drop j* (*W L*)) ∧
(((*i'*, *K'*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)) ⟶ *i'* ∈# *dom-m N* ⟶
K' ∈ *set* (*N* ∝ *i'*) ∧ *K'* ≠ *L* ∧ *correctly-marked-as-binary N* (*i'*, *K'*, *b*)) ∧
((*i'*, *K'*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)) ⟶ *b* ⟶ *i'* ∈# *dom-m N*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* '# *mset* (*take i* (*W L*) @ *drop j* (*W L*))) =
clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))))› **and**
H2: ‹⋀*L i K' b*. *L* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟹ (*L* ≠ *K* ⟹
distinct-watched (*W L*) ∧
(((*i*, *K'*, *b*)∈#*mset* (*W L*) ⟶ *i* ∈# *dom-m N* ⟶ *K'* ∈ *set* (*N* ∝ *i*) ∧ *K'* ≠ *L* ∧
(*correctly-marked-as-binary N* (*i*, *K'*, *b*))) ∧
((*i*, *K'*, *b*)∈#*mset* (*W L*) ⟶ *b* ⟶ *i* ∈# *dom-m N*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* '# *mset* (*W L*)) =
clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))))›
using *corr* **unfolding** *correct-watching-except.simps*
by *fast*+
show *?thesis*
unfolding *correct-watching.simps*
apply (*intro conjI allI impI ballI*)
subgoal for *L*
apply (*cases* ‹*L* = *K*›)
subgoal
using *H1*[*of L*] *j*
by (*auto split*: *if-splits simp*: *all-lits-st-def*)
subgoal
using *H2*[*of L*] *j*
by (*auto split*: *if-splits simp*: *all-lits-st-def*)
done
subgoal for *L x*
apply (*cases* ‹*L* = *K*›)
subgoal
using *H1*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j*
by (*auto split*: *if-splits simp*: *all-lits-st-def*)
subgoal
using *H2*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›]
by *auto*
done
subgoal for *L*
apply (*cases* ‹*L* = *K*›)
subgoal
using *H1*[*of L* - -] *j*
by (*auto split*: *if-splits simp*: *all-lits-st-def*)
subgoal
using *H2*[*of L* - -]
by *auto*
done
subgoal for *L*
apply (*cases* ‹*L* = *K*›)
subgoal
using *H1*[*of L* - -] *j*
by (*auto split*: *if-splits*)
subgoal

using *H2*[*of L - -*]
by *auto*
done
done
qed

lemma *length-ge2I*: ‹$x \neq y \Longrightarrow x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow length\ xs \geq 2$›
using *card-length*[*of xs*] *card-mono*[*of* ‹*set xs*› ‹$\{x, y\}$›]
by *auto*

lemma *correct-watching-except-alt-def*:
‹*correct-watching-except i j K* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) \longleftrightarrow$
$(\forall L \in\# all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W).$
$(L = K \longrightarrow$
distinct-watched $(take\ i\ (W\ L) @ drop\ j\ (W\ L)) \wedge$
$((\forall (i, K, b) \in\# mset\ (take\ i\ (W\ L) @ drop\ j\ (W\ L)).\ i \in\# dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge$
$K \neq L \wedge L \in set\ (watched\text{-}l\ (N \propto i)) \wedge length\ (N \propto i) \geq 2 \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i, K, b)) \wedge$
$(\forall (i, K, b) \in\# mset\ (take\ i\ (W\ L) @ drop\ j\ (W\ L)).\ b \longrightarrow i \in\# dom\text{-}m\ N) \wedge$
filter-mset $(\lambda i.\ i \in\# dom\text{-}m\ N)\ (fst\ `\#\ mset\ (take\ i\ (W\ L) @ drop\ j\ (W\ L))) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\})) \wedge$
$(L \neq K \longrightarrow$
distinct-watched $(W\ L) \wedge$
$((\forall (i, K, b) \in\# mset\ (W\ L).\ i \in\# dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge K \neq L \wedge L \in set\ (watched\text{-}l(N \propto i)) \wedge length\ (N \propto i) \geq 2 \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i, K, b)) \wedge$
$(\forall (i, K, b) \in\# mset\ (W\ L).\ b \longrightarrow i \in\# dom\text{-}m\ N) \wedge$
filter-mset $(\lambda i.\ i \in\# dom\text{-}m\ N)\ (fst\ `\#\ mset\ (W\ L)) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))))$›
proof −
have *1*: ‹*correct-watching-except i j K* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) \longleftrightarrow$
$(\forall L \in\# all\text{-}lits\text{-}st\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W).$
$(L = K \longrightarrow$
distinct-watched $(take\ i\ (W\ L) @ drop\ j\ (W\ L)) \wedge$
$((\forall (i, K, b) \in\# mset\ (take\ i\ (W\ L) @ drop\ j\ (W\ L)).\ i \in\# dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge$
$K \neq L \wedge L \in set\ (watched\text{-}l\ (N \propto i)) \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i, K, b)) \wedge$
$(\forall (i, K, b) \in\# mset\ (take\ i\ (W\ L) @ drop\ j\ (W\ L)).\ b \longrightarrow i \in\# dom\text{-}m\ N) \wedge$
filter-mset $(\lambda i.\ i \in\# dom\text{-}m\ N)\ (fst\ `\#\ mset\ (take\ i\ (W\ L) @ drop\ j\ (W\ L))) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\})) \wedge$
$(L \neq K \longrightarrow$
distinct-watched $(W\ L) \wedge$
$((\forall (i, K, b) \in\# mset\ (W\ L).\ i \in\# dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge K \neq L \wedge L \in set\ (watched\text{-}l\ (N \propto i)) \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i, K, b)) \wedge$
$(\forall (i, K, b) \in\# mset\ (W\ L).\ b \longrightarrow i \in\# dom\text{-}m\ N) \wedge$
filter-mset $(\lambda i.\ i \in\# dom\text{-}m\ N)\ (fst\ `\#\ mset\ (W\ L)) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))))$›
unfolding *correct-watching-except.simps*
apply (*intro impI ballI conjI iffI*)
subgoal by *auto*[]
defer
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal for *K x*
using *distinct-mset-dom*[*of N*]
apply (*clarsimp dest*!: *multi-member-split simp*: *ran-m-def clause-to-update-def*)
apply (*frule bspec*[*of* ‹*set* (*W K*)›], *assumption*)

apply (*auto split*: *if-splits dest*: *in-set-takeD union-single-eq-member filter-mset-eq-add-msetD*
simp: *in-set-conv-decomp*[*of* - ‹*W K*›] *filter-union-or-split filter-eq-replicate-mset*)
done
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal
using *distinct-mset-dom*[*of N*]
apply (*clarsimp dest*!: *multi-member-split simp*: *ran-m-def clause-to-update-def*)
apply (*fastforce split*: *if-splits dest*: *in-set-takeD union-single-eq-member*
simp: *in-set-conv-decomp*[*of* ‹(-, -, -)› ‹*take* - -›] *in-set-conv-decomp*[*of* ‹(-, -, -)› ‹*drop* - -›])[]
done
done
have *H*: ‹(*K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *L* ∈ *set* (*watched-l* (*N* ∝ *i*)) ∧ *P*) ⟷
(*K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *L* ∈ *set* (*watched-l* (*N* ∝ *i*)) ∧ *length* (*N* ∝ *i*) ≥ *2* ∧ *P*)› **for** *i K P L*
using *length-ge2I*[*of K L*] **by** (*auto dest*: *in-set-takeD*)
show *?thesis*
unfolding *1 H*[*symmetric*] **..**
qed

definition *clause-to-update-wl*:: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ '*v clauses-to-update-l*›**where**
‹*clause-to-update-wl L S* =
filter-mset
(λ*C*::*nat*. *L* ∈ *set* (*watched-l* (*get-clauses-wl S* ∝ *C*)))
(*dom-m* (*get-clauses-wl S*))›

fun *watched-by* :: ‹'*v twl-st-wl* ⇒ '*v literal* ⇒ '*v watched*› **where**
‹*watched-by* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) *L* = *W L*›

lemma *watched-by-alt-def*: ‹*watched-by S L* = *get-watched-wl S L*›
by (*cases S*) *auto*

definition *all-atms* :: ‹- ⇒ - ⇒ '*v multiset*› **where**
‹*all-atms N NUE* = *atm-of* `# *all-lits N NUE*›

definition *all-atms-st* :: ‹'*v twl-st-wl* ⇒ '*v multiset*› **where**
‹*all-atms-st S* ≡ *all-atms* (*get-clauses-wl S*) (*get-unit-clauses-wl S* + *get-subsumed-clauses-wl S* + *get-clauses0-wl S*)›

lemma *all-atms-st-alt-def*: ‹*all-atms-st S* = *atm-of* `# *all-lits-st S*›
by (*auto simp*: *all-atms-def all-lits-st-def all-atms-st-def*)

lemmas *all-atms-st-alt-def-sym*[*simp*] = *all-atms-st-alt-def*[*symmetric*]

lemma *in-all-lits-minus-iff*: ‹−*L* ∈# *all-lits N NUE* ⟷ *L* ∈# *all-lits N NUE*›
unfolding *all-lits-def in-all-lits-of-mm-uminus-iff* **..**

lemma *all-lits-of-all-atms-of*: ‹$K \in\# \mathit{all\text{-}lits}\ N\ \mathit{NUE} \longleftrightarrow K \in\# \mathcal{L}_{all}\ (\mathit{all\text{-}atms}\ N\ \mathit{NUE})$›
by (*simp add*: $\mathcal{L}_{all}$*-atm-of-all-lits-of-mm all-atms-def all-lits-def*)

lemma $\mathcal{L}_{all}$*-all-atms-all-lits*: ‹$\mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ (\mathit{all\text{-}atms}\ N\ \mathit{NE})) = \mathit{set\text{-}mset}\ (\mathit{all\text{-}lits}\ N\ \mathit{NE})$›
unfolding *all-atms-def* $\mathcal{L}_{all}$*-atm-of-all-lits-of-mm all-lits-def* **..**

definition *blits-in-*$\mathcal{L}_{in}$ :: ‹*'v twl-st-wl* ⇒ *bool*› **where**
‹*blits-in-*$\mathcal{L}_{in}$ $S = (\forall L \in\# \mathit{all\text{-}lits\text{-}st}\ S.\ \forall (i, K, b) \in \mathit{set}\ (\mathit{watched\text{-}by}\ S\ L).\ K \in\# \mathit{all\text{-}lits\text{-}st}\ S)$›

definition *literals-are-*$\mathcal{L}_{in}$ :: ‹*'v multiset* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}\ S \equiv (\mathit{is\text{-}}\mathcal{L}_{all}\ \mathcal{A}\ (\mathit{all\text{-}lits\text{-}st}\ S) \wedge \mathit{blits\text{-}in\text{-}}\mathcal{L}_{in}\ S)$›

lemma *literals-are-in-*$\mathcal{L}_{in}$*-nth*:
fixes C :: *nat*
assumes *dom*: ‹$C \in\# \mathit{dom\text{-}m}\ (\mathit{get\text{-}clauses\text{-}wl}\ S)$› **and**
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}\ S$›
shows ‹*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}\ (\mathit{mset}\ (\mathit{get\text{-}clauses\text{-}wl}\ S \propto C))$›
proof −
let *?N* = ‹*get-clauses-wl S*›
have ‹*?N* $\propto C \in\# \mathit{ran\text{-}mf}$ *?N*›
using *dom* **by** (*auto simp*: *ran-m-def*)
then have ‹$\mathit{mset}\ (?N \propto C) \in\# \mathit{mset}\ `\#\ (\mathit{ran\text{-}mf}\ ?N)$›
by *blast*
from *all-lits-of-m-subset-all-lits-of-mmD*[*OF this*] **show** *?thesis*
using *assms*(*2*) **unfolding** *is-*$\mathcal{L}_{all}$*-def literals-are-in-*$\mathcal{L}_{in}$*-def literals-are-*$\mathcal{L}_{in}$*-def*
by (*auto simp*: *all-lits-st-def all-lits-of-mm-union all-lits-alt-def*)
qed

lemma *literals-are-*$\mathcal{L}_{in}$*-set-mset-*$\mathcal{L}_{all}$[*simp*]:
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}\ S \Longrightarrow \mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ (\mathit{all\text{-}atms\text{-}st}\ S)) = \mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ \mathcal{A})$›
using *in-all-lits-of-mm-ain-atms-of-iff*
unfolding *set-mset-set-mset-eq-iff is-*$\mathcal{L}_{all}$*-def Ball-def in-*$\mathcal{L}_{all}$*-atm-of-in-atms-of-iff*
in-all-lits-of-mm-ain-atms-of-iff atms-of-$\mathcal{L}_{all}$*-*$\mathcal{A}_{in}$ *literals-are-*$\mathcal{L}_{in}$*-def all-lits-st-def*
all-lits-def
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff all-atms-def simp del*: *all-atms-st-alt-def-sym*
simp: *all-lits-def all-atms-st-def*)

lemma *is-*$\mathcal{L}_{all}$*-all-lits-st-*$\mathcal{L}_{all}$[*simp*]:
‹$\mathit{is\text{-}}\mathcal{L}_{all}\ \mathcal{A}\ (\mathit{all\text{-}lits\text{-}st}\ S) \Longrightarrow$
$\mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ (\mathit{all\text{-}atms\text{-}st}\ S)) = \mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ \mathcal{A})$›
‹$\mathit{is\text{-}}\mathcal{L}_{all}\ \mathcal{A}\ (\mathit{all\text{-}lits}\ N\ \mathit{NUE}) \Longrightarrow$
$\mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ (\mathit{all\text{-}atms}\ N\ \mathit{NUE})) = \mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ \mathcal{A})$›
‹$\mathit{is\text{-}}\mathcal{L}_{all}\ \mathcal{A}\ (\mathit{all\text{-}lits}\ N\ \mathit{NUE}) \Longrightarrow$
$\mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ (\mathit{atm\text{-}of}\ `\#\ \mathit{all\text{-}lits}\ N\ \mathit{NUE})) = \mathit{set\text{-}mset}\ (\mathcal{L}_{all}\ \mathcal{A})$›
using *in-all-lits-of-mm-ain-atms-of-iff*
unfolding *set-mset-set-mset-eq-iff is-*$\mathcal{L}_{all}$*-def Ball-def in-*$\mathcal{L}_{all}$*-atm-of-in-atms-of-iff*
in-all-lits-of-mm-ain-atms-of-iff atms-of-$\mathcal{L}_{all}$*-*$\mathcal{A}_{in}$ *all-lits-st-def all-atms-st-def*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff all-lits-def all-atms-def*)

lemma *in-set-all-atms-iff*:
‹$y \in\# \mathit{all\text{-}atms}\ \mathit{bu}\ \mathit{bw} \longleftrightarrow$
$y \in \mathit{atms\text{-}of\text{-}mm}\ (\mathit{mset}\ `\#\ \mathit{ran\text{-}mf}\ \mathit{bu}) \vee y \in \mathit{atms\text{-}of\text{-}mm}\ \mathit{bw}$›

by (*auto simp*: *all-atms-def all-lits-def in-all-lits-of-mm-ain-atms-of-iff*
atm-of-all-lits-of-mm)

lemma *blits-in-$\mathcal{L}_{in}$-keep-watch*:
assumes ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)› **and**
w:‹*w* < *length* (*watched-by* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*) *K*)›
shows ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g* (*K* := (*g K*)[*j* := *g K* ! *w*]))›
proof −
let *?S* = ‹(*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
let *?T* = ‹(*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g* (*K* := (*g K*)[*j* := *g K* ! *w*]))›
let *?g* = ‹*g* (*K* := (*g K*)[*j* := *g K* ! *w*])›
have *H*: ‹⋀*L i K b*. *L*∈# *all-lits-st ?S* ⟹ (*i*, *K*, *b*) ∈*set* (*g L*) ⟹
K ∈# *all-lits-st ?S*›
using *assms*
unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
by *blast*
have ‹ *L*∈# *all-lits-st ?S* ⟹ (*i*, *K'*, *b'*) ∈*set* (*?g L*) ⟹
K' ∈# *all-lits-st ?S*› **for** *L i K' b'*
using *H*[*of L i K'*] *H*[*of L* ‹*fst* (*g K* ! *w*)› ‹*fst* (*snd* (*g K* ! *w*))›]
nth-mem[*OF w*]
unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
by (*cases* ‹*j* < *length* (*g K*)›; *cases* ‹*g K* ! *w*›)
(*auto split*: *if-splits elim*!: *in-set-upd-cases*)
moreover have ‹*all-atms-st ?S* = *all-atms-st ?T*›
by (*auto simp*: *all-lits-def all-atms-def all-atms-st-def*)
ultimately show *?thesis*
unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
by *force*
qed

lemma *all-lits-swap*[*simp*]:
‹*x1* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1*) ⟹ *n'* < *length* (*x1aa* ∝ *x1*) ⟹
all-lits
(*x1aa*(*x1* ↪ *swap* (*x1aa* ∝ *x1*) *n n'*))
(*x1cx1d*) =
all-lits x1aa (*x1cx1d*)›
using *distinct-mset-dom*[*of x1aa*]
apply (*auto simp*: *ran-m-def all-lits-def dest*!: *multi-member-split*)
apply (*subst image-mset-cong*[*of* - ‹%*x*. *mset*
(*fst* (*the* (*fmlookup x1aa x*)))›])
apply *auto*
done

lemma *blits-in-$\mathcal{L}_{in}$-propagate*:
‹*x1* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1*) ⟹ *n'* < *length* (*x1aa* ∝ *x1*) ⟹
blits-in-$\mathcal{L}_{in}$ (*Propagated A x1'* # *x1b*, *x1aa*
(*x1* ↪ *swap* (*x1aa* ∝ *x1*) *n n'*), *D*, *x1c*, *x1d*,
NEk, *UEk*, *NS*, *US*, *N0*, *U0*, *add-mset A' x1e*, *x2e*) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)›
‹*x1* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1*) ⟹ *n'* < *length* (*x1aa* ∝ *x1*) ⟹
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*
(*x1* ↪ *swap* (*x1aa* ∝ *x1*) *n n'*), *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)›
‹*blits-in-$\mathcal{L}_{in}$*
(*Propagated A x1'* # *x1b*, *x1aa*, *D*, *x1c*, *x1d*,

NEk, *UEk*, *NS*, *US*, *N0*, *U0*, *add-mset A′ x1e*, *x2e*) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)›
‹*x1′* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1′*) ⟹ *n′* < *length* (*x1aa* ∝ *x1′*) ⟹
K ∈# *all-lits-st* (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹ *blits-in-$\mathcal{L}_{in}$*
(*x1a*, *x1aa*(*x1′* ↪ *swap* (*x1aa* ∝ *x1′*) *n n′*), *D*, *x1c*, *x1d*,
NEk, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*
(*x1aa* ∝ *x1′* ! *n′* :=
x2e (*x1aa* ∝ *x1′* ! *n′*) @ [(*x1′*, *K*, *b′*)])) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)›
‹*K* ∈# *all-lits-st* (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*,
NEk, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*
(*K′* := *x2e K′* @ [(*x1′*, *K*, *b′*)])) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)›
unfolding *blits-in-$\mathcal{L}_{in}$-def*
by (*auto split*: *if-splits*)

lemma *blits-in-$\mathcal{L}_{in}$-keep-watch′*:
assumes *K′*: ‹*fst* (*snd w*) ∈# *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)› **and**
w:‹ *blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
shows ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g* (*K* := (*g K*)[*j* := *w*]))›
proof −
let *?A* = ‹*all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
let *?g* = ‹*g* (*K* := (*g K*)[*j* := *w*])›
have *H*: ‹⋀*L i K b′*. *L*∈# *?A* ⟹ (*i*, *K*, *b′*) ∈*set* (*g L*) ⟹ *K* ∈# *?A*›
using *assms*
unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
by *blast*
have ‹*L*∈# *?A* ⟹ (*i*, *K′*, *b′*) ∈*set* (*?g L*) ⟹ *K′* ∈# *?A*› **for** *L i K′ b′*
using *H*[*of L i K′*] *K′*
unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
by (*cases* ‹*j* < *length* (*g K*)›)
(*auto split*: *if-splits elim*!: *in-set-upd-cases*)

then show *?thesis*
unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
by *force*
qed

lemma *blits-in-$\mathcal{L}_{in}$-keep-watch″*:
assumes *K′*: ‹*K′* ∈# *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
‹*L′* ∈# *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)› **and**
w:‹ *blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
shows ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*,
g (*K* := (*g K*)[*j* := (*i*, *K′*, *b′*)], *L* := *g L* @ [(*i′*, *L′*, *b″*)]))›
using *assms*
unfolding *blits-in-$\mathcal{L}_{in}$-def*
by (*auto split*: *if-splits elim*!: *in-set-upd-cases*)

lemma *clause-to-update-wl-alt-def*:
‹*clause-to-update-wl L* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*) =
clause-to-update L (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
unfolding *clause-to-update-wl-def clause-to-update-def* **by** *simp*

lemma *correct-watching-except-alt-def2*:

‹correct-watching-except i j K S ⟷
(∀ L ∈# all-lits-st S.
(L = K ⟶
distinct-watched (take i (watched-by S L) @ drop j (watched-by S L)) ∧
((∀ (i, K, b)∈#mset (take i (watched-by S L) @ drop j (watched-by S L)). i ∈# dom-m (get-clauses-wl S) ⟶ K ∈ set (get-clauses-wl S ∝ i) ∧
K ≠ L ∧ L ∈ set (watched-l (get-clauses-wl S ∝ i)) ∧ length (get-clauses-wl S ∝ i) ≥ 2 ∧ correctly-marked-as-binary (get-clauses-wl S) (i, K, b)) ∧
(∀ (i, K, b)∈#mset (take i (watched-by S L) @ drop j (watched-by S L)). b ⟶ i ∈# dom-m (get-clauses-wl S)) ∧
filter-mset (λi. i ∈# dom-m (get-clauses-wl S)) (fst ‘# mset (take i (watched-by S L) @ drop j (watched-by S L))) = clause-to-update-wl L S)) ∧
(L ≠ K ⟶
distinct-watched (watched-by S L) ∧
((∀ (i, K, b)∈#mset (watched-by S L). i ∈# dom-m (get-clauses-wl S) ⟶ K ∈ set (get-clauses-wl S ∝ i) ∧ K ≠ L ∧ L ∈ set (watched-l (get-clauses-wl S ∝ i)) ∧ length (get-clauses-wl S ∝ i) ≥ 2 ∧ correctly-marked-as-binary (get-clauses-wl S) (i, K, b)) ∧
(∀ (i, K, b)∈#mset (watched-by S L). b ⟶ i ∈# dom-m (get-clauses-wl S)) ∧
filter-mset (λi. i ∈# dom-m (get-clauses-wl S)) (fst ‘# mset (watched-by S L)) = clause-to-update-wl L S)))›
by (*cases S*; *hypsubst*)
(*simp only*: *correct-watching-except-alt-def watched-by.simps*
get-clauses-wl.simps clause-to-update-wl-alt-def get-unit-clauses-wl.simps)

fun *update-watched* :: ‹'v literal ⇒ 'v watched ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl› **where**
‹*update-watched L WL* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*(*L*:= *WL*))›

definition *mop-watched-by-at* :: ‹'v twl-st-wl ⇒ 'v literal ⇒ nat ⇒ 'v watcher nres› **where**
‹*mop-watched-by-at* = (λ*S L w*. *do* {
ASSERT(*L* ∈# *all-lits-st S*);
ASSERT(*w* < *length* (*watched-by S L*));
RETURN (*watched-by S L* ! *w*)
})›

lemma *bspec'*: ‹*x* ∈ *a* ⟹ ∀ *x*∈*a*. *P x* ⟹ *P x*›
by (*rule bspec*)

lemma *correct-watching-exceptD*:
assumes
‹*correct-watching-except i j L S*› **and**
‹*L* ∈# *all-lits-st S*› **and**
w: ‹*w* < *length* (*watched-by S L*)› ‹*w* ≥ *j*› ‹*fst* (*watched-by S L* ! *w*) ∈# *dom-m* (*get-clauses-wl S*)›
shows ‹*fst* (*snd* (*watched-by S L* ! *w*)) ∈ *set* (*get-clauses-wl S* ∝ (*fst* (*watched-by S L* ! *w*)))›
proof −
have *H*: ‹⋀*x*. *x*∈*set* (*take i* (*watched-by S L*)) ∪ *set* (*drop j* (*watched-by S L*)) ⟹
case x of (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m* (*get-clauses-wl S*) ⟶ *K* ∈ *set* (*get-clauses-wl S* ∝ *i*) ∧
K ≠ *L*›
using *assms multi-member-split*[*OF assms*(*2*)]
by (*cases S*; *cases* ‹*watched-by S L* ! *w*›)
(*clarsimp*, *blast*)
have ‹∃ *i*≥*j*. *i* < *length* (*watched-by S L*) ∧
watched-by S L ! *w* = *watched-by S L* ! *i*›
by (*rule exI*[*of* - *w*])
(*use w* **in** *auto*)

then show *?thesis*
using *H*[*of* ‹*watched-by S L ! w*›] *w*
by (*cases* ‹*watched-by S L ! w*›) (*auto simp*: *in-set-drop-conv-nth*)
qed

declare *correct-watching-except.simps*[*simp del*]

fun *st-l-of-wl* :: ‹('*v literal* × *nat*) *option* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-l*› **where**
‹*st-l-of-wl None* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) = (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›
| ‹*st-l-of-wl* (*Some* (*L*, *j*)) (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*,
(*if D* ≠ *None then* {#} *else clauses-to-update-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) *L j*,
Q))›

definition *state-wl-l* :: ‹('*v literal* × *nat*) *option* ⇒ ('*v twl-st-wl* × '*v twl-st-l*) *set*› **where**
‹*state-wl-l L* = {(*T*, *T'*). *T'* = *st-l-of-wl L T*}›

fun *twl-st-of-wl* :: ‹('*v literal* × *nat*) *option* ⇒ ('*v twl-st-wl* × '*v twl-st*) *set*› **where**
‹*twl-st-of-wl L* = *state-wl-l L O twl-st-l* (*map-option fst L*)›

named-theorems *twl-st-wl* ‹*Conversions simp rules*›

lemma [*twl-st-wl*]:
assumes ‹(*S*, *T*) ∈ *state-wl-l L*›
shows
‹*get-trail-l T* = *get-trail-wl S*› **and**
‹*get-clauses-l T* = *get-clauses-wl S*› **and**
‹*get-conflict-l T* = *get-conflict-wl S*› **and**
‹*L* = *None* ⟹ *clauses-to-update-l T* = {#}›
‹*L* ≠ *None* ⟹ *get-conflict-wl S* ≠ *None* ⟹ *clauses-to-update-l T* = {#}›
‹*L* ≠ *None* ⟹ *get-conflict-wl S* = *None* ⟹ *clauses-to-update-l T* =
clauses-to-update-wl S (*fst* (*the L*)) (*snd* (*the L*))› **and**
‹*literals-to-update-l T* = *literals-to-update-wl S*›
‹*get-unit-learned-clss-l T* = *get-unit-learned-clss-wl S*›
‹*get-unit-init-clauses-l T* = *get-unit-init-clss-wl S*›
‹*get-unit-learned-clss-l T* = *get-unit-learned-clss-wl S*›
‹*get-unit-clauses-l T* = *get-unit-clauses-wl S*›
‹*get-kept-unit-clauses-l T* = *get-kept-unit-clauses-wl S*›
‹*get-subsumed-init-clauses-l T* = *get-subsumed-init-clauses-wl S*›
‹*get-subsumed-learned-clauses-l T* = *get-subsumed-learned-clauses-wl S*›
‹*get-subsumed-clauses-l T* = *get-subsumed-clauses-wl S*›
‹*get-init-clauses0-l T* = *get-init-clauses0-wl S*›
‹*get-learned-clauses0-l T* = *get-learned-clauses0-wl S*›
‹*get-clauses0-l T* = *get-clauses0-wl S*›
‹*get-init-clss-l T* = *get-init-clss-wl S*›
‹*all-lits-of-st-l T* = *all-lits-st S*›
‹*get-unkept-learned-clss-l T* = *get-unkept-learned-clss-wl S*›
‹*all-lits-of-mm* (*get-all-clss-l T*) = *all-lits-st S*›
using *assms* **unfolding** *state-wl-l-def all-clss-lf-ran-m*[*symmetric*]
apply (*cases S*; *cases T*; *cases L*; *auto split*: *option.splits simp*: *get-init-clss-l-def*
all-lits-st-def all-lits-def all-lits-of-st-l-def ac-simps; *fail*)+
using *assms* **unfolding** *state-wl-l-def all-clss-lf-ran-m*[*symmetric*] *all-lits-st-def all-lits-def*

apply (*subst all-clss-lf-ran-m*)
by (*cases S*; *cases T*; *cases L*; *auto split*: *option.splits simp*: *get-init-clss-l-def ac-simps*)

lemma [*twl-st-wl*]:
‹(*a*, *a'*) ∈ *state-wl-l None* ⟹
get-learned-clss-l a' = *get-learned-clss-wl a*›
unfolding *state-wl-l-def* **by** (*cases a*; *cases a'*)
(*auto simp*: *get-learned-clss-l-def get-learned-clss-wl-def*)

lemma *remove-one-lit-from-wq-def*:
‹*remove-one-lit-from-wq L S* = *set-clauses-to-update-l* (*clauses-to-update-l S* − {#*L*#}) *S*›
by (*cases S*) *auto*

lemma *correct-watching-set-literals-to-update*[*simp*]:
‹*correct-watching* (*set-literals-to-update-wl WS T'*) = *correct-watching T'*›
by (*cases T'*) (*auto simp*: *correct-watching.simps*)

lemma [*twl-st-wl*]:
‹*get-clauses-wl* (*set-literals-to-update-wl W S*) = *get-clauses-wl S*›
‹*get-unit-init-clss-wl* (*set-literals-to-update-wl W S*) = *get-unit-init-clss-wl S*›
by (*cases S*; *auto*; *fail*)+

lemma *get-conflict-wl-set-literals-to-update-wl*[*twl-st-wl*]:
‹*get-conflict-wl* (*set-literals-to-update-wl P S*) = *get-conflict-wl S*›
‹*get-unit-clauses-wl* (*set-literals-to-update-wl P S*) = *get-unit-clauses-wl S*›
‹*get-init-clauses0-wl* (*set-literals-to-update-wl P S*) = *get-init-clauses0-wl S*›
‹*get-learned-clauses0-wl* (*set-literals-to-update-wl P S*) = *get-learned-clauses0-wl S*›
‹*get-clauses0-wl* (*set-literals-to-update-wl P S*) = *get-clauses0-wl S*›
by (*cases S*; *auto*; *fail*)+

definition *set-conflict-wl-pre* :: ‹*nat* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*set-conflict-wl-pre C S* ⟷
(∃ *S'* *b*. (*S*, *S'*) ∈ *state-wl-l b* ∧ *set-conflict-l-pre C S'* ∧ *blits-in-*$\mathcal{L}_{in}$ *S*)›

definition *set-conflict-wl* :: ‹*nat* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*set-conflict-wl* = (λ*C* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(*set-conflict-wl-pre C* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
RETURN (*M*, *N*, *Some* (*mset* (*N* ∝ *C*)), *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *W*)
})›

lemma *state-wl-l-mark-of-is-decided*:
‹(*x*, *y*) ∈ *state-wl-l b* ⟹
get-trail-wl x ≠ [] ⟹
is-decided (*hd* (*get-trail-l y*)) = *is-decided* (*hd* (*get-trail-wl x*))›
by (*cases* ‹*get-trail-wl x*›; *cases* ‹*get-trail-l y*›; *cases* ‹*hd* (*get-trail-wl x*)›;
cases ‹*hd* (*get-trail-l y*)›; *cases b*; *cases x*)
(*auto simp*: *state-wl-l-def convert-lit.simps st-l-of-wl.simps*)

lemma *state-wl-l-mark-of-is-proped*:
‹(*x*, *y*) ∈ *state-wl-l b* ⟹
get-trail-wl x ≠ [] ⟹
is-proped (*hd* (*get-trail-l y*)) = *is-proped* (*hd* (*get-trail-wl x*))›
by (*cases* ‹*get-trail-wl x*›; *cases* ‹*get-trail-l y*›; *cases* ‹*hd* (*get-trail-wl x*)›;
cases ‹*hd* (*get-trail-l y*)›; *cases b*; *cases x*)
(*auto simp*: *state-wl-l-def convert-lit.simps*)

We here also update the list of watched clauses *WL*.

declare *twl-st-wl*[*simp*]

lemma
  assumes
    x2-T: ‹(*x2*, *T*) ∈ *state-wl-l b*› **and**
    struct: ‹*twl-struct-invs U*› **and**
    T-U: ‹(*T*, *U*) ∈ *twl-st-l b'*›
  shows
   literals-are-$\mathcal{L}_{in}$-literals-are-$\mathcal{L}_{in}$-trail:
    ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2* ⟹ *literals-are-in-$\mathcal{L}_{in}$-trail $\mathcal{A}_{in}$* (*get-trail-wl x2*)›
   (**is** ‹-⟹ *?trail*›) **and**
   literals-are-$\mathcal{L}_{in}$-literals-are-in-$\mathcal{L}_{in}$-conflict:
   ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2* ⟹ *get-conflict-wl x2* ≠ *None* ⟹ *literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$* (*the* (*get-conflict-wl x2*))› **and**
   conflict-not-tautology:
    ‹*get-conflict-wl x2* ≠ *None* ⟹ ¬*tautology* (*the* (*get-conflict-wl x2*))›
proof −
  have
   alien: ‹*cdcl$_W$-restart-mset.no-strange-atm* (*state$_W$-of U*)› **and**
   confl: ‹*cdcl$_W$-restart-mset.cdcl$_W$-conflicting* (*state$_W$-of U*)› **and**
   M-lev: ‹*cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv* (*state$_W$-of U*)› **and**
   dist: ‹*cdcl$_W$-restart-mset.distinct-cdcl$_W$-state* (*state$_W$-of U*)›
   using *struct* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
    pcdcl-all-struct-invs-def state$_W$-of-def
  by *fast+*

  show *lits-trail*: ‹*literals-are-in-$\mathcal{L}_{in}$-trail $\mathcal{A}_{in}$* (*get-trail-wl x2*)›
   if ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2*›
   using *alien that x2-T T-U* **unfolding** *is-$\mathcal{L}_{all}$-def*
    literals-are-in-$\mathcal{L}_{in}$-trail-def cdcl$_W$-restart-mset.no-strange-atm-def
    literals-are-$\mathcal{L}_{in}$-def all-lits-def all-atms-def all-lits-st-def
   by (*subst* (*asm*) *all-clss-l-ran-m*[*symmetric*])
    (*auto 5 2*
      simp del: *all-clss-l-ran-m union-filter-mset-complement*
      simp: *twl-st twl-st-l twl-st-wl all-lits-of-mm-union lits-of-def*
      convert-lits-l-def image-image in-all-lits-of-mm-ain-atms-of-iff
      get-unit-clauses-wl-alt-def Un-assoc ac-simps
     simp flip: *state$_W$-of-def*)

  {
   assume *conf*: ‹*get-conflict-wl x2* ≠ *None*›
   show *lits-confl*: ‹*literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$* (*the* (*get-conflict-wl x2*))›
    if ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2*›
    using *x2-T T-U alien that conf* **unfolding** *is-$\mathcal{L}_{all}$-alt-def*
     cdcl$_W$-restart-mset.no-strange-atm-def literals-are-in-$\mathcal{L}_{in}$-alt-def
     literals-are-$\mathcal{L}_{in}$-def all-lits-def all-atms-def all-lits-st-def
    apply (*subst* (*asm*) *all-clss-l-ran-m*[*symmetric*])
    unfolding *image-mset-union all-lits-of-mm-union*
    by (*auto simp add*: *twl-st all-lits-of-mm-union lits-of-def*
       image-image in-all-lits-of-mm-ain-atms-of-iff
      in-all-lits-of-m-ain-atms-of-iff
      get-unit-clauses-wl-alt-def
      simp del: *all-clss-l-ran-m*

simp flip: $state_W$*-of-def*)

have *M-confl*: ‹*get-trail-wl x2* $\models$*as CNot* (*the* (*get-conflict-wl x2*))›
using *confl conf x2-T T-U* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
by (*auto 5 5 simp*: *twl-st true-annots-def*)
moreover have *n-d*: ‹*no-dup* (*get-trail-wl x2*)›
using *M-lev x2-T T-U* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st*)
ultimately show *4*: ‹¬*tautology* (*the* (*get-conflict-wl x2*))›
using *n-d M-confl*
by (*meson no-dup-consistentD tautology-decomp' true-annots-true-cls-def-iff-negation-in-model*)
}
qed

fun *equality-except-conflict-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*equality-except-conflict-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
(*M'*, *N'*, *D'*, *NE'*, *UE'*, *NEk'*, *UEk'*, *NS'*, *US'*, *N0'*, *U0'*, *WS'*, *Q'*) ⟷
M = *M'* ∧ *N* = *N'* ∧ *NE* = *NE'* ∧ *UE* = *UE'* ∧ *NEk* = *NEk'* ∧ *UEk* = *UEk'* ∧ *NS* = *NS'* ∧ *US* = *US'* ∧
N0 = *N0'* ∧ *U0* = *U0'* ∧ *WS* = *WS'* ∧ *Q* = *Q'*›

fun *equality-except-trail-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*equality-except-trail-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
(*M'*, *N'*, *D'*, *NE'*, *UE'*, *NEk'*, *UEk'*, *NS'*, *US'*, *N0'*, *U0'*, *WS'*, *Q'*) ⟷
N = *N'* ∧ *D* = *D'* ∧ *NE* = *NE'* ∧ *NEk* = *NEk'* ∧ *UEk* = *UEk'* ∧ *NS* = *NS'* ∧ *US* = *US'* ∧ *UE* = *UE'* ∧
N0 = *N0'* ∧ *U0* = *U0'* ∧ *WS* = *WS'* ∧ *Q* = *Q'*›

lemma *equality-except-conflict-wl-get-clauses-wl*:
‹*equality-except-conflict-wl S Y* ⟹ *get-clauses-wl S* = *get-clauses-wl Y*› **and**
equality-except-conflict-wl-get-trail-wl:
‹*equality-except-conflict-wl S Y* ⟹ *get-trail-wl S* = *get-trail-wl Y*› **and**
equality-except-trail-wl-get-conflict-wl:
‹*equality-except-trail-wl S Y* ⟹ *get-conflict-wl S* = *get-conflict-wl Y*› **and**
equality-except-trail-wl-get-clauses-wl:
‹*equality-except-trail-wl S Y*⟹ *get-clauses-wl S* = *get-clauses-wl Y*› **and**
equality-except-trail-wl-get-clauses0-wl:
‹*equality-except-trail-wl S Y*⟹ *get-clauses0-wl S* = *get-clauses0-wl Y*›
by (*cases S*; *cases Y*; *solves auto*)+

definition *unit-prop-body-wl-inv* **where**
‹*unit-prop-body-wl-inv T j i L* ⟷ (*i* < *length* (*watched-by T L*) ∧ *j* ≤ *i* ∧
L ∈# *all-lits-st T* ∧ *blits-in-*$\mathcal{L}_{in}$ *T* ∧
(*fst* (*watched-by T L* ! *i*) ∈# *dom-m* (*get-clauses-wl T*) ⟶
(∃ *T'*. (*T*, *T'*) ∈ *state-wl-l* (*Some* (*L*, *Suc i*)) ∧ *j* ≤ *i* ∧
unit-propagation-inner-loop-body-l-inv L (*fst* (*watched-by T L* ! *i*)) *T'* ∧
correct-watching-except (*Suc j*) (*Suc i*) *L T*)))›

lemma *unit-prop-body-wl-inv-alt-def*:
‹*unit-prop-body-wl-inv T j i L* ⟷ (*i* < *length* (*watched-by T L*) ∧ *j* ≤ *i* ∧
L ∈# *all-lits-st T* ∧ *blits-in-*$\mathcal{L}_{in}$ *T* ∧
(*fst* (*watched-by T L* ! *i*) ∈# *dom-m* (*get-clauses-wl T*) ⟶
(∃ *T'*. (*T*, *T'*) ∈ *state-wl-l* (*Some* (*L*, *Suc i*)) ∧

```
    unit-propagation-inner-loop-body-l-inv L (fst (watched-by T L ! i)) T' ∧
     correct-watching-except (Suc j) (Suc i) L T ∧
    get-conflict-wl T = None ∧
    length (get-clauses-wl T ∝ fst (watched-by T L ! i)) ≥ 2))›
  (is ‹?A = ?B›)
proof
  assume ?B
  then show ?A
    unfolding unit-prop-body-wl-inv-def
    by blast
next
  assume ?A
  then show ?B
  proof (cases ‹fst (watched-by T L ! i) ∈# dom-m (get-clauses-wl T)›)
    case False
    then show ?B
      using ‹?A› unfolding unit-prop-body-wl-inv-def
      by blast
  next
    case True
    then obtain T' where
      ‹i < length (watched-by T L)›
      ‹j ≤ i› and
      TT': ‹(T, T') ∈ state-wl-l (Some (L, Suc i))› and
      inv: ‹unit-propagation-inner-loop-body-l-inv L (fst (watched-by T L ! i))
         T'› and
      ‹L ∈# all-lits-st T›
      ‹correct-watching-except (Suc j) (Suc i) L T›
      using ‹?A› unfolding unit-prop-body-wl-inv-def
      by blast

    obtain x where
      x: ‹(set-clauses-to-update-l
        (clauses-to-update-l T' + {#fst (watched-by T L ! i)#}) T', x)
       ∈ twl-st-l (Some L)› and
      struct-invs: ‹twl-struct-invs x› and
      ‹twl-stgy-invs x› and
      ‹fst (watched-by T L ! i) ∈# dom-m (get-clauses-l T')› and
      ‹0 < fst (watched-by T L ! i)› and
      ‹0 < length (get-clauses-l T' ∝ fst (watched-by T L ! i))› and
      ‹no-dup (get-trail-l T')› and
      ‹(if get-clauses-l T' ∝ fst (watched-by T L ! i) ! 0 = L then 0 else 1)
       < length (get-clauses-l T' ∝ fst (watched-by T L ! i))› and
      ‹1 - (if get-clauses-l T' ∝ fst (watched-by T L ! i) ! 0 = L then 0 else 1)
       < length (get-clauses-l T' ∝ fst (watched-by T L ! i))› and
      ‹L ∈ set (watched-l (get-clauses-l T' ∝ fst (watched-by T L ! i)))› and
      confl: ‹get-conflict-l T' = None›
      using inv unfolding unit-propagation-inner-loop-body-l-inv-def by blast

    have ‹Multiset.Ball (get-clauses x) struct-wf-twl-cls›
      using struct-invs unfolding twl-struct-invs-def twl-st-inv-alt-def by blast
    moreover have ‹twl-clause-of (get-clauses-wl T ∝ fst (watched-by T L ! i)) ∈# get-clauses x›
      using TT' x True by auto
    ultimately have 1: ‹length (get-clauses-wl T ∝ fst (watched-by T L ! i)) ≥ 2›
      by auto
    have 2: ‹get-conflict-wl T = None›
```

```
    using confl TT' x by auto
  show ?B
    using ‹?A› 1 2 unfolding unit-prop-body-wl-inv-def
    by blast
 qed
qed

definition propagate-lit-wl-general :: ‹'v literal ⇒ nat ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
 ‹propagate-lit-wl-general = (λL' C i (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). do {
    ASSERT(C ∈# dom-m N);
    ASSERT(D = None);
    ASSERT(L' ∈# all-lits-st (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W));
    ASSERT(i ≤ 1);
    M ← cons-trail-propagate-l L' C M;
    N ← (if length (N ∝ C) > 2 then mop-clauses-swap N C 0 (Suc 0 − i) else RETURN N);
    RETURN (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, add-mset (−L') Q, W) })›

definition propagate-lit-wl :: ‹'v literal ⇒ nat ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
 ‹propagate-lit-wl = (λL' C i (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). do {
    ASSERT(C ∈# dom-m N);
    ASSERT(D = None);
    ASSERT(L' ∈# all-lits-st (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W));
    ASSERT(i ≤ 1);
    M ← cons-trail-propagate-l L' C M;
    N ← mop-clauses-swap N C 0 (Suc 0 − i);
    RETURN (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, add-mset (−L') Q, W)
  })›

definition propagate-lit-wl-bin :: ‹'v literal ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
 ‹propagate-lit-wl-bin = (λL' C (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). do {
    ASSERT(D = None);
    ASSERT(C ∈# dom-m N);
    ASSERT(L' ∈# all-lits-st (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W));
    M ← cons-trail-propagate-l L' C M;
    RETURN (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, add-mset (−L') Q, W)})›

definition propagate-lit-wl-bin' ::‹ 'v literal ⇒ nat ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres›
  where
‹propagate-lit-wl-bin' L' C i = propagate-lit-wl-bin L' C›

definition keep-watch :: ‹'v literal ⇒ nat ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl› where
 ‹keep-watch = (λL i j (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W).
    (M, N,  D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(L := (W L)[i := W L ! j])))›

definition mop-keep-watch :: ‹'v literal ⇒ nat ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
 ‹mop-keep-watch = (λL i j S. do {
   ASSERT(L ∈# all-lits-st S);
   ASSERT(i < length (watched-by S L));
   ASSERT(j < length (watched-by S L));
   RETURN (keep-watch L i j S)
 })›

lemma length-watched-by-keep-watch[twl-st-wl]:
 ‹length (watched-by (keep-watch L i j S) K) = length (watched-by S K)›
 by (cases S) (auto simp: keep-watch-def)
```

lemma *watched-by-keep-watch-neq*[*twl-st-wl*, *simp*]:
‹*w* < *length* (*watched-by S L*) ⟹ *watched-by* (*keep-watch L j w S*) *L* ! *w* = *watched-by S L* ! *w*›
by (*cases S*) (*auto simp*: *keep-watch-def*)

lemma *watched-by-keep-watch-eq*[*twl-st-wl*, *simp*]:
‹*j* < *length* (*watched-by S L*) ⟹ *watched-by* (*keep-watch L j w S*) *L* ! *j* = *watched-by S L* ! *w*›
by (*cases S*) (*auto simp*: *keep-watch-def*)

definition *update-clause-wl* :: ‹'*v literal* ⇒ '*v literal* ⇒ *nat* ⇒ *bool* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*v twl-st-wl* ⇒
(*nat* × *nat* × '*v twl-st-wl*) *nres*› **where**
‹*update-clause-wl* = (λ(*L*::'*v literal*) *other-watched C b j w i f* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(*C* ∈# *dom-m N* ∧ *j* ≤ *w* ∧ *w* < *length* (*W L*) ∧
correct-watching-except (*Suc j*) (*Suc w*) *L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
ASSERT(*L* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
ASSERT(*other-watched* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
K' ← *mop-clauses-at N C f*;
ASSERT(*K'* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ∧ *L* ≠ *K'*);
N' ← *mop-clauses-swap N C i f*;
RETURN (*j*, *w*+*1*, (*M*, *N'*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*(*K'* := *W K'* @ [(*C*, *L*, *b*)]))))
})›

definition *update-blit-wl* :: ‹'*v literal* ⇒ *nat* ⇒ *bool* ⇒ *nat* ⇒ *nat* ⇒ '*v literal* ⇒ '*v twl-st-wl* ⇒
(*nat* × *nat* × '*v twl-st-wl*) *nres*› **where**
‹*update-blit-wl* = (λ(*L*::'*v literal*) *C b j w K* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(*L* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
ASSERT(*K* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
ASSERT(*j* ≤ *w*);
ASSERT(*w* < *length* (*W L*));
ASSERT(*C* ∈# *dom-m N*);
RETURN (*j*+*1*, *w*+*1*, (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*(*L* := (*W L*)[*j*:=(*C*, *K*, *b*)])))
})›

definition *unit-prop-body-wl-find-unwatched-inv* **where**
‹*unit-prop-body-wl-find-unwatched-inv f C S* ⟷ *True*›

abbreviation *remaining-nondom-wl* **where**
‹*remaining-nondom-wl w L S* ≡
(*if get-conflict-wl S* = *None*
then size (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl S*)) (*mset* (*drop w* (*watched-by S L*)))) *else* 0)›

definition *unit-propagation-inner-loop-wl-loop-inv* **where**
‹*unit-propagation-inner-loop-wl-loop-inv L* = (λ(*j*, *w*, *S*).
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *w*)) ∧ *j*≤ *w* ∧
unit-propagation-inner-loop-l-inv L (*S'*, *remaining-nondom-wl w L S*) ∧
correct-watching-except j w L S ∧ *w* ≤ *length* (*watched-by S L*)))›

lemma *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*:
assumes
j-w: ‹$j \leq w$› **and**
w-le: ‹$w <$ *length* (*watched-by S L*)› **and**
corr: ‹*correct-watching-except j w L S*›
shows ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L* (*keep-watch L j w S*)›
proof −
obtain *M N D NE UE NEk UEk NS US N0 U0 Q W* **where** *S*: ‹$S = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)$› **by** (*cases S*)
let *?L* = ‹*all-lits-st S*›
have
Hneq: ‹$\bigwedge La.\ La \in\# ?\mathcal{L} \longrightarrow$
$(La \neq L \longrightarrow$
distinct-watched (*W La*) ∧
$(\forall (i, K, b) \in\#$ *mset* (*W La*). $i \in\#$ *dom-m N* $\longrightarrow K \in$ *set* $(N \propto i) \wedge K \neq La \wedge$
correctly-marked-as-binary N $(i, K, b)) \wedge$
$(\forall (i, K, b) \in\#$ *mset* (*W La*). $b \longrightarrow i \in\#$ *dom-m N*) ∧
$\{\#i \in\#$ *fst* ‘# *mset* (*W La*). $i \in\#$ *dom-m N*$\#\} =$ *clause-to-update La* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))$› **and**
Heq: ‹$\bigwedge La.\ La \in\# ?\mathcal{L} \longrightarrow$
$(La = L \longrightarrow$
distinct-watched (*take j* (*W La*) @ *drop w* (*W La*)) ∧
$(\forall (i, K, b) \in\#$ *mset* (*take j* (*W La*) @ *drop w* (*W La*)). $i \in\#$ *dom-m N* $\longrightarrow K \in$ *set* $(N \propto i) \wedge$
$K \neq La \wedge$ *correctly-marked-as-binary N* $(i, K, b)) \wedge$
$(\forall (i, K, b) \in\#$ *mset* (*take j* (*W La*) @ *drop w* (*W La*)). $b \longrightarrow i \in\#$ *dom-m N*) ∧
$\{\#i \in\#$ *fst* ‘# *mset* (*take j* (*W La*) @ *drop w* (*W La*)). $i \in\#$ *dom-m N*$\#\} =$
clause-to-update La $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))$›
using *corr* **unfolding** *S correct-watching-except.simps*
by *fast+*

have *eq*: ‹*mset* (*take* (*Suc j*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @ *drop* (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)) =
mset (*take j* (*W La*) @ *drop w* (*W La*))› **if** [*simp*]: ‹*La* = *L*› **for** *La*
using *w-le j-w*
by (*auto simp*: *S take-Suc-conv-app-nth Cons-nth-drop-Suc*[*symmetric*]
list-update-append)

have ‹*case x of* $(i, K, b) \Rightarrow i \in\#$ *dom-m N* $\longrightarrow K \in$ *set* $(N \propto i) \wedge K \neq La \wedge$
correctly-marked-as-binary N (i, K, b)›
if
‹*La* $\in\#$ *?L*› **and**
‹*La* = *L*› **and**
‹*x* $\in\#$ *mset* (*take* (*Suc j*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))›
for *La* :: ‹*'a literal*› **and** *x* :: ‹*nat* × *'a literal* × *bool*›
using *that Heq*[*of L*]
apply (*subst* (*asm*) *eq*)
by (*simp-all add*: *eq*)
moreover have ‹*case x of* $(i, K, b) \Rightarrow b \longrightarrow i \in\#$ *dom-m N*›
if
‹*La* $\in\#$ *?L*› **and**
‹*La* = *L*› **and**
‹*x* $\in\#$ *mset* (*take* (*Suc j*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))›
for *La* :: ‹*'a literal*› **and** *x* :: ‹*nat* × *'a literal* × *bool*›
using *that Heq*[*of L*]

```
  by (subst (asm) eq) blast+
moreover have ‹{#i ∈# fst `#
          mset
           (take (Suc j) ((W(L := (W L)[j := W L ! w])) La) @
            drop (Suc w) ((W(L := (W L)[j := W L ! w])) La)).
     i ∈# dom-m N#} =
    clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})›
  if
    ‹La ∈# ?L› and
    ‹La = L›
  for La :: ‹'a literal›
  using that Heq[of L]
  by (subst eq) simp-all
moreover have ‹case x of (i, K, b) ⇒ i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧
      correctly-marked-as-binary N (i, K, b)›
  if
    ‹La ∈# ?L› and
    ‹La ≠ L› and
    ‹x ∈# mset ((W(L := (W L)[j := W L ! w])) La)›
  for La :: ‹'a literal› and x :: ‹nat × 'a literal × bool›
  using that Hneq[of La]
  by simp
moreover have ‹case x of (i, K, b) ⇒ b ⟶ i ∈# dom-m N›
  if
    ‹La ∈# ?L› and
    ‹La ≠ L› and
    ‹x ∈# mset ((W(L := (W L)[j := W L ! w])) La)›
  for La :: ‹'a literal› and x :: ‹nat × 'a literal × bool›
  using that Hneq[of La]
  by auto
moreover have ‹{#i ∈# fst `# mset ((W(L := (W L)[j := W L ! w])) La). i ∈# dom-m N#} =
    clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})›
  if
    ‹La ∈# ?L› and
    ‹La ≠ L›
  for La :: ‹'a literal›
  using that Hneq[of La]
  by simp
moreover have ‹distinct-watched ((W(L := (W L)[j := W L ! w])) La)›
  if
    ‹La ∈# ?L› and
    ‹La ≠ L›
  for La :: ‹'a literal›
  using that Hneq[of La]
  by simp
moreover have ‹distinct-watched (take (Suc j) ((W(L := (W L)[j := W L ! w])) La) @
          drop (Suc w) ((W(L := (W L)[j := W L ! w])) La))›
  if
    ‹La ∈# ?L› and
    ‹La = L›
  for La :: ‹'a literal›
  using that Heq[of La]
  apply (subst distinct-mset-mset-distinct[symmetric])
  apply (subst mset-map)
  apply (subst eq)
  apply (simp add: that)
```

apply (*subst mset-map*[*symmetric*])
apply (*subst distinct-mset-mset-distinct*)
apply *simp*
done
ultimately show *?thesis*
unfolding *S keep-watch-def prod.simps correct-watching-except.simps all-lits-st-simps*
by *meson*
qed

lemma *correct-watching-except-update-blit*:
assumes
corr: ‹*correct-watching-except i j L* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)]))› **and**
C': ‹*C'* ∈# *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
‹*C'* ∈ *set* (*b* ∝ *x1*)›
‹*C'* ≠ *L*› **and**
corr-watched: ‹*correctly-marked-as-binary b* (*x1*, *C'*, *b'*)›
shows ‹*correct-watching-except i j L* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*(*L* := (*g L*)[*j'* := (*x1*, *C'*, *b'*)]))›
proof −
let *?L* = ‹*all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
have
Hdisteq: ‹⋀*La i' K' b''*. *La*∈#*?L* ⟹
(*La* = *L* ⟶
distinct-watched (*take i* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*) @ *drop j* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)))› **and**
Heq: ‹⋀*La i' K' b''*. *La*∈#*?L* ⟹
(*La* = *L* ⟶
((*i'*, *K'*, *b''*)∈#*mset* (*take i* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*) @ *drop j* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)) ⟶
i' ∈# *dom-m b* ⟶ *K'* ∈ *set* (*b* ∝ *i'*) ∧ *K'* ≠ *La* ∧ *correctly-marked-as-binary b* (*i'*, *K'*, *b''*)) ∧
((*i'*, *K'*, *b''*)∈#*mset* (*take i* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*) @ *drop j* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)) ⟶
b'' ⟶ *i'* ∈# *dom-m b*)) ∧
{#*i* ∈# *fst* ‘# *mset* (*take i* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*) @ *drop j* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)).
i ∈# *dom-m b*#} =
clause-to-update La (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))› **and**
Hdistneq: ‹⋀*La i' K' b''*. *La*∈#*?L* ⟹
(*La* ≠ *L* ⟶ *distinct-watched* (((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)))› **and**
Hneq: ‹⋀*La i K b''*. *La*∈#*?L* ⟹ *La* ≠ *L* ⟹
distinct-watched (((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)) ∧
((*i*, *K*, *b''*)∈#*mset* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)⟶ *i* ∈# *dom-m b* ⟶
K ∈ *set* (*b* ∝ *i*) ∧ *K* ≠ *La* ∧ *correctly-marked-as-binary b* (*i*, *K*, *b''*)) ∧
((*i*, *K*, *b''*)∈#*mset* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*)⟶ *b''* ⟶ *i* ∈# *dom-m b*) ∧
{#*i* ∈# *fst* ‘# *mset* ((*g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])) *La*). *i* ∈# *dom-m b*#} =
clause-to-update La (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *corr* **unfolding** *correct-watching-except.simps all-lits-st-simps*
by *fast+*
define *g'* **where** ‹*g'* = *g*(*L* := (*g L*)[*j'* := (*x1*, *C*, *b'*)])›
have *g-g'*: ‹*g*(*L* := (*g L*)[*j'* := (*x1*, *C'*, *b'*)]) = *g'*(*L* := (*g' L*)[*j'* := (*x1*, *C'*, *b'*)])›
unfolding *g'-def* **by** *auto*

have *H2*: ‹*fst* ‘# *mset* ((*g'*(*L* := (*g' L*)[*j'* := (*x1*, *C'*, *b'*)])) *La*) = *fst* ‘# *mset* (*g' La*)› **for** *La*

unfolding *g′-def*
by (*auto simp flip*: *mset-map simp*: *map-update*)
have *H3*: ‹*fst* ‘# *mset* (*take i* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*) @ *drop j* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*)) = *fst* ‘# *mset* (*take i* (*g′ La*) @ *drop j* (*g′ La*))› **for** *La*
unfolding *g′-def*
by (*auto simp flip*: *mset-map drop-map simp*: *map-update*)
have [*simp*]:
‹*fst* ‘# *mset* ((*take i* (*g′ L*))[*j′* := (*x1*, *C′*, *b′*)]) = *fst* ‘# *mset* (*take i* (*g′ L*))›
‹*fst* ‘# *mset* ((*drop j* ((*g′ L*)[*j′* := (*x1*, *C′*, *b′*)]))) = *fst* ‘# *mset* (*drop j* (*g′ L*))›
‹¬*j′* < *j* ⟹ *fst* ‘# *mset* ((*drop j* (*g′ L*))[*j′* − *j* := (*x1*, *C′*, *b′*)]) = *fst* ‘# *mset* (*drop j* (*g′ L*))›
unfolding *g′-def*
apply (*auto simp flip*: *mset-map drop-map simp*: *map-update drop-update-swap*; *fail*)
apply (*auto simp flip*: *mset-map drop-map simp*: *map-update drop-update-swap*; *fail*)
apply (*auto simp flip*: *mset-map drop-map simp*: *map-update drop-update-swap*; *fail*)
done
have ‹*j′* < *length* (*g′ L*) ⟹ *j′* < *i* ⟹ (*x1*, *C*, *b′*) ∈ *set* ((*take i* (*g L*))[*j′* := (*x1*, *C*, *b′*)])›
using *nth-mem*[*of* ‹*j′*› ‹(*take i* (*g L*))[*j′* := (*x1*, *C*, *b′*)]›] **unfolding** *g′-def*
by *auto*
then have *H*: ‹*L* ∈# *?L* ⟹ *j′* < *length* (*g′ L*) ⟹ *j′* < *i* ⟹ *b′* ⟹ *x1* ∈# *dom-m b*›
using *C′ Heq*[*of L x1 C b′*]
by (*cases* ‹*j′* < *j*›) (*simp*, *auto*)
have ‹¬ *j′* < *j* ⟹ *j′* − *j* < *length* (*g′ L*) − *j* ⟹ (*x1*, *C*, *b′*) ∈ *set* (*drop j* ((*g L*)[*j′* := (*x1*, *C*, *b′*)]))›
using *nth-mem*[*of* ‹*j′*−*j*› ‹*drop j* ((*g L*)[*j′* := (*x1*, *C*, *b′*)])›] **unfolding** *g′-def*
by *auto*
then have *H′*: ‹*L* ∈# *?L* ⟹ ¬ *j′* < *j* ⟹ *j′* − *j* < *length* (*g′ L*) − *j* ⟹ *b′* ⟹ *x1* ∈# *dom-m b*›
using *C′ Heq*[*of L x1 C b′*] **unfolding** *g′-def*
by *auto*

have *dist*: ‹*La*∈# *?L* ⟹ *La* = *L* ⟹ *distinct-watched* (*take i* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*) @ *drop j* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*)›
for *La*
using *Hdisteq*[*of L*] **unfolding** *g-g′*[*symmetric*]
by (*cases* ‹*j′* < *j*›)
(*auto simp*: *map-update drop-update-swap*)
have *L-alt*: ‹*?L* = *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g′*)›
unfolding *g′-def* **by** *simp*
have ‹*La*∈# *?L* ⟹ *La* = *L* ⟹ *distinct-watched* (*take i* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*) @ *drop j* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*)) ∧ ((*i′*, *K*, *b″*)∈#*mset* (*take i* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*) @ *drop j* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*)) ⟶ *i′* ∈# *dom-m b* ⟶ *K* ∈ *set* (*b* ∝ *i′*) ∧ *K* ≠ *La* ∧ *correctly-marked-as-binary b* (*i′*, *K*, *b″*)) ∧ ((*i′*, *K*, *b″*)∈#*mset* (*take i* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*) @ *drop j* ((*g′*(*L* := (*g′ L*)[*j′* := (*x1*, *C′*, *b′*)])) *La*)) ⟶

```
      b'' ⟶ i' ∈# dom-m b) ∧
    {#i ∈# fst `# mset (take i ((g'(L := (g' L)[j' := (x1, C', b')])) La) @ drop j ((g'(L := (g' L)[j'
:= (x1, C', b')])) La)).
      i ∈# dom-m b#} =
    clause-to-update La (a, b, c, d, e, NEk, UEk, NS, US, N0, U0, {#}, {#})› for La i' K b''
  using C' Heq[of La i' K] Heq[of La i' K b'] H H' dist[of La] corr-watched unfolding g-g'
g'-def[symmetric]
  apply (auto elim!: in-set-upd-cases simp: drop-update-swap simp del: distinct-append)
  apply (cases ‹j' < j›; auto elim!: in-set-upd-cases simp: drop-update-swap simp del: distinct-append)+
  done
 moreover have ‹La∈#?L ⟹
    (La ≠ L ⟶
     distinct-watched ((g'(L := (g' L)[j' := (x1, C', b')])) La) ∧
     (∀(i, K, ba)∈#mset ((g'(L := (g' L)[j' := (x1, C', b')])) La).
        i ∈# dom-m b ⟶
        K ∈ set (b ∝ i) ∧
        K ≠ La ∧ correctly-marked-as-binary b (i, K, ba)) ∧
     (∀(i, K, ba)∈#mset ((g'(L := (g' L)[j' := (x1, C', b')])) La).
        ba ⟶ i ∈# dom-m b) ∧
     {#i ∈# fst `# mset ((g'(L := (g' L)[j' := (x1, C', b')])) La).
      i ∈# dom-m b#} =
     clause-to-update La (a, b, c, d, e, NEk, UEk, NS, US, N0, U0, {#}, {#}))›
   for La
  using Hneq Hdistneq
  unfolding correct-watching-except.simps g-g' g'-def[symmetric]
  by auto
 ultimately show ?thesis
  unfolding correct-watching-except.simps g-g' g'-def[symmetric]
  unfolding H2 H3 all-lits-st-simps L-alt
  by blast
qed


lemma correct-watching-except-correct-watching-except-Suc-notin:
 assumes
  ‹fst (watched-by S L ! w) ∉# dom-m (get-clauses-wl S)› and
  j-w: ‹j ≤ w› and
  w-le: ‹w < length (watched-by S L)› and
  corr: ‹correct-watching-except j w L S›
 shows ‹correct-watching-except j (Suc w) L (keep-watch L j w S)›
proof −
 obtain M N D NE UE NEk UEk NS US N0 U0 Q W where
  S: ‹S = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)› by (cases S)
 let ?L = ‹all-lits-st S›
 have [simp]: ‹fst (W L ! w) ∉# dom-m N›
  using assms unfolding S by auto
 have
  Hneq: ‹⋀La. La∈#?L ⟶
    (La ≠ L ⟶
 distinct-watched (W La) ∧
     ((∀(i, K, b)∈#mset (W La). i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧
        correctly-marked-as-binary N (i, K, b)) ∧
      (∀(i, K, b)∈#mset (W La). b ⟶ i ∈# dom-m N)) ∧
      {#i ∈# fst `# mset (W La). i ∈# dom-m N#} = clause-to-update La (M, N, D, NE, UE,
NEk, UEk, NS, US, N0, U0, {#}, {#}))› and
  Heq: ‹⋀La. La∈#?L ⟶
```

```
      (La = L ⟶
  distinct-watched (take j (W La) @ drop w (W La)) ∧
       ((∀(i, K, b)∈#mset (take j (W La) @ drop w (W La)). i ∈# dom-m N ⟶
           K ∈ set (N ∝ i) ∧ K ≠ La ∧ correctly-marked-as-binary N (i, K, b)) ∧
        (∀(i, K, b)∈#mset (take j (W La) @ drop w (W La)). b ⟶ i ∈# dom-m N) ∧
       {#i ∈# fst `# mset (take j (W La) @ drop w (W La)). i ∈# dom-m N#} =
       clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})))›
    using corr unfolding S correct-watching-except.simps
    by fast+

  have eq: ‹mset (take j ((W(L := (W L)[j := W L ! w])) La) @ drop (Suc w) ((W(L := (W L)[j :=
W L ! w])) La)) =
    remove1-mset (W L ! w) (mset (take j (W La) @ drop w (W La)))› if [simp]: ‹La = L› for La
    using w-le j-w
    by (auto simp: S take-Suc-conv-app-nth Cons-nth-drop-Suc[symmetric]
       list-update-append)

  have ‹case x of (i, K, b) ⇒ i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧
      correctly-marked-as-binary N (i, K, b)›
    if
      ‹La ∈# ?L› and
      ‹La = L› and
      ‹x ∈# mset (take j ((W(L := (W L)[j := W L ! w])) La) @
              drop (Suc w) ((W(L := (W L)[j := W L ! w])) La))›
    for La :: ‹'a literal› and x :: ‹nat × 'a literal × bool›
    using that Heq[of L] w-le j-w
    by (subst (asm) eq) (auto dest!: in-diffD)
  moreover have ‹case x of (i, K, b) ⇒ b ⟶ i ∈# dom-m N›
    if
      ‹La ∈# ?L› and
      ‹La = L› and
      ‹x ∈# mset (take j ((W(L := (W L)[j := W L ! w])) La) @
              drop (Suc w) ((W(L := (W L)[j := W L ! w])) La))›
    for La :: ‹'a literal› and x :: ‹nat × 'a literal × bool›
    using that Heq[of L] w-le j-w unfolding S
    by (subst (asm) eq) (fastforce simp: S dest!: in-diffD)+
  moreover have ‹{#i ∈# fst `#
            mset
             (take j ((W(L := (W L)[j := W L ! w])) La) @
              drop (Suc w) ((W(L := (W L)[j := W L ! w])) La)).
       i ∈# dom-m N#} =
      clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})›
    if
      ‹La ∈# ?L› and
      ‹La = L›
    for La :: ‹'a literal›
    using that Heq[of L] w-le j-w
    by (subst eq) (auto dest!: in-diffD simp: image-mset-remove1-mset-if)
  moreover have ‹case x of (i, K, b) ⇒ i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧
      correctly-marked-as-binary N (i, K, b)›
    if
      ‹La ∈# ?L› and
      ‹La ≠ L› and
      ‹x ∈# mset ((W(L := (W L)[j := W L ! w])) La)›
    for La :: ‹'a literal› and x :: ‹nat × 'a literal × bool›
    using that Hneq[of La]
```

```
    by simp
  moreover have ‹case x of (i, K, b) ⇒ b ⟶ i ∈# dom-m N›
    if
      ‹La ∈# ?L› and
      ‹La ≠ L› and
      ‹x ∈# mset ((W(L := (W L)[j := W L ! w])) La)›
    for La :: ‹'a literal› and x :: ‹nat × 'a literal × bool›
    using that Hneq[of La]
    by auto
  moreover have ‹{#i ∈# fst `# mset ((W(L := (W L)[j := W L ! w])) La). i ∈# dom-m N#} =
      clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})›
    if
      ‹La ∈# ?L› and
      ‹La ≠ L›
    for La :: ‹'a literal›
    using that Hneq[of La]
    by simp
  moreover have ‹distinct-watched ((W(L := (W L)[j := W L ! w])) La)›
    if
      ‹La ∈# ?L› and
      ‹La ≠ L›
    for La :: ‹'a literal›
    using that Hneq[of La]
    by simp
  moreover have ‹distinct-watched (take j ((W(L := (W L)[j := W L ! w])) La) @
              drop (Suc w) ((W(L := (W L)[j := W L ! w])) La))›
    if
      ‹La ∈# ?L› and
      ‹La = L›
    for La :: ‹'a literal›
    using that Heq[of L] w-le j-w apply −
    apply (subst distinct-mset-mset-distinct[symmetric])
    apply (subst mset-map)
    apply (subst eq)
    apply (solves simp)
    apply (subst (asm) distinct-mset-mset-distinct[symmetric])
    apply (subst (asm) mset-map)
    apply (rule distinct-mset-mono[of - ‹{#i. (i, j, k) ∈# mset (take j (W L) @ drop w (W L))#}›])
    by (auto simp: image-mset-remove1-mset-if split: if-splits)
  ultimately show ?thesis
    unfolding S keep-watch-def prod.simps correct-watching-except.simps
    by simp
qed

lemma correct-watching-except-correct-watching-except-update-clause:
  assumes
    corr: ‹correct-watching-except (Suc j) (Suc w) L
      (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(L := (W L)[j := W L ! w]))› and
    j-w: ‹j ≤ w› and
    w-le: ‹w < length (W L)› and
    L': ‹L' ∈# all-lits-st (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)›
      ‹L' ∈ set (N ∝ x1)›and
    L-L: ‹L ∈# all-lits-st (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)› and
    L: ‹L ≠ N ∝ x1 ! xa› and
    dom: ‹x1 ∈# dom-m N› and
    i-xa: ‹i < length (N ∝ x1)› ‹xa < length (N ∝ x1)› and
```

[*simp*]: ‹*W L ! w = (x1, x2, b)*› **and**
N-i: ‹*N ∝ x1 ! i = L*› ‹*N ∝ x1 ! (1 − i) ≠ L*›‹*N ∝ x1 ! xa ≠ L*› **and**
N-xa: ‹*N ∝ x1 ! xa ≠ N ∝ x1 ! i*› ‹*N ∝ x1 ! xa ≠ N ∝ x1 ! (Suc 0 − i)*›**and**
i-2: ‹*i < 2*› **and** ‹*xa ≥ 2*› **and**
L-neq: ‹*L′ ≠ N ∝ x1 ! xa*› — The new blocking literal is not the new watched literal.
shows ‹*correct-watching-except j (Suc w) L*
(M, N(x1 ↪ swap (N ∝ x1) i xa), D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W
(L := (W L)[j := (x1, x2, b)],
N ∝ x1 ! xa := W (N ∝ x1 ! xa) @ [(x1, L′, b)])›
proof −
define *W′* **where** ‹*W′ ≡ W(L := (W L)[j := W L ! w])*›
have ‹*length (N ∝ x1) > 2*›
using *i-2 i-xa assms*
by (*auto simp*: *correctly-marked-as-binary.simps*)
let *?L* = ‹*all-lits-st (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)*›
have *L*: ‹*?L = all-lits-st (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W′)*›
using *assms* **unfolding** *W′-def* **by** *simp*
have
Heq: ‹⋀*La i K b. La∈# ?L ⟹*
La = L ⟹
distinct-watched (take (Suc j) (W′ La) @ drop (Suc w) (W′ La)) ∧
((i, K, b)∈#mset (take (Suc j) (W′ La) @ drop (Suc w) (W′ La)) ⟶
i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧ correctly-marked-as-binary N (i, K, b)) ∧
((i, K, b)∈#mset (take (Suc j) (W′ La) @ drop (Suc w) (W′ La)) ⟶
b ⟶ i ∈# dom-m N) ∧
{#i ∈# fst ‘#
mset
(take (Suc j) (W′ La) @ drop (Suc w) (W′ La)).
i ∈# dom-m N#} =
clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})› **and**
Hneq: ‹⋀*La i K b. La∈#?L ⟹*
La ≠ L ⟹
distinct-watched (W′ La) ∧
((i, K, b)∈#mset (W′ La) ⟶ i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧
correctly-marked-as-binary N (i, K, b)) ∧
((i, K, b)∈#mset (W′ La) ⟶ b ⟶ i ∈# dom-m N) ∧
{#i ∈# fst ‘# mset (W′ La). i ∈# dom-m N#} =
clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})› **and**
Hneq2: ‹⋀*La. La∈#?L ⟹*
(La ≠ L ⟶
distinct-watched (W′ La) ∧
{#i ∈# fst ‘# mset (W′ La). i ∈# dom-m N#} =
clause-to-update La (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#}))›
using *corr* **unfolding** *correct-watching-except.simps W′-def*[*symmetric*] *L*
by *fast+*
have *H1*: ‹*mset ‘# ran-mf (N(x1 ↪ swap (N ∝ x1) i xa)) = mset ‘# ran-mf N*›
using *dom i-xa distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def dest*!: *multi-member-split intro*!: *image-mset-cong2*)
have *W-W′*: ‹*W*
(L := (W L)[j := (x1, x2, b)], N ∝ x1 ! xa := W (N ∝ x1 ! xa) @ [(x1, L′, b)]) =
W′(N ∝ x1 ! xa := W (N ∝ x1 ! xa) @ [(x1, L′, b)])›
unfolding *W′-def*
by *auto*
have *W-W2*: ‹*W (N ∝ x1 ! xa) = W′ (N ∝ x1 ! xa)*›
using *L* **unfolding** *W′-def* **by** *auto*
have *H2*: ‹*set (swap (N ∝ x1) i xa) = set (N ∝ x1)*›

using *i-xa* **by** *auto*
have [*simp*]:
‹*set* (*fst* (*the* (*if x1 = ia then Some* (*swap* (*N* ∝ *x1*) *i xa*, *irred N x1*) *else fmlookup N ia*))) =
set (*fst* (*the* (*fmlookup N ia*)))› **for** *ia*
using *H2*
by *auto*
have *H3*: ‹*i* = *x1* ∨ *i* ∈# *remove1-mset x1* (*dom-m N*) ⟷ *i* ∈# *dom-m N*› **for** *i*
using *dom* **by** (*auto dest*: *multi-member-split*)
have *set-N-swap-x1*: ‹*set* (*watched-l* (*swap* (*N* ∝ *x1*) *i xa*)) = {*N* ∝ *x1* ! (*1* −*i*), *N* ∝ *x1* ! *xa*}›
using *i-2 i-xa* ‹*xa* ≥ *2*› *N-i*
by (*cases* ‹*N* ∝ *x1*›; *cases* ‹*tl* (*N* ∝ *x1*)›; *cases i*; *cases* ‹*i*−*1*›; *cases xa*)
(*auto simp*: *swap-def split*: *nat.splits*)
have *set-N-x1*: ‹*set* (*watched-l* (*N* ∝ *x1*)) = {*N* ∝ *x1* ! (*1* −*i*), *N* ∝ *x1* ! *i*}›
using *i-2 i-xa* ‹*xa* ≥ *2*› *N-i*
by (*cases i*) (*auto simp*: *swap-def take-2-if*)

have *La-in-notin-swap*: ‹*La* ∈ *set* (*watched-l* (*N* ∝ *x1*)) ⟹
La ∉ *set* (*watched-l* (*swap* (*N* ∝ *x1*) *i xa*)) ⟹ *La* = *L*› **for** *La*
using *i-2 i-xa* ‹*xa* ≥ *2*› *N-i*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)

have *L-notin-swap*: ‹*L* ∉ *set* (*watched-l* (*swap* (*N* ∝ *x1*) *i xa*))›
using *i-2 i-xa* ‹*xa* ≥ *2*› *N-i*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)
have *N-xa-in-swap*: ‹*N* ∝ *x1* ! *xa* ∈ *set* (*watched-l* (*swap* (*N* ∝ *x1*) *i xa*))›
using *i-2 i-xa* ‹*xa* ≥ *2*› *N-i*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)
have *H4*: ‹(*i* = *x1* ⟶ *K* ∈ *set* (*N* ∝ *x1*) ∧ *K* ≠ *La*) ∧ (*i* ∈# *remove1-mset x1* (*dom-m N*) ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La*) ⟷
(*i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La*)› **for** *i P K La*
using *dom* **by** (*auto dest*: *multi-member-split*)
have [*simp*]: ‹*x1* ∉# *Ab* ⟹
{#*C* ∈# *Ab*.
(*x1* = *C* ⟶ *Q C*) ∧
(*x1* ≠ *C* ⟶ *R C*)#} =
{#*C* ∈# *Ab*. *R C*#}› **for** *Ab Q R*
by (*auto intro*: *filter-mset-cong*)
have *bin*:
‹*correctly-marked-as-binary N* (*x1*, *x2*, *b*)›
using *Heq*[*of L* ‹*fst* (*W L* ! *w*)› ‹*fst* (*snd* (*W L* ! *w*))› ‹*snd* (*snd* (*W L* ! *w*))›] *j-w w-le dom L*′
L-L **unfolding** *all-lits-def*
by (*auto simp*: *take-Suc-conv-app-nth W*′*-def list-update-append ac-simps*)
have *x1-new*: ‹*x1* ∉ *fst* ‘ *set* (*W* (*N* ∝ *x1* ! *xa*))›
proof (*rule ccontr*)
assume *H*: ¬ *?thesis*
have ‹*N* ∝ *x1* ! *xa* ∈# *?L*›
using *dom i-xa* **by** *auto*
then have ‹{#*i* ∈# *fst* ‘# *mset* (*W* (*N* ∝ *x1* ! *xa*)). *i* ∈# *dom-m N*#} =
clause-to-update (*N* ∝ *x1* ! *xa*) (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *Hneq*[*of* ‹*N* ∝ *x1* ! *xa*›] *L* **unfolding** *W*′*-def L*
by *simp*
then have ‹*x1* ∈# *clause-to-update* (*N* ∝ *x1* ! *xa*) (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *H dom* **by** (*metis* (*no-types*, *lifting*) *mem-Collect-eq set-image-mset*
set-mset-filter set-mset-mset)
then show *False*

using *N-xa i-2 i-xa*
by (*cases i*; *cases* ‹*N* ∝ *x1* ! *xa*›)
(*auto simp*: *clause-to-update-def take-2-if split*: *if-splits*)
qed

let *?N* = ‹*N*(*x1* ↪ *swap* (*N* ∝ *x1*) *i xa*)›
have ‹*L* ∈# *?L* ⟹ *La* = *L* ⟹
x ∈ *set* (*take j* (*W L*)) ∨ *x* ∈ *set* (*drop* (*Suc w*) (*W L*)) ⟹
case x of (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧
correctly-marked-as-binary ?N (*i*, *K*, *b*)› **for** *La x*
using *Heq*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le* **unfolding** *L*[*symmetric*]
by (*clarsimp simp*: *take-Suc-conv-app-nth W'-def list-update-append correctly-marked-as-binary.simps split*: *if-splits*)
moreover have ‹*L* ∈# *?L* ⟹ *La* = *L* ⟹
x ∈ *set* (*take j* (*W L*)) ∨ *x* ∈ *set* (*drop* (*Suc w*) (*W L*)) ⟹
case x of (*i*, *K*, *b*) ⇒*b* ⟶ *i* ∈# *dom-m N*› **for** *La x*
using *Heq*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le*
by (*auto simp*: *take-Suc-conv-app-nth W'-def list-update-append correctly-marked-as-binary.simps split*: *if-splits*)
moreover have ‹*L* ∈# *?L* ⟹
La = *L* ⟹
distinct-watched (*take j* (*W L*) @ *drop* (*Suc w*) (*W L*)) ∧
{#*i* ∈# *fst* ‘# *mset* (*take j* (*W L*)). *i* ∈# *dom-m N*#} + {#*i* ∈# *fst* ‘# *mset* (*drop* (*Suc w*) (*W L*)). *i* ∈# *dom-m N*#} =
clause-to-update L (*M*, *N*(*x1* ↪ *swap* (*N* ∝ *x1*) *i xa*), *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **for** *La*
using *Heq*[*of L x1 x2 b*] *j-w w-le dom L-notin-swap N-xa-in-swap distinct-mset-dom*[*of N*] *i-xa i-2 assms*(*12*)
by (*auto simp*: *take-Suc-conv-app-nth W'-def list-update-append set-N-x1 assms*(*11*) *clause-to-update-def dest*!: *multi-member-split split*: *if-splits intro*: *filter-mset-cong2*)

moreover have ‹*La* ∈# *?L* ⟹
La ≠ *L* ⟹
x ∈ *set* (*if La* = *N* ∝ *x1* ! *xa*
then W' (*N* ∝ *x1* ! *xa*) @ [(*x1*, *L'*, *b*)]
else (*W*(*L* := (*W L*)[*j* := (*x1*, *x2*, *b*)])) *La*) ⟹
case x of
(*i*, *K*, *b*) ⇒ *i* ∈# *dom-m ?N* ⟶ *K* ∈ *set* (*?N* ∝ *i*) ∧ *K* ≠ *La* ∧ *correctly-marked-as-binary ?N* (*i*, *K*, *b*)› **for** *La x*
using *Hneq*[*of La* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le L' L-neq bin dom* **unfolding** *L*
by (*fastforce simp*: *take-Suc-conv-app-nth W'-def list-update-append correctly-marked-as-binary.simps split*: *if-splits*)
moreover have ‹*La* ∈# *?L* ⟹
La ≠ *L* ⟹
x ∈ *set* (*if La* = *N* ∝ *x1* ! *xa*
then W' (*N* ∝ *x1* ! *xa*) @ [(*x1*, *L'*, *b*)]
else (*W*(*L* := (*W L*)[*j* := (*x1*, *x2*, *b*)])) *La*) ⟹
case x of (*i*, *K*, *b*) ⇒ *b* ⟶ *i* ∈# *dom-m N*› **for** *La x*
using *Hneq*[*of La* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le L' L-neq* ‹*length* (*N* ∝ *x1*) > *2*› *dom*
by (*auto simp*: *take-Suc-conv-app-nth W'-def list-update-append correctly-marked-as-binary.simps split*: *if-splits*)
moreover have ‹*La* ∈# *?L* ⟹
La ≠ *L* ⟹ *distinct-watched* ((*W*
(*L* := (*W L*)[*j* := (*x1*, *x2*, *b*)],

$N \propto x1 \,!\, xa := W\ (N \propto x1 \,!\, xa) \ @\ [(x1,\ L',\ b)])\ La\rangle$ **for** $La\ x$
using *Hneq*[*of La*] *j-w w-le L′ L-neq* ‹*length* $(N \propto x1) > 2$›
dom x1-new
by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append correctly-marked-as-binary.simps split*: *if-splits*)
moreover {
have ‹$N \propto x1 \,!\, xa \notin$ *set* (*watched-l* $(N \propto x1)$)›
using *N-xa*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)

then have ‹ $\bigwedge$*Ab Ac La*.
$?\mathcal{L} =$ *add-mset L′* (*add-mset* $(N \propto x1 \,!\, xa)$ *Ac*) $\Longrightarrow$
dom-m N = *add-mset x1 Ab* $\Longrightarrow$
$N \propto x1 \,!\, xa \neq L \Longrightarrow$
$\{\#i \in\# fst\ `\#\ mset\ (W\ (N \propto x1 \,!\, xa)).\ i = x1 \vee i \in\# Ab\#\} =$
$\{\#C \in\# Ab.\ N \propto x1 \,!\, xa \in$ *set* (*watched-l* $(N \propto C)\#\}$ ›
using *Hneq2*[*of* ‹$N \propto x1 \,!\, xa$›] *L-neq* **unfolding** *W-W′ W-W2* $\mathcal{L}$
by (*auto simp*: *clause-to-update-def split*: *if-splits*)
from *this*[*symmetric*] **have** ‹*La* $\in\#$ $?\mathcal{L} \Longrightarrow$
$La \neq L \Longrightarrow$
distinct-watched $(W'\ La) \wedge$
$(x1 \in\#$ *dom-m* $N \longrightarrow$
$(La = N \propto x1 \,!\, xa \longrightarrow$
add-mset x1 $\{\#i \in\# fst\ `\#\ mset\ (W'\ (N \propto x1 \,!\, xa)).\ i \in\#$ *dom-m* $N\#\} =$
clause-to-update $(N \propto x1 \,!\, xa)$ $(M,\ N(x1 \hookrightarrow swap\ (N \propto x1)\ i\ xa),\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ \{\#\})) \wedge$
$(La \neq N \propto x1 \,!\, xa \longrightarrow$
$\{\#i \in\# fst\ `\#\ mset\ (W\ La).\ i \in\#$ *dom-m* $N\#\} =$
clause-to-update La $(M,\ N(x1 \hookrightarrow swap\ (N \propto x1)\ i\ xa),\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ \{\#\}))) \wedge$
$(x1 \notin\#$ *dom-m* $N \longrightarrow$
$(La = N \propto x1 \,!\, xa \longrightarrow$
$\{\#i \in\# fst\ `\#\ mset\ (W'\ (N \propto x1 \,!\, xa)).\ i \in\#$ *dom-m* $N\#\} =$
clause-to-update $(N \propto x1 \,!\, xa)$ $(M,\ N(x1 \hookrightarrow swap\ (N \propto x1)\ i\ xa),\ D,\ NS,\ US,\ NEk,\ UEk,\ NE,\ UE,\ N0,\ U0,\ \{\#\},\ \{\#\})) \wedge$
$(La \neq N \propto x1 \,!\, xa \longrightarrow$
$\{\#i \in\# fst\ `\#\ mset\ (W\ La).\ i \in\#$ *dom-m* $N\#\} =$
clause-to-update La $(M,\ N(x1 \hookrightarrow swap\ (N \propto x1)\ i\ xa),\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ \{\#\}))))$› **for** *La*
using *Hneq2*[*of La*] *j-w w-le L′ dom distinct-mset-dom*[*of N*] *L-notin-swap N-xa-in-swap L-neq*
apply (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append clause-to-update-def*
add-mset-eq-add-mset set-N-x1 set-N-swap-x1 assms(11) *N-i all-lits-def ac-simps*
eq-commute[*of x1*]
eq-commute[*of* ‹$\{\#y \in\#$ -. $y \neq$ - $\longrightarrow N \propto x1 \,!\, xa \in$ *set* (*watched-l* $(N \propto y))\#\}$›]
dest!: *multi-member-split La-in-notin-swap*
split: *if-splits*
intro: *image-mset-cong2 intro*: *filter-mset-cong2*)
by (*smt* (*verit*, *ccfv-SIG*) *filter-mset-cong2*)
}
ultimately show *?thesis*
using *L j-w dom i-xa*
unfolding *correct-watching-except.simps H1 W′-def*[*symmetric*] *W-W′ H2 W-W2 H4 H3* $\mathcal{L}$[*symmetric*]
by (*intro conjI impI ballI*)
(*simp-all add*: *L′ W-W′ W-W2 H3 H4 drop-map*)
qed

definition *unit-propagation-inner-loop-wl-loop-pre* **where**
‹*unit-propagation-inner-loop-wl-loop-pre L* = (λ(*j*, *w*, *S*).
w < *length* (*watched-by S L*) ∧ *j* ≤ *w* ∧ *blits-in-*$\mathcal{L}_{in}$ *S* ∧
unit-propagation-inner-loop-wl-loop-inv L (*j*, *w*, *S*))›

definition *mop-watched-by* :: ‹'*v twl-st-wl* ⇒ '*v literal* ⇒ *nat* ⇒ - *nres*› **where**
‹*mop-watched-by S L w* = *do* {
ASSERT(*w* < *length* (*watched-by S L*));
RETURN ((*watched-by S L*) ! *w*)
}›

definition *mop-polarity-wl* :: ‹'*v twl-st-wl* ⇒ '*v literal* ⇒ *bool option nres*› **where**
‹*mop-polarity-wl S L* = *do* {
ASSERT(*L* ∈# *all-lits-st S*);
ASSERT(*no-dup* (*get-trail-wl S*));
RETURN(*polarity* (*get-trail-wl S*) *L*)
}›

lemma *mop-polarity-wl-mop-polarity-l*:
‹(*uncurry mop-polarity-wl*, *uncurry mop-polarity-l*) ∈ *state-wl-l b* ×$_r$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
unfolding *mop-polarity-l-def mop-polarity-wl-def uncurry-def*
by (*cases b*)
(*intro frefI nres-relI*; *refine-rcg*; *auto simp*: *state-wl-l-def all-lits-def all-lits-of-mm-union*
all-lits-st-def)+

definition *is-nondeleted-clause-pre* :: ‹*nat* ⇒ '*v literal* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*is-nondeleted-clause-pre C L S* ⟷ *C* ∈ *fst* ‘ *set* (*watched-by S L*) ∧ *L* ∈# *all-lits-st S*›
definition *other-watched-wl* :: ‹'*v twl-st-wl* ⇒ '*v literal* ⇒ *nat* ⇒ *nat* ⇒ '*v literal nres*› **where**
‹*other-watched-wl S L C i* = *do* {
ASSERT(*get-clauses-wl S* ∝ *C* ! *i* = *L* ∧ *i* < *length* (*get-clauses-wl S* ∝ *C*) ∧ *i* < 2 ∧
C ∈# *dom-m* (*get-clauses-wl S*) ∧ 1−*i* < *length* (*get-clauses-wl S* ∝ *C*));
mop-clauses-at (*get-clauses-wl S*) *C* (1 − *i*)
}›

lemma *other-watched-wl-other-watched-l*:
‹(*uncurry3 other-watched-wl*, *uncurry3 other-watched-l*) ∈ *state-wl-l b* ×$_f$ *Id* ×$_f$ *Id* ×$_f$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
unfolding *other-watched-wl-def other-watched-l-def uncurry-def*
by (*intro frefI nres-relI*)
(*refine-rcg*, *auto simp*: *state-wl-l-def*)

lemma *other-watched-wl-other-watched-l-spec-itself*:
‹((*S*, *L*, *C*, *i*), (*S'*, *L'*, *C'*, *i'*)) ∈ *state-wl-l b* ×$_r$ (*Id* :: ('*v literal* × -) *set*) ×$_r$ *nat-rel* ×$_r$
nat-rel ⟹
other-watched-wl S L C i ≤
⇓{(*L*, *L'*). *L* = *L'* ∧ *L* = *get-clauses-wl S* ∝ *C* ! (1 − *i*)}
(*other-watched-l S' L' C' i'*)›
using *twl-st-wl*(*2*)[*of S S' b*]
unfolding *other-watched-wl-def other-watched-l-def*
mop-clauses-at-def
by *refine-vcg clarsimp-all*

It was too hard to align the program unto a refinable form directly.

definition *unit-propagation-inner-loop-body-wl-int* :: ‹'*v literal* ⇒ *nat* ⇒ *nat* ⇒ '*v twl-st-wl* ⇒
(*nat* × *nat* × '*v twl-st-wl*) *nres*› **where**
‹*unit-propagation-inner-loop-body-wl-int L j w S* = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*));

```
    (C, K, b) ← mop-watched-by-at S L w;
    S ← mop-keep-watch L j w S;
    ASSERT(is-nondeleted-clause-pre C L S);
    val-K ← mop-polarity-wl S K;
    if val-K = Some True
    then RETURN (j+1, w+1, S)
    else do { — Now the costly operations:
      if C ∉# dom-m (get-clauses-wl S)
      then RETURN (j, w+1, S)
      else do {
        ASSERT(unit-prop-body-wl-inv S j w L);
        i ← pos-of-watched (get-clauses-wl S) C L;
        ASSERT(i ≤ 1);
        L' ← other-watched-wl S L C i;
        val-L' ← mop-polarity-wl S L';
        if val-L' = Some True
        then update-blit-wl L C b j w L' S
        else do {
          f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S) C;
          ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
          case f of
            None ⇒ do {
              if val-L' = Some False
              then do {S ← set-conflict-wl C S; RETURN (j+1, w+1, S)}
              else do {S ← propagate-lit-wl-general L' C i S; RETURN (j+1, w+1, S)}
            }
          | Some f ⇒ do {
              ASSERT(C ∈# dom-m (get-clauses-wl S) ∧ f < length (get-clauses-wl S ∝ C) ∧ f ≥ 2);
              K ← mop-clauses-at (get-clauses-wl S) C f;
              val-L' ← mop-polarity-wl S K;
              if val-L' = Some True
              then update-blit-wl L C b j w K S
              else update-clause-wl L L' C b j w i f S
            }
        }
      }
    }
  }›

definition propagate-proper-bin-case where
  ‹propagate-proper-bin-case L L' S C ⟷
    C ∈# dom-m (get-clauses-wl S) ∧ length ((get-clauses-wl S)∝C) = 2 ∧
    set (get-clauses-wl S∝C) = {L, L'} ∧ L ≠ L'›

definition unit-propagation-inner-loop-body-wl :: ‹'v literal ⇒ nat ⇒ nat ⇒ 'v twl-st-wl ⇒
    (nat × nat × 'v twl-st-wl) nres› where
  ‹unit-propagation-inner-loop-body-wl L j w S = do {
    ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
    (C, K, b) ← mop-watched-by-at S L w;
    S ← mop-keep-watch L j w S;
    ASSERT(is-nondeleted-clause-pre C L S);
    val-K ← mop-polarity-wl S K;
    if val-K = Some True
    then RETURN (j+1, w+1, S)
    else do {
      if b then do {
```

```
        ASSERT(propagate-proper-bin-case L K S C);
        if val-K = Some False
        then do {S ← set-conflict-wl C S;
          RETURN (j+1, w+1, S)}
        else do {
          S ← propagate-lit-wl-bin K C S;
          RETURN (j+1, w+1, S)}
      } — Now the costly operations:
      else if C ∉# dom-m (get-clauses-wl S)
      then RETURN (j, w+1, S)
      else do {
        ASSERT(unit-prop-body-wl-inv S j w L);
        i ← pos-of-watched (get-clauses-wl S) C L;
        ASSERT(i ≤ 1);
        L' ← other-watched-wl S L C i;
        val-L' ← mop-polarity-wl S L';
        if val-L' = Some True
        then update-blit-wl L C b j w L' S
        else do {
          f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S) C;
          ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
          case f of
            None ⇒ do {
              if val-L' = Some False
              then do {S ← set-conflict-wl C S;
                 RETURN (j+1, w+1, S)}
              else do {S ← propagate-lit-wl L' C i S; RETURN (j+1, w+1, S)}
            }
          | Some f ⇒ do {
              ASSERT(C ∈# dom-m (get-clauses-wl S) ∧ f < length (get-clauses-wl S ∝ C) ∧ f ≥ 2);
              K ← mop-clauses-at (get-clauses-wl S) C f;
              val-L' ← mop-polarity-wl S K;
              if val-L' = Some True
              then update-blit-wl L C b j w K S
              else update-clause-wl L L' C b j w i f S
            }
        }
      }
    }
  }›

lemma [twl-st-wl]: ‹get-clauses-wl (keep-watch L j w S) = get-clauses-wl S›
  by (cases S) (auto simp: keep-watch-def)


lemma unit-propagation-inner-loop-body-wl-int-alt-def:
 ‹unit-propagation-inner-loop-body-wl-int L j w S = do {
      ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
      (C, K, b) ← mop-watched-by-at S L w;
      let - = (C ∉# dom-m (get-clauses-wl S));
      S ← mop-keep-watch L j w S;
      ASSERT(is-nondeleted-clause-pre C L S);
      let b' = (C ∉# dom-m (get-clauses-wl S));
      if b' then do {
        let K = K;
        val-K ← mop-polarity-wl S K;
```

```
    if val-K = Some True
    then RETURN (j+1, w+1, S)
    else — Now the costly operations:
      RETURN (j, w+1, S)
  }
  else do {
    let S = S;
    K ← SPEC((=) K);
    val-K ← mop-polarity-wl S K;
    if val-K = Some True
    then RETURN (j+1, w+1, S)
    else do { — Now the costly operations:
      ASSERT(unit-prop-body-wl-inv S j w L);
      i ← pos-of-watched (get-clauses-wl S) C L;
      ASSERT(i ≤ 1);
      L' ← other-watched-wl S L C i;
      val-L' ← mop-polarity-wl S L';
      if val-L' = Some True
      then update-blit-wl L C b j w L' S
      else do {
        f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S) C;
        ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
        case f of
          None ⇒ do {
            if val-L' = Some False
            then do {S ← set-conflict-wl C S; RETURN (j+1, w+1, S)}
            else do {S ← propagate-lit-wl-general L' C i S; RETURN (j+1, w+1, S)}
          }
        | Some f ⇒ do {
            ASSERT(C ∈# dom-m (get-clauses-wl S) ∧ f < length (get-clauses-wl S ∝ C) ∧ f ≥ 2);
            K ← mop-clauses-at (get-clauses-wl S) C f;
            val-L' ← mop-polarity-wl S K;
            if val-L' = Some True
            then update-blit-wl L C b j w K S
            else update-clause-wl L L' C b j w i f S
          }
      }
    }
  }
}›
proof –
```

We first define an intermediate step where both then and else branches are the same.

```
  have E: ‹unit-propagation-inner-loop-body-wl-int L j w S = do {
    ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
    (C, K, b) ← mop-watched-by-at S L w;
    S'← mop-keep-watch L j w S;
    ASSERT(is-nondeleted-clause-pre C L S');
    let b' = C ∉# dom-m (get-clauses-wl S');
    if b' then do {
      let K = K;
      val-K ← mop-polarity-wl S' K;
      if val-K = Some True
      then RETURN (j+1, w+1, S')
      else do { — Now the costly operations:
        if b'
```

```
    then RETURN (j, w+1, S')
    else do {
      ASSERT(unit-prop-body-wl-inv S' j w L);
      i ← pos-of-watched (get-clauses-wl S') C L;
      ASSERT(i ≤ 1);
      L' ← other-watched-wl S' L C i;
      val-L' ← mop-polarity-wl S' L';
      if val-L' = Some True
      then update-blit-wl L C b j w L' S'
      else do {
        f ← find-unwatched-l (get-trail-wl S') (get-clauses-wl S') C;
        ASSERT (unit-prop-body-wl-find-unwatched-inv f C S');
        case f of
          None ⇒ do {
            if val-L' = Some False
            then do {S' ← set-conflict-wl C S'; RETURN (j+1, w+1, S')}
            else do {S' ← propagate-lit-wl-general L' C i S'; RETURN (j+1, w+1, S')}
          }
        | Some f ⇒ do {
          ASSERT(C ∈# dom-m (get-clauses-wl S') ∧ f < length (get-clauses-wl S' ∝ C) ∧ f ≥ 2);
          K ← mop-clauses-at (get-clauses-wl S') C f;
          val-L' ← mop-polarity-wl S' K;
          if val-L' = Some True
          then update-blit-wl L C b j w K S'
          else update-clause-wl L L' C b j w i f S'
          }
      }
    }
  }
}
else do {
  K ← SPEC((=) K);
  val-K ← mop-polarity-wl S' K;
  if val-K = Some True
  then RETURN (j+1, w+1, S')
  else do { — Now the costly operations:
    if b'
    then RETURN (j, w+1, S')
    else do {
      ASSERT(unit-prop-body-wl-inv S' j w L);
      i ← pos-of-watched (get-clauses-wl S') C L;
      ASSERT(i ≤ 1);
      L' ← other-watched-wl S' L C i;
      val-L' ← mop-polarity-wl S' L';
      if val-L' = Some True
      then update-blit-wl L C b j w L' S'
      else do {
        f ← find-unwatched-l (get-trail-wl S') (get-clauses-wl S') C;
        ASSERT (unit-prop-body-wl-find-unwatched-inv f C S');
        case f of
          None ⇒ do {
            if val-L' = Some False
            then do {S' ← set-conflict-wl C S'; RETURN (j+1, w+1, S')}
            else do {S'←propagate-lit-wl-general L' C i S';RETURN (j+1, w+1, S')}
          }
        | Some f ⇒ do {
```

```
            ASSERT(C ∈# dom-m (get-clauses-wl S') ∧ f < length (get-clauses-wl S' ∝ C) ∧ f ≥ 2);
             K ← mop-clauses-at (get-clauses-wl S') C f;
             val-L' ← mop-polarity-wl S' K;
             if val-L' = Some True
             then update-blit-wl L C b j w K S'
             else update-clause-wl L L' C b j w i f S'
             }
          }
        }
       }
     }
   }›
  (is ‹- = do {
     ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
     (C, K, b) ← mop-watched-by-at S L w;
     S'← mop-keep-watch L j w S;
     ASSERT(is-nondeleted-clause-pre C L S');
     let b' = (C ∉# dom-m (get-clauses-wl S'));
     if b' then
       ?P S' C K b b'
     else
       ?Q S' C K b b'
   }›)
   unfolding unit-propagation-inner-loop-body-wl-int-def if-not-swap bind-to-let-conv
     SPEC-eq-is-RETURN twl-st-wl
   unfolding Let-def if-not-swap bind-to-let-conv
     SPEC-eq-is-RETURN twl-st-wl
   apply (subst if-cancel)
   apply (intro bind-cong case-prod-cong if-cong[OF refl] refl if-cong)
   done
 show ?thesis
   unfolding E
   apply (subst if-replace-cond[of - ‹?P - - - -›])
   unfolding if-True if-False Let-def
   apply (intro bind-cong-nres case-prod-cong if-cong[OF refl] refl)
   done
qed
```

6.5 The Functions

6.5.1 Inner Loop

```
lemma clause-to-update-mapsto-upd-If:
 assumes
   i: ‹i ∈# dom-m N›
 shows
 ‹clause-to-update L (M, N(i ↪ C'), C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q) =
   (if L ∈ set (watched-l C')
    then add-mset i (remove1-mset i (clause-to-update L (M, N, C, NE, UE, NEk, UEk, NS, US, N0,
U0, WS, Q)))
     else remove1-mset i (clause-to-update L (M, N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS,
Q)))›
proof −
 define D' where ‹D' = dom-m N − {#i#}›
 then have [simp]: ‹dom-m N = add-mset i D'›
```

using *assms* **by** (*simp add*: *mset-set.remove*)
have [*simp*]: ‹$i \notin\# D'$›
using *assms distinct-mset-dom*[*of N*] **unfolding** *D'-def* **by** *auto*

have ‹{#$C \in\# D'$.
($i = C \longrightarrow L \in set\ (watched\text{-}l\ C')$) ∧
($i \neq C \longrightarrow L \in set\ (watched\text{-}l\ (N \propto C))$)#} =
{#$C \in\# D'$. $L \in set\ (watched\text{-}l\ (N \propto C))$#}›
by (*rule filter-mset-cong2*) *auto*
then show *?thesis*
unfolding *clause-to-update-def*
by *auto*
qed

lemma *keep-watch-st-wl*[*twl-st-wl*]:
‹*get-unit-clauses-wl* (*keep-watch L j w S*) = *get-unit-clauses-wl S*›
‹*get-init-clauses0-wl* (*keep-watch L j w S*) = *get-init-clauses0-wl S*›
‹*get-learned-clauses0-wl* (*keep-watch L j w S*) = *get-learned-clauses0-wl S*›
‹*get-clauses0-wl* (*keep-watch L j w S*) = *get-clauses0-wl S*›
‹*get-conflict-wl* (*keep-watch L j w S*) = *get-conflict-wl S*›
‹*get-trail-wl* (*keep-watch L j w S*) = *get-trail-wl S*›
by (*cases S*; *auto simp*: *keep-watch-def*; *fail*)+

declare *twl-st-wl*[*simp*]

lemma *correct-watching-except-correct-watching-except-propagate-lit-wl*:
assumes
corr: ‹*correct-watching-except j w L S*› **and**
i-le: ‹*Suc 0* < *length* (*get-clauses-wl S* ∝ *C*)› **and**
C: ‹*C* ∈# *dom-m* (*get-clauses-wl S*)› **and** *undef*: ‹*undefined-lit* (*get-trail-wl S*) *L'*› **and**
confl: ‹*get-conflict-wl S* = *None*› **and**
lit: ‹*L'* ∈# *all-lits-st S*› **and**
i: ‹*i* ≤ *1*›
shows ‹*propagate-lit-wl-general L' C i S* ≤ *SPEC*(λ*c*. *correct-watching-except j w L c*)›
proof −
obtain *M N D NE UE NEk UEk NS US N0 U0 Q W* **where** *S*: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **by** (*cases S*)
let *?L* = ‹*all-lits-st S*›
have
Hneq: ‹⋀*La*. *La*∈#*?L* ⟹
La ≠ *L* ⟹
(∀(*i*, *K*, *b*)∈#*mset* (*W La*). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)) ∧
(∀(*i*, *K*, *b*)∈#*mset* (*W La*). *b* ⟶ *i* ∈# *dom-m N*) ∧
{#*i* ∈# *fst* '# *mset* (*W La*). *i* ∈# *dom-m N*#} = *clause-to-update La* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **and**
Heq: ‹⋀*La*. *La*∈#*?L* ⟹
La = *L* ⟹
(∀(*i*, *K*, *b*)∈#*mset* (*take j* (*W La*) @ *drop w* (*W La*)). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)) ∧
(∀(*i*, *K*, *b*)∈#*mset* (*take j* (*W La*) @ *drop w* (*W La*)). *b* ⟶ *i* ∈# *dom-m N*) ∧
{#*i* ∈# *fst* '# *mset* (*take j* (*W La*) @ *drop w* (*W La*)). *i* ∈# *dom-m N*#} =
clause-to-update La (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
using *corr* **unfolding** *S correct-watching-except.simps*
by *fast*+

define N' **where** ‹$N' \equiv$ *if length* $(N \propto C) > 2$ *then* $N(C \hookrightarrow$ *swap* $(N \propto C)$ 0 $(Suc\ 0 - i))$ *else* N›

have ‹$Suc\ 0 - i <$ *length* $(N \propto C)$› **and** ‹$0 <$ *length* $(N \propto C)$›
using *i-le*
by (*auto simp*: *S*)
then have [*simp*]: ‹*mset* (*swap* $(N \propto C)$ 0 $(Suc\ 0 - i)) =$ *mset* $(N \propto C)$›
by (*auto simp*: *S*)
have *H1*[*simp*]: ‹{#*mset* (*fst x*). *x* ∈# *ran-m* N'#} =
{#*mset* (*fst x*). *x* ∈# *ran-m* *N*#}›
using *C*
by (*auto dest*!: *multi-member-split simp*: *ran-m-def S* N'*-def*
intro!: *image-mset-cong*)

have *H2*: ‹*mset* ‘# *ran-mf* $N' =$ *mset* ‘# *ran-mf* *N*›
using *H1* **by** *auto*
have *H3*: ‹*dom-m* $N' =$ *dom-m* *N*›
using *C* **by** (*auto simp*: *S* N'*-def*)
have *H4*: ‹*set* $(N' \propto ia) =$
set $(N \propto ia)$› **for** *ia*
using *i-le*
by (*cases* ‹$C = ia$›) (*auto simp*: *S* N'*-def*)
have *H5*: ‹*set* (*watched-l* $(N' \propto ia)) =$ *set* (*watched-l* $(N \propto ia))$› **for** *ia*
using *i-le*
by (*cases* ‹$C = ia$›; *cases i*; *cases* ‹$N \propto ia$›; *cases* ‹*tl* $(N \propto ia)$›) (*auto simp*: N'*-def S swap-def*)
have [*iff*]: ‹*correctly-marked-as-binary* $N\ C' \longleftrightarrow$ *correctly-marked-as-binary* $N'\ C'$› **for** C' *ia*
by (*cases* C')
(*auto simp*: *correctly-marked-as-binary.simps* N'*-def*)
have *H6*: ‹*propagate-lit-wl-general* $L'\ C\ i$ $(M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$
=
RETURN (*Propagated* $L'\ C \#\ M,\ N',\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,$ *add-mset* $(-\ L')$
$Q,\ W)$›
using *assms* ‹$Suc\ 0 - i <$ *length* $(N \propto C)$› *undef confl lit*
by (*auto simp*: *mop-clauses-swap-def S propagate-lit-wl-general-def* N'*-def*
cons-trail-propagate-l-def)
have $\mathcal{L}$: ‹*all-lits-st* $(M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
all-lits-st $(M,\ N',\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
using *assms i-le* **unfolding** N'*-def* **by** (*auto simp*: *S*)
show *?thesis*
using *corr lit* **unfolding** *S H6* **apply** −
apply (*rule RETURN-rule*)
unfolding *S prod.simps Let-def correct-watching-except.simps* $\mathcal{L}$
H1 H2 H3 H4 H5 clause-to-update-def get-clauses-l.simps H5 all-lits-st-simps
by *simp*
qed

lemma *unit-propagation-inner-loop-body-wl-int-alt-def2*:
‹*unit-propagation-inner-loop-body-wl-int* $L\ j\ w\ S =$ *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre* L $(j,\ w,\ S)$);
$(C,\ K,\ b) \leftarrow$ *mop-watched-by-at* $S\ L\ w$;
$S \leftarrow$ *mop-keep-watch* $L\ j\ w\ S$;
ASSERT(*is-nondeleted-clause-pre* $C\ L\ S$);
val-K ← *mop-polarity-wl* $S\ K$;
if val-K = *Some True*
then RETURN (j+1, w+1, S)
else do { — Now the costly operations:

```
if b then
  if C ∉# dom-m (get-clauses-wl S)
  then RETURN (j, w+1, S)
  else do {
    ASSERT(unit-prop-body-wl-inv S j w L);
    i ← pos-of-watched (get-clauses-wl S) C L;
    ASSERT(i ≤ 1);
    L' ← other-watched-wl S L C i;
    val-L' ← mop-polarity-wl S L';
    if val-L' = Some True
    then update-blit-wl L C b j w L' S
    else do {
      f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S) C;
      ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
      case f of
        None ⇒ do {
          if val-L' = Some False
          then do {S ← set-conflict-wl C S; RETURN (j+1, w+1, S)}
          else do {S ← propagate-lit-wl-general L' C i S; RETURN (j+1, w+1, S)}
        }
      | Some f ⇒ do {
          ASSERT(C ∈# dom-m (get-clauses-wl S) ∧ f < length (get-clauses-wl S ∝ C) ∧ f ≥ 2);
          K ← mop-clauses-at (get-clauses-wl S) C f;
          val-L' ← mop-polarity-wl S K;
          if val-L' = Some True
          then update-blit-wl L C b j w K S
          else update-clause-wl L L' C b j w i f S
        }
    }
  }
else
  if C ∉# dom-m (get-clauses-wl S)
  then RETURN (j, w+1, S)
  else do {
    ASSERT(unit-prop-body-wl-inv S j w L);
    i ← pos-of-watched (get-clauses-wl S) C L;
    ASSERT(i ≤ 1);
    L' ← other-watched-wl S L C i;
    val-L' ← mop-polarity-wl S L';
    if val-L' = Some True
    then update-blit-wl L C b j w L' S
    else do {
      f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S) C;
      ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
      case f of
        None ⇒ do {
          if val-L' = Some False
          then do {S ← set-conflict-wl C S; RETURN (j+1, w+1, S)}
          else do {S ← propagate-lit-wl-general L' C i S; RETURN (j+1, w+1, S)}
        }
      | Some f ⇒ do {
          ASSERT(C ∈# dom-m (get-clauses-wl S) ∧ f < length (get-clauses-wl S ∝ C) ∧ f ≥ 2);
          K ← mop-clauses-at (get-clauses-wl S) C f;
          val-L' ← mop-polarity-wl S K;
          if val-L' = Some True
          then update-blit-wl L C b j w K S
```

```
            else update-clause-wl L L' C b j w i f S
          }
        }
      }
    }
  }›
  unfolding unit-propagation-inner-loop-body-wl-int-def if-not-swap bind-to-let-conv
    SPEC-eq-is-RETURN twl-st-wl
  unfolding Let-def if-not-swap bind-to-let-conv
    SPEC-eq-is-RETURN twl-st-wl
  apply (subst if-cancel)
  apply (intro bind-cong-nres case-prod-cong if-cong[OF refl] refl)
  done

lemma unit-propagation-inner-loop-body-wl-alt-def:
  ‹unit-propagation-inner-loop-body-wl (L :: 'v literal) j w S = do {
      ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
      (C, K, b) ← mop-watched-by-at S L w;
      S ← mop-keep-watch L j w S;
      ASSERT(is-nondeleted-clause-pre C L S);
      val-K ← mop-polarity-wl S K;
      if val-K = Some True
      then RETURN (j+1, w+1, S)
      else do {
        if b then do {
          if False
          then RETURN (j, w+1, S)
          else do{
            let i::nat = (if ((get-clauses-wl S)∝C) ! 0 = L then 0 else 1);
                K = K; val-L' = val-K in
            if False ⟷ val-L' = Some True
            then RETURN (j, w+1, S)
            else do {
              f ← RETURN (None :: nat option);
              case f of
               None ⇒ do {
                 ASSERT(propagate-proper-bin-case L K S C);
                 if val-K = Some False
                 then do {S ← set-conflict-wl C S; RETURN (j+1, w+1, S)}
                 else do {
                   S ← propagate-lit-wl-bin' K C (if ((get-clauses-wl S)∝C) ! 0 = L then 0 else 1) S;
                   RETURN (j+1, w+1, S)}
               }
             | - ⇒ RETURN (j, w+1, S)
            }}
        } — Now the costly operations:
        else if C ∉# dom-m (get-clauses-wl S)
        then RETURN (j, w+1, S)
        else do {
          ASSERT(unit-prop-body-wl-inv S j w L);
          i ← pos-of-watched (get-clauses-wl S) C L;
          ASSERT(i ≤ 1);
          L' ← other-watched-wl S L C i;
          val-L' ← mop-polarity-wl S L';
          if val-L' = Some True
          then update-blit-wl L C b j w L' S
```

```
        else do {
          f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S) C;
          ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
          case f of
            None ⇒ do {
              if val-L' = Some False
              then do {S ← set-conflict-wl C S; RETURN (j+1, w+1, S)}
              else do {S ← propagate-lit-wl L' C i S; RETURN (j+1, w+1, S)}
            }
          | Some f ⇒ do {
              ASSERT(C ∈# dom-m (get-clauses-wl S) ∧ f < length (get-clauses-wl S ∝ C) ∧ f ≥ 2);
              K ← mop-clauses-at (get-clauses-wl S) C f;
              val-L'← mop-polarity-wl S K;
              if val-L' = Some True
              then update-blit-wl L C b j w K S
              else update-clause-wl L L' C b j w i f S
            }
        }
      }
    }
  }›
  unfolding unit-propagation-inner-loop-body-wl-def if-not-swap bind-to-let-conv
    SPEC-eq-is-RETURN twl-st-wl propagate-lit-wl-bin'-def
  unfolding Let-def if-not-swap bind-to-let-conv
    SPEC-eq-is-RETURN twl-st-wl if-False
  apply (intro bind-cong-nres case-prod-cong if-cong[OF refl] refl option.case-cong)
  apply auto
  done

lemma find-unwatched-l-itself:
  ‹(uncurry2 find-unwatched-l, uncurry2 find-unwatched-l) ∈ Id ×f Id ×f Id →f ⟨⟨Id⟩option-rel⟩nres-rel›
  by (auto intro!: frefI nres-relI)

lemma
  fixes S :: ‹'v twl-st-wl› and S' :: ‹'v twl-st-l› and L :: ‹'v literal› and w :: nat
  defines [simp]: ‹C' ≡ fst (watched-by S L ! w)›
  defines
    [simp]: ‹T ≡ remove-one-lit-from-wq C' S'›

  defines
    [simp]: ‹C'' ≡ get-clauses-l S' ∝ C'›
  assumes
    S-S': ‹(S, S') ∈ state-wl-l (Some (L, w))› and
    w-le: ‹w < length (watched-by S L)› and
    j-w: ‹j ≤ w› and
    corr-w: ‹correct-watching-except j w L S› and
    inner-loop-inv: ‹unit-propagation-inner-loop-wl-loop-inv L (j, w, S)› and
    n: ‹n = size (filter-mset (λ(i, -). i ∉# dom-m (get-clauses-wl S)) (mset (drop w (watched-by S L))))›
and
    confl-S: ‹get-conflict-wl S = None›
  shows unit-propagation-inner-loop-body-wl-wl-int: ‹unit-propagation-inner-loop-body-wl L j w S ≤
    ⇓ Id (unit-propagation-inner-loop-body-wl-int L j w S)›
proof −
  have n-d: ‹no-dup (get-trail-wl S)›
    using inner-loop-inv unfolding unit-propagation-inner-loop-wl-loop-inv-def prod.case
      unit-propagation-inner-loop-l-inv-def twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
```

cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def pcdcl-all-struct-invs-def **apply** −
by *normalize-goal+* (*simp only*: *twl-st-wl twl-st-l twl-st*)

have *pos-of-watched*: ‹((*N*, *C*, *L*), (*N'*, *C'*, *L'*)) ∈ *Id* ⟹ *pos-of-watched N C L* ≤ ⇓ *nat-rel* (*pos-of-watched N' C' L'*)›
for *N N' C C' L L'*
by *auto*
define *SLW* **where** ‹*SLW* ≡ *mop-watched-by-at S L w*›
define *R-SLW* **where** [*simp*]: ‹*R-SLW* = {((*C*, *K*, *b*), (*C'*, *K'*, *b'*)). *C* = *C'* ∧ *K* = *K'* ∧ *b* = *b'* ∧
(*b* ⟶ *length* (*get-clauses-wl S* ∝ *C*) = 2 ∧ *C* ∈# *dom-m* (*get-clauses-wl S*) ∧
K ∈ *set*(*get-clauses-wl S* ∝ *C*) ∧ *L* ∈ *set*(*get-clauses-wl S* ∝ *C*) ∧ *K* ≠ *L*) ∧
(*C* ∈# *dom-m* (*get-clauses-wl S*) ∧ ¬*b* ⟶ *length* (*get-clauses-wl S* ∝ *C*) > 2)}›
define *keep* **where** ‹*keep* ≡ *mop-keep-watch L j w S*›
have [*refine*]: ‹*unit-propagation-inner-loop-wl-loop-pre L*(*j*, *w*, *S*) ⟹
SLW ≤ ⇓ *R-SLW* (*mop-watched-by-at S L w*)›
using *corr-w*
unfolding *SLW-def mop-watched-by-at-def R-SLW-def*
apply (*refine-rcg*, *cases S*)
apply (*clarsimp dest*!: *multi-member-split simp*: *mop-watched-by-def ASSERT-refine-right*
correctly-marked-as-binary.simps
correct-watching-except-alt-def simp flip: *Cons-nth-drop-Suc*
dest!:)
apply *auto*
done
have [*refine*]: ‹*unit-propagation-inner-loop-wl-loop-pre L*(*j*, *w*, *S*) ⟹
keep ≤ ⇓ {(*T*, *T'*). (*T*, *T'*) ∈ *Id* ∧ *get-clauses-wl T* = *get-clauses-wl S* ∧ *get-trail-wl T* = *get-trail-wl S*}
(*mop-keep-watch L j w S*)› (**is** ‹- ⟹ - ≤ ⇓ *?keep* -›)
using *corr-w*
unfolding *keep-def mop-keep-watch-def*
by (*refine-rcg*, *cases S*)
(*clarsimp dest*!: *multi-member-split simp*: *mop-watched-by-def ASSERT-refine-right*
correct-watching-except.simps simp flip: *Cons-nth-drop-Suc*)

have [*refine*]: ‹*unit-propagation-inner-loop-wl-loop-pre L*(*j*, *w*, *S*) ⟹
(*x*, *x'*) ∈ *R-SLW* ⟹
(*T'*, *val-K*) ∈ *?keep* ⟹
¬ *x1* ∉# *dom-m* (*get-clauses-wl val-K*) ⟹
x'= (*x1*, *x2*) ⟹
x= (*x2b*, *x1c*) ⟹
RETURN (*if get-clauses-wl T'*∝ *x2b*! *0* = *L then 0 else 1*)
≤ ⇓ {(*K'*, *K*). *K* = *K'* ∧ *K* = (*if get-clauses-wl T'*∝ *x2b*! *0* = *L then 0 else 1*)}
(*pos-of-watched* (*get-clauses-wl val-K*) *x1 L*)›
(**is** ‹⟦-; -; -; -; -; -⟧ ⟹ - ≤ ⇓ *?pos* -›)
for *x x' x1 x2 x2a x1b x2b x1c x2c T T' val-Ka val-aK val-K*
unfolding *pos-of-watched-def R-SLW-def*
by *refine-rcg auto*

have *bin-mop-clauses-at*: ‹*unit-propagation-inner-loop-wl-loop-pre L*(*j*, *w*, *S*) ⟹
(*x*, *x'*) ∈ *R-SLW* ⟹
(*T'*, *val-K*) ∈ *?keep* ⟹
¬ *x1* ∉# *dom-m* (*get-clauses-wl val-K*) ⟹
x2= (*x2a*, *x1b*) ⟹
x'= (*x1*, *x2*) ⟹
x1c = (*x2d*, *x1d*) ⟹
x= (*x2b*, *x1c*) ⟹ *x1b* ⟹

(*if get-clauses-wl T′∝ x2b! 0 = L then 0 else 1*, *i*) ∈ *?pos x2b T′* ⟹
RETURN x2d
≤ ⇓ {(*K*, *K′*). *K* = *K′* ∧ *x2d* = *K′*}
(*other-watched-wl val-K L x1 i*)›
for *x x′ x1 x2 x2a x1b x2b x1c x2c T T′ val-Ka val-aK val-K x2d x1d i*
unfolding *mop-clauses-at-def R-SLW-def other-watched-wl-def*
by *refine-rcg* (*auto simp*: *length-list-2*)

have *bin-mop-polarity-wl*: ‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*) ⟹
unit-propagation-inner-loop-wl-loop-pre L (*j*, *w*, *S*) ⟹
(*Saa*, *val-K*) ∈ *?keep* ⟹
(*val-Ka*, *i*) ∈{(*v*, *v′*). *v* = *v′* ∧ *v* = *polarity* (*get-trail-wl Saa*) *x2c*} ⟹
¬ *x2* ∉# *dom-m* (*get-clauses-wl val-K*) ⟹
(*if get-clauses-wl Saa* ∝ *x2b* ! *0* = *L then 0 else 1*, *K*)
∈ *?pos x2b Saa* ⟹
(*x2c*, *K′*) ∈{(*K*, *K′*). *K* = *K′* ∧ *x2c* = *K′*} ⟹
(*x′*, *x1*) ∈ *R-SLW* ⟹
x1a =(*x2a*, *x1b*) ⟹
x1 =(*x2*, *x1a*) ⟹
x1c =(*x2c*, *Sa*) ⟹
x′ =(*x2b*, *x1c*) ⟹
val-Ka ≠ *Some True* ⟹
i ≠ *Some True* ⟹
Sa ⟹
x1b ⟹
RETURN val-Ka
≤ ⇓ (⟨*Id*⟩*option-rel*)
(*mop-polarity-wl val-K K′*)›
for *x′ x1 x2 x1a x2a x1b x2b x1c x2c Sa Saa val-K val-Ka i K K′*
unfolding *mop-polarity-wl-def*
by *refine-rcg auto*

have *bin-find-unwatched*: ‹*RETURN None*
≤ ⇓ ({(*b*,*b′*). (*b*, *b′*) ∈ ⟨*Id*⟩*option-rel* ∧ *b* = *None*})
(*find-unwatched-l* (*get-trail-wl Saa*) (*get-clauses-wl Saa*) *x1*)›
(**is** ‹- ≤ ⇓ *?bin-unw* -›)
if
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
‹(*x*, *x′*) ∈ *R-SLW*› **and**
‹*x2* =(*x1a*, *x2a*)› **and**
‹*x′* =(*x1*, *x2*)› **and**
‹*x2b* =(*x1c*, *x2c*)› **and**
‹*x* =(*x1b*, *x2b*)› **and**
‹(*Sa*, *Saa*) ∈ *?keep*› **and**
‹(*val-K*, *val-Ka*)
∈{(*v*, *v′*). *v* = *v′* ∧ *v* = *polarity* (*get-trail-wl Sa*) *x1c*}› **and**
‹*val-K* ≠ *Some True*› **and**
‹*val-Ka* ≠ *Some True*› **and**
‹*x2c*› **and**
‹*x2a*› **and**
‹¬ *False*› **and**
‹¬ *x1* ∉# *dom-m* (*get-clauses-wl Saa*)› **and**
‹*unit-prop-body-wl-inv Saa j w L*› **and**
‹(*if get-clauses-wl Sa* ∝ *x1b* ! *0* = *L then 0 else 1*, *i*)
∈{(*K′*, *K*).

$K = K' \wedge$
$K = (\textit{if get-clauses-wl Sa} \propto \textit{x1b} ! 0 = L \textit{ then 0 else 1})\}\rangle$ **and**
$\langle(\textit{x1c}, L') \in \{(K, K'). K = K' \wedge \textit{x1c} = K'\}\rangle$ **and**
$\langle(\textit{val-K}, \textit{val-L}') \in \langle \textit{bool-rel}\rangle \textit{option-rel}\rangle$ **and**
$\langle\neg \textit{False}\rangle$ **and**
$\langle \textit{val-L}' \neq \textit{Some True}\rangle$
for *x x' x1 x2 x1a x2a x1b x2b x1c x2c Sa Saa val-K val-Ka i L' val-L'*
using *that n-d*
unfolding *find-unwatched-l-def*
by (*auto simp*: *RETURN-RES-refine-iff length-list-2*)

have *bin-unw-Id*: $\langle x \in \textit{?bin-unw} \Longrightarrow x \in \langle \textit{Id}\rangle \textit{option-rel}\rangle$ **for** *x*
by *auto*
have *prop-bin-prop-gen*[*THEN fref-to-Down-curry3*, *refine*]:
$\langle(\textit{uncurry3 propagate-lit-wl-bin}', \textit{uncurry3 propagate-lit-wl-general}) \in$
$[\lambda(((L', C), -), S).\ \textit{length}\ (\textit{get-clauses-wl}\ S \propto C) = 2]_f\ (\textit{Id} \times_f \textit{nat-rel} \times_f \{\text{-}.\ \textit{True}\} \times_f \textit{Id}) \to \langle \textit{Id}\rangle \textit{nres-rel}\rangle$
by (*auto simp*: *propagate-lit-wl-bin'-def propagate-lit-wl-general-def cons-trail-propagate-l-def le-ASSERT-iff propagate-lit-wl-bin-def*
intro!: *frefI nres-relI*)

have *prop-prop-gen*[*THEN fref-to-Down-curry3*, *refine*]: $\langle(\textit{uncurry3 propagate-lit-wl}, \textit{uncurry3 propagate-lit-wl-general}) \in$
$[\lambda(((L', C), -), S).\ \textit{length}\ (\textit{get-clauses-wl}\ S \propto C) > 2]_f\ (\textit{Id} \times_f \textit{nat-rel} \times_f \textit{nat-rel} \times_f \textit{Id}) \to \langle \textit{Id}\rangle \textit{nres-rel}\rangle$
by (*auto simp*: *propagate-lit-wl-def propagate-lit-wl-general-def*
intro!: *frefI nres-relI*)

have [*THEN fref-to-Down-curry2*, *refine*]:
$\langle(\textit{uncurry2 mop-clauses-at}, \textit{uncurry2 mop-clauses-at}) \in \textit{Id} \times_f \textit{Id} \times_f \textit{Id} \to_f \langle \textit{Id}\rangle \textit{nres-rel}\rangle$
by (*auto intro*!: *frefI nres-relI*)
have [*refine*]:
$\langle((S, L), (S', L')) \in \textit{Id} \Longrightarrow \textit{mop-polarity-wl}\ S\ L \leq \Downarrow \{(v, v').\ v{=}v' \wedge v = \textit{polarity}\ (\textit{get-trail-wl}\ S)\ L\}$
$(\textit{mop-polarity-wl}\ S'\ L')\rangle$ **for** *S S' L L'*
by (*auto intro*!: *frefI nres-relI simp*: *mop-polarity-wl-def ASSERT-refine-right*)

have [*THEN fref-to-Down-curry*, *refine*]:
$\langle(\textit{uncurry set-conflict-wl}, \textit{uncurry set-conflict-wl}) \in \textit{Id} \times_f \textit{Id} \to_f \langle \textit{Id}\rangle \textit{nres-rel}\rangle$
by (*auto intro*!: *frefI nres-relI*)

have [*THEN fref-to-Down-curry3*, *refine*]:
$\langle(\textit{uncurry3 other-watched-wl}, \textit{uncurry3 other-watched-wl}) \in \textit{Id} \times_f \textit{Id} \times_f \textit{Id} \times_f \textit{Id} \to_f \langle \textit{Id}\rangle \textit{nres-rel}\rangle$
by (*auto intro*!: *frefI nres-relI*)

show *?thesis*
supply [[*goals-limit=1*]]
unfolding *unit-propagation-inner-loop-body-wl-int-alt-def2*
unit-propagation-inner-loop-body-wl-alt-def
apply (*subst SLW-def*[*symmetric*])
apply (*subst keep-def*[*symmetric*])
apply (*refine-rcg pos-of-watched*
find-unwatched-l-itself[*THEN fref-to-Down-curry2*])
subgoal for *SLw SLw' C x2a bin bL C' x2b bin' bL'*
by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
apply *assumption+*
apply (*rule bin-mop-clauses-at*; *assumption*)
apply (*rule bin-mop-polarity-wl*; *assumption*)
subgoal by *auto*
subgoal by *auto*
apply (*rule bin-find-unwatched*; *assumption*)
apply (*rule bin-unw-Id*; *assumption*)
subgoal
unfolding *propagate-proper-bin-case-def* **by** (*auto simp*: *length-list-2*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
apply *assumption*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

lemma *nres-add-unrelated*:
‹$P \leq \Downarrow \{(S, S'). R1\ S\}\ Q \Longrightarrow P \leq \Downarrow \{(S, S'). R2\ S\ S'\}\ Q \Longrightarrow P \leq \Downarrow\{(S, S'). R1\ S \wedge R2\ S\ S'\}\ Q$› **for** *P R1 R2 Q*
by (*simp add*: *pw-ref-iff*)

lemma *nres-add-unrelated2*:
‹$P \leq SPEC\ R1 \Longrightarrow P \leq \Downarrow \{(S, S'). R2\ S\ S'\}\ Q \Longrightarrow P \leq \Downarrow\{(S, S'). R1\ S \wedge R2\ S\ S'\}\ Q$› **for** *P R1 R2 Q*
using *nres-add-unrelated*[*of P R1 Q R2*]
by (*auto simp add*: *pw-ref-iff dest*: *inres-SPEC*)

lemma *nres-add-unrelated3*:
‹$P \leq \Downarrow \{(S, S'). R1\ S\}\ Q \Longrightarrow P \leq \Downarrow \{(S, S'). R2\ S\}\ Q \Longrightarrow P \leq \Downarrow\{(S, S'). R1\ S \wedge R2\ S\}\ Q$› **for** *P R1 R2 Q*
by (*simp add*: *pw-ref-iff*)

lemma *blits-in-$\mathcal{L}_{in}$-keep-watch2*: ‹$w < length\ (watched\text{-}by\ S\ L) \Longrightarrow blits\text{-}in\text{-}\mathcal{L}_{in}\ S \Longrightarrow blits\text{-}in\text{-}\mathcal{L}_{in}\ (keep\text{-}watch\ L\ j\ w\ S)$›
apply (*cases S*; *clarsimp simp*: *blits-in-$\mathcal{L}_{in}$-def keep-watch-def*)
using *nth-mem*[*of w* ‹(*watched-by S L*)›]
by (*auto elim*!: *in-set-upd-cases simp*: *eq-commute*[*of* - ‹- ! *w*›] *simp del*: *nth-mem*
dest!: *multi-member-split*)

lemma *unit-propagation-inner-loop-body-l-with-skip-alt-def*:
‹*unit-propagation-inner-loop-body-l-with-skip L* (*S'*, *n*) = *do* {
ASSERT (*clauses-to-update-l S'* $\neq \{\#\} \vee 0 <$ *n*);
ASSERT (*unit-propagation-inner-loop-l-inv L* (*S'*, *n*));
let - = ();
b ← *SPEC* (λ*b*. (*b* $\longrightarrow 0 <$ *n*) $\wedge$ ($\neg$ *b* $\longrightarrow$ *clauses-to-update-l S'* $\neq \{\#\}$));
let S' = *S'*;
let b = *b*;
if $\neg$ *b*
then do {
ASSERT (*clauses-to-update-l S'* $\neq \{\#\}$);
X2 ← *select-from-clauses-to-update S'*;
ASSERT (*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*));
x ← *SPEC* (λ*K*. *K* $\in$ *set* (*get-clauses-l* (*fst X2*) $\propto$ *snd X2*));
v ← *mop-polarity-l* (*fst X2*) *x*;
if v = *Some True then let T* = *fst X2 in RETURN* (*T*, *if get-conflict-l T* = *None then n else 0*)
else do {
v ← *pos-of-watched* (*get-clauses-l* (*fst X2*)) (*snd X2*) *L*;
va ← *other-watched-l* (*fst X2*) *L* (*snd X2*) *v*;
vaa ← *mop-polarity-l* (*fst X2*) *va*;
if vaa = *Some True*
then let T = *fst X2 in RETURN* (*T*, *if get-conflict-l T* = *None then n else 0*)
else do {
x ← *find-unwatched-l* (*get-trail-l* (*fst X2*)) (*get-clauses-l* (*fst X2*)) (*snd X2*);
case x of
None ⇒
if vaa = *Some False*
then do { *T* ← *set-conflict-l* (*snd X2*) (*fst X2*);
RETURN (*T*, *if get-conflict-l T* = *None then n else 0*) }
else do { *T* ← *propagate-lit-l va* (*snd X2*) *v* (*fst X2*);
RETURN (*T*, *if get-conflict-l T* = *None then n else 0*)}

```
          | Some a ⇒ do {
              x ← ASSERT (a < length (get-clauses-l (fst X2) ∝ snd X2));
              K ← mop-clauses-at (get-clauses-l (fst X2)) (snd X2) a;
              val-K ← mop-polarity-l (fst X2) K;
              if val-K = Some True
              then let T = fst X2 in RETURN (T, if get-conflict-l T = None then n else 0)
              else do {
                    T ← update-clause-l (snd X2) v a (fst X2);
                    RETURN (T, if get-conflict-l T = None then n else 0)
                  }
            }
          }
        }
       }
       else RETURN (S', n − 1)
    }›
proof −
  have remove-pairs: ‹do {(x2, x2') ← (b0 :: - nres); F x2 x2'} =
       do {X2 ← b0; F (fst X2) (snd X2)}› for T a0 b0 a b c f t F
    by (meson case-prod-unfold)

  have H1: ‹do {T ← do {x ← a ; b x}; RETURN (f T)} =
       do {x ← a; T ← b x; RETURN (f T)}› for T a0 b0 a b c f t
    by auto
  have H2: ‹do{T ← let v = val in g v; (f T :: - nres)} =
        do{let v = val; T ← g v; f T} › for g f T val
    by auto
  have H3: ‹do{T ← if b then g else g'; (f T :: - nres)} =
       (if b then do{T ← g; f T} else do{T ← g'; f T}) › for g g' f T b
    by auto
  have H4: ‹do{T ← case x of None ⇒ g | Some a ⇒ g' a; (f T :: - nres)} =
       (case x of None ⇒ do{T ← g; f T} | Some a ⇒ do{T ← g' a; f T}) › for g g' f T b x
    by (cases x) auto
  show ?thesis
    apply (subst Let-def)
    apply (subst Let-def)
    apply (subst Let-def)
    apply (subst Let-def)
    unfolding unit-propagation-inner-loop-body-l-with-skip-def prod.case
      unit-propagation-inner-loop-body-l-def remove-pairs
    unfolding H1 H2 H3 H4 bind-to-let-conv
    by (intro bind-cong[OF refl] if-cong refl option.case-cong) auto
qed

lemma get-subsumed-init-clauses-l-remove-one-lit-from-wq[simp]:
  ‹get-subsumed-init-clauses-l (remove-one-lit-from-wq a S) = get-subsumed-init-clauses-l S›
  ‹get-subsumed-clauses-wl (keep-watch L j w S') = get-subsumed-clauses-wl S'›
  ‹get-subsumed-clauses-wl (set-literals-to-update-wl x S') = get-subsumed-clauses-wl S'›
  by (cases S; cases S'; auto simp: keep-watch-def; fail)+

lemma nth-in-all-lits-stI[simp]:
  ‹x1 ∈# dom-m (get-clauses-wl S) ⟹ i < length (get-clauses-wl S ∝ x1) ⟹
  get-clauses-wl S ∝ x1 ! ( i) ∈# all-lits-st S›
  using in-clause-in-all-lits-of-m[of ‹get-clauses-wl S ∝ x1 ! i› ‹mset(get-clauses-wl S ∝ x1)›]
  by (auto simp: all-lits-def all-lits-st-def all-lits-of-mm-union ran-m-def
      all-lits-of-mm-add-mset dest!: multi-member-split)
```

lemma *all-lits-st-keep-watch*[*simp*]: ‹*all-lits-st* (*keep-watch L j w S*) = *all-lits-st S*›
by (*cases S*) (*auto simp*: *all-lits-st-def*)

lemma
fixes *S* :: ‹'*v twl-st-wl*› **and** *S'* :: ‹'*v twl-st-l*› **and** *L* :: ‹'*v literal*› **and** *w* :: *nat*
defines [*simp*]: ‹*C'* ≡ *fst* (*watched-by S L* ! *w*)›
defines
[*simp*]: ‹*T* ≡ *remove-one-lit-from-wq C' S'*›

defines
[*simp*]: ‹*C''* ≡ *get-clauses-l S'* ∝ *C'*›
assumes
S-S': ‹(*S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *w*))› **and**
w-le: ‹*w* < *length* (*watched-by S L*)› **and**
j-w: ‹*j* ≤ *w*› **and**
corr-w: ‹*correct-watching-except j w L S*› **and**
blit-in-lit: ‹*blits-in-*$\mathcal{L}_{in}$ *S*› **and**
inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)› **and**
n: ‹*n* = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl S*)) (*mset* (*drop w* (*watched-by S L*))))›
and
confl-S: ‹*get-conflict-wl S* = *None*›
shows *unit-propagation-inner-loop-body-wl-int-spec*: ‹*unit-propagation-inner-loop-body-wl-int L j w S*
≤
⇓{((*i*, *j*, *T'*), (*T*, *n*)).
(*T'*, *T*) ∈ *state-wl-l* (*Some* (*L*, *j*)) ∧
correct-watching-except i j L T' ∧
blits-in-$\mathcal{L}_{in}$ *T'* ∧
j ≤ *length* (*watched-by T' L*) ∧
length (*watched-by S L*) = *length* (*watched-by T' L*) ∧
i ≤ *j* ∧
(*get-conflict-wl T'* = *None* ⟶
n = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl T'*)) (*mset* (*drop j* (*watched-by T' L*)))) ∧
(*get-conflict-wl T'* ≠ *None* ⟶ *n* = 0)}
(*unit-propagation-inner-loop-body-l-with-skip L* (*S'*, *n*))› (**is** ‹- ≤ ⇓ *?unit* -›)**and**
unit-propagation-inner-loop-body-wl-update:
‹*unit-propagation-inner-loop-body-l-inv L C' T* ⟹
mset ‘# (*ran-mf* ((*get-clauses-wl S*) (*C'*↪ (*swap* (*get-clauses-wl S* ∝ *C'*) 0
(1 − (*if* (*get-clauses-wl S*)∝*C'* ! 0 = *L then* 0 *else* 1)))))) =
mset ‘# (*ran-mf* (*get-clauses-wl S*))› (**is** ‹- ⟹ *?eq*›)
proof −
obtain *bL* **where** *SLw*: ‹*watched-by S L* ! *w* = (*C'*, *bL*)›
using *C'-def* **by** (*cases* ‹*watched-by S L* ! *w*›) *auto*
let *?M* = ‹*get-trail-wl S*›

define *i* :: *nat* **where**
‹*i* ≡ (*if get-clauses-wl S* ∝ *C'* ! 0 = *L then* 0 *else* 1)›
have *n-d*: ‹*no-dup ?M*›
using *S-S' inner-loop-inv* **apply** −
unfolding *unit-propagation-inner-loop-wl-loop-inv-def*
unit-propagation-inner-loop-l-inv-def twl-struct-invs-def
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def prod.case*
cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def pcdcl-all-struct-invs-def state*$_W$*-of-def*
by *normalize-goal+*

```
  (simp only: twl-st-l twl-st-wl twl-st)
have
  alien-L':
    ‹L ∈# all-lits-st S›
    (is ?alien')
using inner-loop-inv twl-struct-invs-no-alien-in-trail[of - ‹−L›] unfolding unit-propagation-inner-loop-wl-loop-inv-def
  unit-propagation-inner-loop-l-inv-def prod.case apply − apply normalize-goal+
  by (drule twl-struct-invs-no-alien-in-trail[of - ‹−L›])
  (simp-all only: twl-st-wl twl-st-l twl-st multiset.map-comp comp-def clause-twl-clause-of
      ac-simps in-all-lits-uminus-iff)

have
  clause-ge-0: ‹0 < length (get-clauses-l T ∝ C')› (is ?ge) and
  L-def: ‹defined-lit (get-trail-wl S) L› ‹−L ∈ lits-of-l (get-trail-wl S)›
    ‹L ∉ lits-of-l (get-trail-wl S)› (is ?L-def) and
  i-le: ‹i < length (get-clauses-wl S ∝ C')› (is ?i-le) and
  i-le2: ‹1−i < length (get-clauses-wl S ∝ C')› (is ?i-le2) and
  C'-dom: ‹C' ∈# dom-m (get-clauses-l T)› (is ?C'-dom) and
  L-watched: ‹L ∈ set (watched-l (get-clauses-l T ∝ C'))› (is ?L-w) and
  dist-clss: ‹distinct-mset-mset (mset '# ran-mf (get-clauses-wl S))› and
  confl: ‹get-conflict-l T = None› (is ?confl) and
  alien-L:
    ‹L ∈# all-lits-st S›
    (is ?alien) and
  correctly-marked-as-binary: ‹correctly-marked-as-binary (get-clauses-wl S) (C', bL)›
if
  ‹unit-propagation-inner-loop-body-l-inv L C' T›
proof −
  have ‹unit-propagation-inner-loop-body-l-inv L C' T›
    using that unfolding unit-prop-body-wl-inv-def by fast+
  then obtain T' where
    T-T': ‹(set-clauses-to-update-l (clauses-to-update-l T + {#C'#}) T, T') ∈ twl-st-l (Some L)› and
    struct-invs: ‹twl-struct-invs T'› and
    ‹twl-stgy-invs T'› and
    C'-dom: ‹C' ∈# dom-m (get-clauses-l T)› and
    ‹0 < C'› and
    ge-0: ‹0 < length (get-clauses-l T ∝ C')› and
    ‹no-dup (get-trail-l T)› and
    i-le: ‹(if get-clauses-l T ∝ C' ! 0 = L then 0 else 1)
      < length (get-clauses-l T ∝ C')› and
    i-le2: ‹1 − (if get-clauses-l T ∝ C' ! 0 = L then 0 else 1)
      < length (get-clauses-l T ∝ C')› and
    L-watched: ‹L ∈ set (watched-l (get-clauses-l T ∝ C'))› and
    confl: ‹get-conflict-l T = None›
    unfolding unit-propagation-inner-loop-body-l-inv-def by blast
  show ?i-le and ?C'-dom and ?L-w and ?i-le2
    using S-S' i-le C'-dom L-watched i-le2 unfolding i-def by auto
  have
      alien: ‹cdcl_W-restart-mset.no-strange-atm (state_W-of T')› and
      dup: ‹no-duplicate-queued T'› and
      lev: ‹cdcl_W-restart-mset.cdcl_W-M-level-inv (state_W-of T')› and
      dist: ‹cdcl_W-restart-mset.distinct-cdcl_W-state (state_W-of T')›
    using struct-invs unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      pcdcl-all-struct-invs-def state_W-of-def
    by blast+
  have n-d: ‹no-dup (trail (state_W-of T'))›
```

```
    using lev unfolding cdcl_W-restart-mset.cdcl_W-M-level-inv-def by auto
  have 1: ‹C ∈# clauses-to-update T' ⟹
      add-mset (fst C) (literals-to-update T') ⊆#
      uminus `# lit-of `# mset (get-trail T')› for C
    using dup unfolding no-duplicate-queued-alt-def
    by blast
  have H: ‹(L, twl-clause-of C'') ∈# clauses-to-update T'›
    using twl-st-l(5)[OF T-T'] 
    by (auto simp: twl-st-l)
  have uL-M: ‹-L ∈ lits-of-l (get-trail T')›
    using mset-le-add-mset-decr-left2[OF 1[OF H]]
    by (auto simp: lits-of-def)
  then show ‹defined-lit (get-trail-wl S) L› ‹-L ∈ lits-of-l (get-trail-wl S)›
    ‹L ∉ lits-of-l (get-trail-wl S)›
    using S-S' T-T' n-d by (auto simp: Decided-Propagated-in-iff-in-lits-of-l twl-st
      dest: no-dup-consistentD)
  show L: ?alien
    using alien uL-M T-T' struct-invs twl-st-l(25)[OF T-T'] S-S'
      twl-struct-invs-no-alien-in-trail[of T' ‹-L›]
      unit-init-clauses-get-unit-init-clauses-l[OF T-T']
    unfolding cdcl_W-restart-mset.no-strange-atm-def
    by (auto simp: in-all-lits-of-mm-ain-atms-of-iff)
  then have l-wl-inv: ‹(S, S') ∈ state-wl-l (Some (L, w)) ∧
      unit-propagation-inner-loop-body-l-inv L (fst (watched-by S L ! w))
       (remove-one-lit-from-wq (fst (watched-by S L ! w)) S') ∧
      L ∈# all-lits-st S ∧
      correct-watching-except j w L S ∧
      w < length (watched-by S L) ∧ get-conflict-wl S = None›
    using that assms L unfolding unit-prop-body-wl-inv-def unit-propagation-inner-loop-body-l-inv-def
    by (auto simp: twl-st)


  show ?ge
    by (rule ge-0)
  show ‹distinct-mset-mset (mset `# ran-mf (get-clauses-wl S))›
  using dist S-S' twl-st-l(1-8)[OF T-T'] T-T' unfolding cdcl_W-restart-mset.distinct-cdcl_W-state-alt-def
    by (auto simp: twl-st)
  show ?confl
    using confl .
  have ‹watched-by S L ! w ∈ set (take j (watched-by S L)) ∪ set (drop w (watched-by S L))›
    using L alien-L' C'-dom SLw w-le
    by (cases S)
      (auto simp: in-set-drop-conv-nth)
  then show ‹correctly-marked-as-binary (get-clauses-wl S) (C', bL)›
    using corr-w alien-L' C'-dom SLw S-S'
    by (cases S; cases ‹watched-by S L ! w›)
     (clarsimp simp: correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def
        simp del: Un-iff
         simp flip: all-lits-alt-def2 all-lits-def
       dest!: multi-member-split[of L])
qed

have i-def': ‹i = (if get-clauses-l T ∝ C' ! 0 = L then 0 else 1)›
  using S-S' unfolding i-def by auto

have [simp]: ‹length (watched-by (keep-watch L j w S) L) = length (watched-by S L)› for S j w L
```

by (*cases S*) (*auto simp*: *keep-watch-def*)
have *keep-watch-state-wl*: ‹*fst* (*watched-by S L* ! *w*) ∉# *dom-m* (*get-clauses-wl S*) ⟹
(*keep-watch L j w S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *Suc w*))›
using *S-S' w-le j-w* **by** (*cases S*; *cases S'*)
(*auto simp*: *state-wl-l-def keep-watch-def Cons-nth-drop-Suc*[*symmetric*]
drop-map)
have [*simp*]: ‹*drop* (*Suc w*) (*watched-by* (*keep-watch L j w S*) *L*) = *drop* (*Suc w*) (*watched-by S L*)›
using *j-w w-le* **by** (*cases S*) (*auto simp*: *keep-watch-def*)
have [*simp*]: ‹*get-clauses-wl* (*keep-watch L j w S*) = *get-clauses-wl S*› **for** *L j w S*
by (*cases S*) (*auto simp*: *keep-watch-def*)

have *trail-keep-w*: ‹*get-trail-wl* (*keep-watch L j w S*) = *get-trail-wl S*› **for** *L j w S*
by (*cases S*) (*auto simp*: *keep-watch-def*)

let *?pos-of-watched* = ‹{(*j*, *j'*). *j* = *j'* ∧ *j* < *2* ∧ *j* = *i*}›

have *pos-of-watched*:
‹((*N*, *C*, *K*), (*N'*, *C'*, *K'*)) ∈ *Id* ×*r* *Id* ×*r* *Id* ∧ *N* = *get-clauses-wl S* ∧ *C* = *fst* (*watched-by S L* ! *w*) ∧ *K* = *L* ⟹
pos-of-watched N C K ≤ ⇓ *?pos-of-watched* (*pos-of-watched N' C' K'*)›
for *N N' C C' K K'*
unfolding *pos-of-watched-def* **by** *refine-rcg* (*auto simp*: *i-def*)
have [*refine*]: ‹*mop-watched-by-at S L w* ≤ ⇓ {(*Slw*, -). *Slw* = *watched-by S L* ! *w* ∧
fst (*snd* (*Slw*)) ∈# *all-lits-st S* ∧
(*fst Slw* ∈# *dom-m* (*get-clauses-wl S*) ⟶
fst (*snd Slw*) ∈ *set* (*get-clauses-l S'* ∝ *fst* (*watched-by S L* ! *w*)))} (*RETURN* ())› (**is** ‹- ≤ ⇓ *?Slw* -›)
if
‹*clauses-to-update-l S'* ≠ {#} ∨ *0* < *n*› **and**
‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)›
using *assms that*
unfolding *mop-watched-by-at-def unit-propagation-inner-loop-l-inv-def*
prod.case **apply** −
apply *normalize-goal+*
apply (*cases* ‹*watched-by S L* ! *w*›)
subgoal for *T*
using *nth-mem*[*OF w-le*] *twl-struct-invs-no-alien-in-trail*[*of T* ‹−*L*›] **apply** −
by *refine-vcg*
(*auto simp*: *in-all-lits-of-mm-uminus-iff*
mset-take-mset-drop-mset' correct-watching-except-alt-def2
blits-in-$\mathcal{L}_{in}$*-def in-all-lits-minus-iff*
dest!: *multi-member-split*[*of L* ‹*all-lits-st S*›]
simp flip: *Cons-nth-drop-Suc all-lits-def all-lits-alt-def2*)
done
have [*refine*]:
‹*mop-keep-watch L j w S* ≤ ⇓ {(*T*, *T'*). *T* = *keep-watch L j w S* ∧ *T'* = *S'*} (*RETURN S'*)›
using *j-w w-le alien-L'* **unfolding** *mop-keep-watch-def all-lits-def*
by *refine-rcg* (*auto simp*: *ac-simps*)

have *keep-watch*: ‹*RETURN Sa*
≤ ⇓ {(*T*, (*T'*, *C*)). (*T*, *T'*) ∈ *state-wl-l* (*Some* (*L*, *Suc w*)) ∧
C = *C'* ∧ *T'* = *set-clauses-to-update-l* (*clauses-to-update-l S'* − {#*C*#}) *S'*}
(*select-from-clauses-to-update S'*)›
if

‹clauses-to-update-l $S' \neq \{\#\} \vee 0 < n$› **and**
‹unit-propagation-inner-loop-l-inv L (S', n)› **and**
‹unit-propagation-inner-loop-wl-loop-pre L (j, w, S)› **and**
‹$(x', ()) \in$?Slw› **and**
‹$x' = (x1, x2)$› **and**
‹$(x1 \notin\# \text{dom-m } (\text{get-clauses-wl } S), b) \in$ bool-rel› **and**
‹$(Sa, S') \in \{(T, T'). T = \text{keep-watch } L\ j\ w\ S \wedge T' = S'\}$› **and**
‹$\neg\ x1 \notin\# \text{dom-m } (\text{get-clauses-wl } Sa)$›
for x' x1 x2 x1a x2a b Sa
proof −
have ‹get-conflict-l $S' = None$›
using *that* **unfolding** unit-propagation-inner-loop-l-inv-def twl-struct-invs-def prod.case
apply −
apply normalize-goal+
by *auto*
then show ?thesis
using S-S′ *that* w-le j-w
unfolding select-from-clauses-to-update-def keep-watch-def
by (*cases S*)
(*auto intro*!: RETURN-RES-refine *simp*: state-wl-l-def drop-map
Cons-nth-drop-Suc[*symmetric*] eq-commute[*of* - ‹- ! w›])
qed

have f: ‹$((M, N, C), (M', N', C')) \in Id \Longrightarrow$ find-unwatched-l $M\ N\ C$
$\leq \Downarrow \{(found, found').\ found = found' \wedge$
$(found = None \longleftrightarrow (\forall L \in set\ (unwatched\text{-}l\ (N \propto C)).\ -L \in lits\text{-}of\text{-}l\ M)) \wedge$
$(\forall j.\ found = Some\ j \longrightarrow (j < length\ (N \propto C) \wedge (undefined\text{-}lit\ M\ ((N \propto C)!j) \vee (N \propto C)!j \in lits\text{-}of\text{-}l\ M) \wedge j \geq 2)) \wedge distinct\ (N \propto C)\}$
(*find-unwatched-l* $M'\ N'\ C'$)›
(**is** ‹- $\Longrightarrow$ - $\leq \Downarrow$?find -›) **for** $M\ M'\ N\ N'\ C\ C'$
using S-S′ **unfolding** find-unwatched-l-def
by *refine-rcg* (*auto intro*!: RES-refine)

have update-blit-wl: ‹update-blit-wl L x1 x2a j w L' Sa
$\leq \Downarrow$?unit
(*RETURN*(*fst X2*, *if* get-conflict-l (*fst X2*) = None *then* n *else* 0))›
if
cond: ‹clauses-to-update-l $S' \neq \{\#\} \vee 0 < n$› **and**
loop-inv: ‹unit-propagation-inner-loop-l-inv L (S', n)› **and**
‹unit-propagation-inner-loop-wl-loop-pre L (j, w, S)› **and**
‹inres (mop-watched-by-at S L w) x'› **and**
x': ‹$(x', ()) \in$?Slw›
‹$x2 = (x1a, x2a)$›
‹$x' = (x1, x2)$› **and**
‹$(x1 \notin\# \text{dom-m } (\text{get-clauses-wl } S), b) \in$ bool-rel› **and**
‹inres (mop-keep-watch L j w S) Sa› **and**
Sa: ‹$(Sa, S') \in \{(T, T'). T = \text{keep-watch } L\ j\ w\ S \wedge T' = S'\}$› **and**
dom: ‹$\neg\ x1 \notin\# \text{dom-m } (\text{get-clauses-wl } Sa)$› **and**
‹$\neg\ b$› **and**
‹clauses-to-update-l $S' \neq \{\#\}$› **and**
X2: ‹$(Sa, X2) \in \{(T, T', C).$
$(T, T') \in state\text{-}wl\text{-}l\ (Some(L, Suc\ w)) \wedge$
$C = C' \wedge$

```
      T' =
      set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S'))
       S'}› and
    l-inv: ‹unit-propagation-inner-loop-body-l-inv L (snd X2) (fst X2)› and
    ‹(K, x) ∈ Id› and
    ‹K ∈ Collect ((=) x1a)› and
    ‹x ∈{K. K ∈ set (get-clauses-l (fst X2) ∝ snd X2)}› and
    ‹(val-K, v) ∈ Id› and
    ‹val-K ≠ Some True› and
    ‹v ≠ Some True› and
    wl-inv: ‹unit-prop-body-wl-inv Sa j w L› and
    i: ‹(ia, va) ∈{(j, j'). j = j' ∧ j < 2 ∧ j = i}› and
    L': ‹(L', vaa)
     ∈{(L, L'). L = L' ∧ L = get-clauses-wl Sa ∝ x1 ! (1 − ia)}› and
    ‹val-L' = Some True› and
    ‹(val-L', vaaa) ∈ Id› and
    ‹vaaa = Some True›
  for x' x1 x2 x1a x2a b Sa X2 K x val-K v ia va L' vaa val-L' vaaa
proof −
  have confl: ‹get-conflict-wl S = None› and [simp]: ‹Sa = keep-watch L j w S›
    using S-S' loop-inv cond Sa unfolding unit-propagation-inner-loop-l-inv-def prod.case apply −
    by normalize-goal+ auto

  then obtain M N NE UE NEk UEk NS US N0 U0 Q W where
    S: ‹S = (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)›
    by (cases S) (auto simp: twl-st-l)
  have dom': ‹x1 ∈# dom-m (get-clauses-wl (keep-watch L j w S)) ⟷ True›
    using dom by auto
  then have SLW-dom': ‹fst (watched-by Sa L ! w)
     ∈# dom-m (get-clauses-wl Sa)› and
    SLw': ‹watched-by S L ! w = (x1, x1a, x2a)›
    using w-le x' by (auto simp: eq-commute[of - ‹- ! w›])
  have bin: ‹correctly-marked-as-binary N (x1, N ∝ x1 ! (Suc 0 − i), x2a)›
    using X2 correctly-marked-as-binary l-inv x' dom' SLw
    by (cases ‹bL›; cases ‹W L ! w›)
     (auto simp: S remove-one-lit-from-wq-def correctly-marked-as-binary.simps)

  obtain x where
    S-x: ‹(Sa, x) ∈ state-wl-l (Some (L, Suc w))› and
    unit-loop-inv:
      ‹unit-propagation-inner-loop-body-l-inv L (fst (watched-by Sa L ! w))
    x› and
    L: ‹L ∈# all-lits-st Sa› and
    corr-Sa: ‹correct-watching-except (Suc j) (Suc w) L Sa› and
    ‹w < length (watched-by Sa L)› and
    ‹get-conflict-wl Sa = None›
    using wl-inv SLW-dom' unfolding unit-prop-body-wl-inv-alt-def
    by blast
  obtain x' where
    x-x': ‹(set-clauses-to-update-l
     (clauses-to-update-l
       x +
       {#fst (watched-by Sa L ! w)#})
     x,
     x') ∈ twl-st-l (Some L)› and
    ‹twl-struct-invs x'› and
```

ge0: ‹(*if get-clauses-l x* ∝ *fst* (*watched-by Sa L* ! *w*) ! *0* = *L*
then 0 else 1) < *length* (*get-clauses-l x* ∝ *fst* (*watched-by Sa L* ! *w*))› **and**
ge1i: ‹*1* −
(*if get-clauses-l x* ∝ *fst* (*watched-by Sa L* ! *w*) ! *0* = *L then 0 else 1*)
< *length*
(*get-clauses-l x* ∝
fst (*watched-by Sa L* ! *w*))› **and**
L-watched: ‹*L* ∈ *set* (*watched-l*
(*get-clauses-l x* ∝
fst (*watched-by Sa L* ! *w*)))› **and**
‹*get-conflict-l x* = *None*›
using *unit-loop-inv*
unfolding *unit-propagation-inner-loop-body-l-inv-def*
by *blast*

have *unit-T*: ‹*unit-propagation-inner-loop-body-l-inv L C′ T*›
using *that*
by (*auto simp*: *remove-one-lit-from-wq-def*)

have ‹*twl-st-inv x′*›
using ‹*twl-struct-invs x′*› **unfolding** *twl-struct-invs-def* **by** *fast*
then have ‹∃ *x*. *twl-st-inv*
(*x*,{#*TWL-Clause* (*mset* (*watched-l* (*fst x*)))
(*mset* (*unwatched-l* (*fst x*)))
. *x* ∈# *init-clss-l N*#},
{#*TWL-Clause* (*mset* (*watched-l* (*fst x*)))
(*mset* (*unwatched-l* (*fst x*)))
. *x* ∈# *learned-clss-l N*#},
None, *NE*+*NEk*, *UE*+*UEk*, *NS*, *US*, *N0*, *U0*,
add-mset
(*L*, *TWL-Clause*
(*mset* (*watched-l* (*N* ∝ *fst* ((*W L*)[*j* := *W L* ! *w*] ! *w*))))
(*mset* (*unwatched-l* (*N* ∝ *fst* ((*W L*)[*j* := *W L* ! *w*] ! *w*)))))
{#(*L*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *x*)))
(*mset* (*unwatched-l* (*N* ∝ *x*))))
. *x* ∈#{#*i* ∈# *mset*
(*drop* (*Suc w*) (*map fst* ((*W L*)[*j* := *W L* ! *w*]))).
i ∈# *dom-m N*#}#},
Q)›
using *x-x′ S-x*
apply (*cases x*)
apply (*auto simp*: *S twl-st-l-def state-wl-l-def keep-watch-def*
simp del: *struct-wf-twl-cls.simps*)
done
then have ‹*Multiset.Ball*
({#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* ∈# *ran-m N*#})
struct-wf-twl-cls›
unfolding *twl-st-inv.simps image-mset-union*[*symmetric*] *all-clss-l-ran-m*
by *blast*
then have *distinct-N-x1*: ‹*distinct* (*N* ∝ *x1*)›
using *dom*
by (*auto simp*: *S ran-m-def mset-take-mset-drop-mset′ dest*!: *multi-member-split*)

have *watch-by-S-w*: ‹*watched-by* (*keep-watch L j w S*) *L* ! *w* = (*x1*, *x1a*, *x2a*)›
using *j-w w-le SLw x′ Sa* **unfolding** *i-def*

by (*cases S*)
(*auto simp*: *keep-watch-def split*: *if-splits*)
have *i-le*: ‹*i* < *length* (*N* ∝ *x1*)› ‹*1*−*i* < *length* (*N* ∝ *x1*)›
using *watch-by-S-w ge0 ge1i S-x assms i*
by (*auto simp*: *S*)
have *X2*: ‹*X2* = (*set-clauses-to-update-l* (*remove1-mset x1* (*clauses-to-update-l S'*)) *S'*, *x1*)›
using *SLw X2 S-S' x'* **unfolding** *i-def Sa*
by (*auto simp add*: *twl-st-wl eq-commute*[*of* - ‹- ! *w*›])
let *?L* = ‹*all-lits-st S*›
have *N-x1-in-L*: ‹*N* ∝ *x1* ! (*Suc 0* − *i*) ∈# *?L*›
using *dom i-le* **by** (*auto simp*: *ran-m-def S all-lits-of-mm-add-mset*
intro!: *in-clause-in-all-lits-of-m*
dest!: *multi-member-split*)
then have *L-i*: ‹*L* = *N* ∝ *x1* ! *i*›
using *j-w w-le L-watched ge0 ge1i SLw S-x i Sa SLw'* **unfolding** *i-def*
by (*auto simp*: *take-2-if twl-st-wl S keep-watch-def split*: *if-splits*)

have ‹((*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W* (*L* := (*W L*)[*j* := (*x1*, *L'*, *x2a*)])),
fst X2) ∈ *state-wl-l* (*Some* (*L*, *Suc w*))›
using *S-S' X2 j-w w-le SLw x' dom L' i SLw' dom* **unfolding** *Sa*
by (*cases* ‹*W L* ! *w*›)
(*auto simp*: *state-wl-l-def S keep-watch-def drop-map Cons-nth-drop-Suc*[*symmetric*])
moreover have ‹*n* = *size* {#(*i*, -) ∈# *mset* (*drop* (*Suc w*) (*watched-by S L*)).
i ∉# *dom-m* (*get-clauses-wl S*)#}›
using *dom n w-le*
by (*clarsimp-all simp add*: *SLw' Cons-nth-drop-Suc*[*symmetric*]
dest!: *multi-member-split*)
moreover {
have ‹*Suc 0* − *i* ≠ *i*›
using *i* **by** (*auto simp*: *i-def split*: *if-splits*)
then have ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*M*, *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*(*L* := (*W L*)[*j* := (*x1*, *L'*, *x2a*)]))›
using *SLw* **apply** −
apply (*rule correct-watching-except-update-blit*)
using *N-x1-in-L corr-Sa i-le distinct-N-x1 i-le bin SLw' L' i* **unfolding** *S Sa*
by (*auto simp*: *L-i keep-watch-def nth-eq-iff-index-eq S all-lits-def ac-simps*)
}
ultimately show *?thesis*
using *j-w w-le i alien-L' dom unit-T N-x1-in-L L' N-x1-in-L blit-in-lit*
unfolding *i*[*symmetric*] *T-def*[*symmetric*]
by (*auto simp*: *S update-blit-wl-def keep-watch-def all-lits-def ac-simps*
intro!: *ASSERT-leI blits-in-$\mathcal{L}_{in}$-keep-watch'*
blits-in-$\mathcal{L}_{in}$-propagate)

qed

have *set-conflict-wl*: ‹*set-conflict-wl x1 Sa*
≤ ⇓ {(*U*, *U'*). *get-conflict-l U'* ≠ *None* ∧ ((*Suc j*, *Suc w*, *U*), (*U'*, *0*)) ∈ *?unit*}
(*set-conflict-l* (*snd X2*) (*fst X2*))›
if
‹*clauses-to-update-l S'* ≠ {#} ∨ *0* < *n*› **and**
‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
‹*inres* (*mop-watched-by-at S L w*) *x'*› **and**
x': ‹(*x'*, ()) ∈ *?Slw*›

‹x2 =(x1a, x2a)›
‹x′ =(x1, x2)› **and**
‹(x1 ∉# dom-m (get-clauses-wl S), b) ∈ bool-rel› **and**
‹inres (mop-keep-watch L j w S) Sa› **and**
Sa: ‹(Sa, S′) ∈{(T, T′). T = keep-watch L j w S ∧ T′ = S′}› **and**
‹¬ x1 ∉# dom-m (get-clauses-wl Sa)› **and**
‹¬ b› **and**
‹clauses-to-update-l S′ ≠ {#}› **and**
X2: ‹(Sa, X2)
∈{(T, T′, C).
(T, T′) ∈ state-wl-l (Some(L, Suc w)) ∧
C = C′ ∧
T′ =
set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S′))
S′}› **and**
‹unit-propagation-inner-loop-body-l-inv L (snd X2) (fst X2)› **and**
‹(K, x) ∈ Id› **and**
‹K ∈ Collect ((=) x1a)› **and**
‹x ∈{K. K ∈ set (get-clauses-l (fst X2) ∝ snd X2)}› **and**
‹(val-K, v) ∈ Id› **and**
‹val-K ≠ Some True› **and**
‹v ≠ Some True› **and**
‹unit-prop-body-wl-inv Sa j w L› **and**
‹(ia, va) ∈{(j, j′). j = j′ ∧ j < 2 ∧ j = i}› **and**
‹(L′, vaa)
∈{(L, L′). L = L′ ∧ L = get-clauses-wl Sa ∝ x1 ! (1 − ia)}› **and**
‹(val-L′, vaaa) ∈ Id› **and**
‹val-L′ ≠ Some True› **and**
‹vaaa ≠ Some True› **and**
‹(f, x′a) ∈ ?find (get-trail-wl Sa) (get-clauses-wl Sa) x1› **and**
‹unit-prop-body-wl-find-unwatched-inv f x1 Sa› **and**
‹f = None› **and**
‹x′a = None› **and**
‹val-L′ = Some False› **and**
‹vaaa = Some False›
for x′ x1 x2 x1a x2a b Sa X2 K x val-K v ia va L′ vaa val-L′ vaaa f x′a
proof −
have [simp]: ‹correct-watching-except a b L (M, N, Some D′, NE, UE, NEk, UEk, NS, US, N0, U0, W, oth) ⟷
correct-watching-except a b L (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, W, oth)›
‹NO-MATCH {#} W ⟹ correct-watching-except a b L (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, W, oth) ⟷
correct-watching-except a b L (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, oth)›
for a b M N D′ NE UE W oth NS US N0 U0 NEk UEk
by (simp-all add: correct-watching-except.simps all-lits-st-def
set-conflict-wl-def prod.case clause-to-update-def get-clauses-l.simps)
have [simp]: ‹NO-MATCH None d ⟹ blits-in-$\mathcal{L}_{in}$(x1b, x1aa, d, x1c, x1d, NEk, UEk, NS, US, N0, U0, x2e, g) ⟷
blits-in-$\mathcal{L}_{in}$(x1b, x1aa, None, x1c, x1d, NEk, UEk, NS, US, N0, U0, x2e, g)› **and**
[simp]: ‹NO-MATCH {#} x2e ⟹ blits-in-$\mathcal{L}_{in}$(x1b, x1aa, None, x1c, x1d, NEk, UEk, NS, US, N0, U0, x2e, g) ⟷
blits-in-$\mathcal{L}_{in}$(x1b, x1aa, None, x1c, x1d, NEk, UEk, NS, US, N0, U0, {#}, g)› **for** x1b x1aa x1c x1d x2e g d NS US N0 U0 NEk UEk
unfolding blits-in-$\mathcal{L}_{in}$-def **by** auto
have ‹get-conflict-wl (keep-watch L j w S) = None›
using confl-S **by** (cases S) (auto simp: keep-watch-def)

then have ‹*set-conflict-wl-pre D* (*keep-watch L j w S*) ⟹
set-conflict-wl D (*keep-watch L j w S*) ≤ (*SPEC*(*correct-watching-except* (*Suc j*) (*Suc w*) *L*))› **for** *D*
using *S-S′ SLw w-le j-w n that corr-w*
correct-watching-except-correct-watching-except-Suc-Suc-keep-watch[*of j w S L*]
by (*cases* ‹*keep-watch L j w S*›)
(*auto intro*!: *frefI nres-relI simp*: *set-conflict-wl-def state-wl-l-def*
keep-watch-state-wl
corr-w correct-watching-except-correct-watching-except-Suc-notin
intro!: *ASSERT-refine-right intro*: *ASSERT-leI*)

moreover have *N-x1-in-L*: ‹*get-clauses-wl S* ∝ *x1* ! (*Suc 0* − *i*) ∈# *all-lits-st S*›
using *that i-le2* **by** (*auto simp*: *ran-m-def all-lits-of-mm-add-mset all-lits-def*
remove-one-lit-from-wq-def eq-commute[*of* - ‹- ! *w*›] *all-lits-st-def*
intro!: *in-clause-in-all-lits-of-m nth-mem*
dest!: *multi-member-split*[*of x1*])
ultimately show *?thesis*
using *S-S′ Sa X2 SLw w-le j-w n corr-w n blit-in-lit x′*
unfolding *set-conflict-wl-def set-conflict-l-def*
apply (*cases S*; *cases S′*; *cases X2*; *cases* ‹*watched-by S L* ! *w*›)
apply (*refine-rcg*)
subgoal premises *p* **for** *a ba c d e fa g aa baa ca da ea faa ga x1b x2b x1aa x2aa x1ba x2ba x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h x1i x2i x1j x2j x1k x2k* **unfolding** *set-conflict-wl-pre-def*
using *p* **apply** − **by** (*rule exI*[*of* - ‹*fst X2*›], *rule exI*[*of* - ‹*Some*(*L*, *Suc w*)›])
(*auto simp add*: *keep-watch-def eq-commute* [*of* - ‹- ! *w*›] *all-lits-def ac-simps*
intro!: *blits-in-*$\mathcal{L}_{in}$*-keep-watch′ blits-in-*$\mathcal{L}_{in}$*-propagate*
simp del:)
using *that*
by (*simp-all flip*: *all-lits-alt-def2 all-lits-def*
add: *set-conflict-l-def set-conflict-wl-def state-wl-l-def*
keep-watch-state-wl keep-watch-def drop-map Cons-nth-drop-Suc[*symmetric*]
correct-watching-except-correct-watching-except-Suc-Suc-keep-watch
corr-w correct-watching-except-correct-watching-except-Suc-notin
blits-in-$\mathcal{L}_{in}$*-keep-watch′ blits-in-*$\mathcal{L}_{in}$*-propagate*)
qed

have *propagate-lit-wl-general*: ‹*propagate-lit-wl-general L′ x1 ia Sa*
≤ ⇓ {(*U*, *U′*). ((*Suc j*, *Suc w*, *U*), (*U′*, *if get-conflict-l U′* = *None then n else 0*)) ∈ *?unit*}
(*propagate-lit-l vaa* (*snd X2*) *va* (*fst X2*))›
if
‹*clauses-to-update-l S′* ≠ {#} ∨ *0* < *n*› **and**
‹*unit-propagation-inner-loop-l-inv L* (*S′*, *n*)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
‹*inres* (*mop-watched-by-at S L w*) *x′*› **and**
Slw: ‹(*x′*, ()) ∈ *?Slw*› **and**
x′: ‹*x2* =(*x1a*, *x2a*)›
‹*x′* =(*x1*, *x2*)› **and**
‹(*x1* ∉# *dom-m* (*get-clauses-wl S*), *b*) ∈ *bool-rel*› **and**
Sa: ‹(*Sa*, *S′*) ∈{(*T*, *T′*). *T* = *keep-watch L j w S* ∧ *T′* = *S′*}› **and**
x1-dom: ‹¬ *x1* ∉# *dom-m* (*get-clauses-wl Sa*)› **and**
‹¬ *b*› **and**
‹*clauses-to-update-l S′* ≠ {#}› **and**
X2: ‹(*Sa*, *X2*)
∈{(*T*, *T′*, *C*).
(*T*, *T′*) ∈ *state-wl-l* (*Some*(*L*, *Suc w*)) ∧
C = *C′* ∧

```
      T' =
      set-clauses-to-update-l (remove1-mset C (clauses-to-update-l S'))
       S'}› and
    l-inv: ‹unit-propagation-inner-loop-body-l-inv L (snd X2) (fst X2)› and
    ‹(K, x) ∈ Id› and
    ‹K ∈ Collect ((=) x1a)› and
    ‹x ∈{K. K ∈ set (get-clauses-l (fst X2) ∝ snd X2)}› and
    ‹(val-K, v) ∈ Id› and
    ‹val-K ≠ Some True› and
    ‹v ≠ Some True› and
    ‹unit-prop-body-wl-inv Sa j w L› and
    i: ‹(ia, va) ∈{(j, j'). j = j' ∧ j < 2 ∧ j = i}› and
    L': ‹(L', vaa)
     ∈{(L, L'). L = L' ∧ L = get-clauses-wl Sa ∝ x1 ! (1 − ia)}› and
    ‹(val-L', vaaa) ∈ Id› and
    ‹val-L' ≠ Some True› and
    ‹vaaa ≠ Some True› and
    ‹(f, x'a) ∈ ?find (get-trail-wl Sa) (get-clauses-wl Sa) x1› and
    ‹unit-prop-body-wl-find-unwatched-inv f x1 Sa› and
    ‹f = None› and
    ‹x'a = None› and
    ‹val-L' ≠ Some False› and
    ‹vaaa ≠ Some False›
  for x' x1 x2 x1a x2a b Sa X2 K x val-K v ia va L' vaa val-L' vaaa f x'a
proof −
  define i' :: nat where ‹i' ≡ if get-clauses-wl (keep-watch L j w S) ∝ x1 ! 0 = L then 0 else 1›
  have x[simp]: ‹watched-by S L ! w = (x1, x1a, x2a)› ‹Sa = keep-watch L j w S›
    using x' Slw Sa by auto
  have i-alt-def: ‹i' = (if get-clauses-l (fst X2) ∝ snd X2 ! 0 = L then 0 else 1)› ‹i = i'›
    using X2 S-S' i unfolding i'-def x' i-def by (auto)
  have x1-dom[simp]: ‹x1 ∈# dom-m (get-clauses-wl S)›
    using x1-dom X2 by (cases S) (auto simp: keep-watch-def)
  have [simp]: ‹get-clauses-wl S ∝ x1 ! 0 ≠ L ⟹ get-clauses-wl S ∝ x1 ! Suc 0 = L› and
    ‹Suc 0 < length (get-clauses-wl S ∝ x1)›
    using l-inv X2 S-S' SLw unfolding unit-propagation-inner-loop-body-l-inv-def
    apply − apply normalize-goal+
    by (cases ‹get-clauses-wl S ∝ x1›; cases ‹tl (get-clauses-wl S ∝ x1)›)
      auto

  have n: ‹n = size {#(i, -) ∈# mset (drop (Suc w) (watched-by S L)).
      i ∉# dom-m (get-clauses-wl S)#}›
    using n
    apply (subst (asm) Cons-nth-drop-Suc[symmetric])
    subgoal using w-le by simp
    subgoal using n SLw X2 S-S' unfolding i-def by auto
    done
  have [simp]: ‹get-conflict-l (fst X2) = get-conflict-wl S›
    using X2 S-S' by auto

  have eq: ‹a = b ∧ a = get-clauses-wl S ⟹ (a, b) ∈ {(a, b). a = b ∧ b = get-clauses-wl S}›
    for a b
    by auto
  have ignored: ‹P ≤ SPEC P' ⟹ P ≤ ⇓ {(S, S'). P' S} (RETURN Q)› for P P' Q
    by (auto simp: conc-fun-RES RETURN-def)
  have [simp]: ‹i≤ Suc 0›
    by (simp add: i'-def i-alt-def(2))
```

have ‹*get-clauses-wl S* ∝ *x1* ! (*Suc 0* − *i*) ∈# *all-lits-st S*›
using *i-le x1-dom* ‹*Suc 0* < *length* (*get-clauses-wl S* ∝ *x1*)›
by (*auto simp*: *i′-def*)
then have
‹(*propagate-lit-wl-general* (*get-clauses-wl S* ∝ *x1* ! (*Suc 0* − *i*)) *x1 i* (*keep-watch L j w S*)
≤ ⇓ (*state-wl-l* (*Some* (*L*, *Suc w*)))
(*propagate-lit-l* (*get-clauses-l* (*fst X2*) ∝ *snd X2* ! (*Suc 0* − *i*)) (*snd X2*) *i* (*fst X2*)))›
using *X2 S-S′ SLw j-w w-le multi-member-split*[*OF x1-dom*] *confl-S x*
by (*cases S*; *cases S′*)
(*auto simp*: *state-wl-l-def propagate-lit-wl-general-def keep-watch-def*
propagate-lit-l-def drop-map mop-clauses-swap-def cons-trail-propagate-l-def
simp flip: *all-lits-alt-def2*
intro!: *ASSERT-refine-right*)
have ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L* (*keep-watch L j w S*)›
by (*simp add*: *corr-w correct-watching-except-correct-watching-except-Suc-Suc-keep-watch j-w w-le*)
then have ‹*propagate-lit-wl-general L′ x1 ia Sa*
≤ ⇓ {(*S*, *S′*). (*correct-watching-except* (*Suc j*) (*Suc w*) *L S*)}
(*propagate-lit-l vaa* (*snd X2*) *va* (*fst X2*))›
using *X2 Sa*
apply (*cases X2*, *cases* ‹*fst X2*›, *hypsubst*)
unfolding *propagate-lit-l-def cons-trail-propagate-l-def nres-monad3*
mop-clauses-swap-def If-bind-distrib **apply** −
apply (*clarsimp*; *intro conjI impI ASSERT-refine-right ignored*;
(*rule correct-watching-except-correct-watching-except-propagate-lit-wl*)*?*)
using *w-le j-w* ‹*Suc 0* < *length* (*get-clauses-wl S* ∝ *x1*)› *confl-S i S-S′ L′*
by (*cases S*; *auto simp*: *keep-watch-def state-wl-l-def simp flip*: *all-lits-alt-def2*; *fail*)+

moreover have *N-x1-in-L*: ‹*x1a* ∈# *all-lits-st S*›
using *that i-le2* **by** (*auto simp*: *ran-m-def all-lits-of-mm-add-mset all-lits-def*
remove-one-lit-from-wq-def eq-commute[*of* - ‹- ! *w*›] *ac-simps*
intro!: *in-clause-in-all-lits-of-m nth-mem*
dest!: *multi-member-split*[*of x1*])
then have ‹*propagate-lit-wl-general L′ x1 ia Sa*
≤ ⇓ {(*S*, *S′*). (*blits-in-*$\mathcal{L}_{in}$ *S*)}
(*propagate-lit-l vaa* (*snd X2*) *va* (*fst X2*))›
using *X2 Sa*
apply (*cases X2*, *cases* ‹*fst X2*›, *hypsubst*)
unfolding *propagate-lit-l-def propagate-lit-wl-general-def cons-trail-propagate-l-def nres-monad3*
mop-clauses-swap-def If-bind-distrib **apply** −
apply (*clarsimp*; *intro conjI impI ASSERT-refine-right ignored*)
using *w-le j-w* ‹*Suc 0* < *length* (*get-clauses-wl S* ∝ *x1*)› *confl-S i S-S′ L′ x x1-dom blit-in-lit*
apply (*cases S*; *auto simp*: *keep-watch-def state-wl-l-def blits-in-*$\mathcal{L}_{in}$*-keep-watch′*
blits-in-$\mathcal{L}_{in}$*-propagate simp flip*: *all-lits-alt-def2 all-lits-def*)
using *w-le j-w* ‹*Suc 0* < *length* (*get-clauses-wl S* ∝ *x1*)› *confl-S i S-S′ L′ x x1-dom blit-in-lit*
apply (*cases S*; *auto simp*: *keep-watch-def state-wl-l-def blits-in-*$\mathcal{L}_{in}$*-keep-watch′*
blits-in-$\mathcal{L}_{in}$*-propagate simp flip*: *all-lits-alt-def2 all-lits-def*)
done
moreover have ‹*propagate-lit-wl-general L′ x1 ia Sa*
≤ ⇓{(*U*, *U′*).
((*Suc j*, *Suc w*, *U*), *U′*, *if get-conflict-l U′* = *None then n else 0*)
∈{((*i*, *j*, *T′*), *T*, *n*).
(*T′*, *T*) ∈ *state-wl-l* (*Some* (*L*, *Suc w*)) ∧
j ≤ *length* (*watched-by T′ L*) ∧
length (*watched-by S L*) = *length* (*watched-by T′ L*) ∧
i ≤ *j* ∧

```
        (get-conflict-wl T' = None ⟶
         n =
         size
          {#(i, -)∈# mset (drop j (watched-by T' L)).
           i ∉# dom-m (get-clauses-wl T')#}) ∧
        (get-conflict-wl T' ≠ None ⟶ n = 0)}}
    (propagate-lit-l vaa (snd X2) va (fst X2))›
   using w-le n j-w x1-dom that S-S' confl-S i-le2 x
   unfolding i'-def[symmetric] i-alt-def[symmetric] propagate-lit-wl-general-def
     propagate-lit-l-def If-bind-distrib
   apply (refine-rcg refine-itself2[of cons-trail-propagate-l, THEN fref-to-Down-curry2])
   subgoal by (cases S) (auto simp: keep-watch-def)
   subgoal by (cases S) (auto simp: keep-watch-def)
   subgoal using S-S' by (cases S) (auto simp: keep-watch-def state-wl-l-def all-lits-st-def
       all-lits-of-st-l-def all-lits-def
     simp flip: all-lits-alt-def2)
   subgoal by fast
   subgoal by (auto simp: state-wl-l-def)
   subgoal by (auto simp: state-wl-l-def)
   subgoal by (auto simp: state-wl-l-def)
   apply (rule mop-clauses-swap-itself-spec2)
   subgoal by (auto simp: state-wl-l-def)
   subgoal
    by (cases X2; cases S; cases S')
     (clarsimp simp: state-wl-l-def mop-clauses-swap-def drop-map
      op-clauses-swap-def keep-watch-def)
   apply (rule eq)
   subgoal by (cases X2; cases S; cases S')
     (auto simp: twl-st-wl simp: op-clauses-swap-def keep-watch-def
        propagate-lit-l-def mop-clauses-swap-def drop-map state-wl-l-def
       intro!: ASSERT-refine-right)
   subgoal by (cases X2; cases S; cases S')
     (clarsimp simp add:  op-clauses-swap-def keep-watch-def
      propagate-lit-l-def mop-clauses-swap-def  state-wl-l-def
      intro!: ASSERT-refine-right)
   done
 ultimately show ?thesis
   apply (rule nres-add-unrelated[OF nres-add-unrelated3, THEN order-trans])
   apply (rule conc-fun-R-mono)
   apply fastforce
   done
qed

have update-blit: ‹update-blit-wl L x1 x2a j w Ka Sa
    ≤ ⇓?unit
       (RETURN(fst X2, if get-conflict-l (fst X2) = None then n else 0))›
 if
   cond: ‹clauses-to-update-l S' ≠ {#} ∨ 0 < n› and
   loop-inv: ‹unit-propagation-inner-loop-l-inv L (S', n)› and
   ‹unit-propagation-inner-loop-wl-loop-pre L (j, w, S)› and
   ‹inres (mop-watched-by-at S L w) x'› and
   x': ‹(x', ()) ∈ ?Slw›
     ‹x2 =(x1a, x2a)›
     ‹x' =(x1, x2)› and
   ‹(x1 ∉# dom-m (get-clauses-wl S), b) ∈ bool-rel› and
   ‹inres (mop-keep-watch L j w S) Sa› and
```

Sa: ‹(*Sa*, *S'*) ∈{(*T*, *T'*). *T* = *keep-watch L j w S* ∧ *T'* = *S'*}› **and**
dom: ‹¬ *x1* ∉# *dom-m* (*get-clauses-wl Sa*)› **and**
‹¬ *b*› **and**
‹*clauses-to-update-l S'* ≠ {#}› **and**
X2: ‹(*Sa*, *X2*)
∈{(*T*, *T'*, *C*).
(*T*, *T'*) ∈ *state-wl-l* (*Some*(*L*, *Suc w*)) ∧
C = *C'* ∧
T' =
set-clauses-to-update-l (*remove1-mset C* (*clauses-to-update-l S'*))
S'}› **and**
‹*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)› **and**
‹(*K*, *x*) ∈ *Id*› **and**
‹*K* ∈ *Collect* ((=) *x1a*)› **and**
‹*x* ∈{*K*. *K* ∈ *set* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)}› **and**
‹(*val-K*, *v*) ∈ *Id*› **and**
‹*val-K* ≠ *Some True*› **and**
‹*v* ≠ *Some True*› **and**
‹*unit-prop-body-wl-inv Sa j w L*› **and**
‹(*ia*, *va*) ∈{(*j*, *j'*). *j* = *j'* ∧ *j* < 2 ∧ *j* = *i*}› **and**
‹(*L'*, *vaa*)
∈{(*L*, *L'*). *L* = *L'* ∧ *L* = *get-clauses-wl Sa* ∝ *x1* ! (1 − *ia*)}› **and**
‹(*val-L'*, *vaaa*) ∈ *Id*› **and**
‹*val-L'* ≠ *Some True*› **and**
‹*vaaa* ≠ *Some True*› **and**
fx': ‹(*f*, *x'a*) ∈ *?find* (*get-trail-wl Sa*) (*get-clauses-wl Sa*) *x1*›
‹*f* = *Some xa*›
‹*x'a* = *Some x'b*› **and**
‹*unit-prop-body-wl-find-unwatched-inv f x1 Sa*› **and**
‹(*xa*, *x'b*) ∈ *nat-rel*› **and**
‹*x'b* < *length* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)› **and**
Ka: ‹(*Ka*, *Kb*) ∈{(*L*, *L'*). *L* = *L'* ∧ *L* = *get-clauses-wl Sa* ∝ *x1* ! *xa*}› **and**
‹(*val-L'a*, *val-Ka*) ∈ *Id*› **and**
‹*val-L'a* = *Some True*› **and**
‹*val-Ka* = *Some True*›
for *x' x1 x2 x1a x2a b Sa X2 K x val-K v ia va L' vaa val-L' vaaa f x'a xa*
x'b Ka Kb val-L'a val-Ka
proof −
have *confl*: ‹*get-conflict-wl S* = *None*›
using *S-S' loop-inv cond* **unfolding** *unit-propagation-inner-loop-l-inv-def prod.case* **apply** −
by *normalize-goal+ auto*

have *unit-T*: ‹*unit-propagation-inner-loop-body-l-inv L C' T*›
using *that* **by** (*auto simp*: *remove-one-lit-from-wq-def*)

have ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L* (*keep-watch L j w S*)›
by (*simp add*: *corr-w correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
j-w w-le)
moreover have ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*a*, *b*, *None*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *ga*(*L* := (*ga L*)[*j* := (*x1*, *b* ∝ *x1* ! *xa*, *x2a*)]))›
if
corr: ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*a*, *b*, *None*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *ga*(*L* := (*ga L*)[*j* := (*x1*, *x1a*, *x2a*)]))› **and**
‹*ga L* ! *w* = (*x1*, *x1a*, *x2a*)› **and**
S[*simp*]: ‹*S* = (*a*, *b*, *None*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *ga*)› **and**
‹*X2* = (*set-clauses-to-update-l* (*remove1-mset x1* (*clauses-to-update-l S'*)) *S'*, *x1*)› **and**

```
    ‹(a, b, None, d, e, NEk, UEk, NS, US, N0, U0,
   {#i ∈# mset (drop (Suc w) (map fst ((ga L)[j := (x1, x1a, x2a)]))). i ∈# dom-m b#}, f) =
   set-clauses-to-update-l (remove1-mset x1 (clauses-to-update-l S')) S'›
   for a :: ‹('v literal, 'v literal,nat) annotated-lit list› and
     b :: ‹(nat, 'v literal list ×  bool) fmap› and
     d e NS US N0 U0 NEk UEk :: ‹'v literal multiset multiset› and
     f :: ‹'v literal multiset› and
     ga :: ‹'v literal ⇒ (nat × 'v literal × bool) list›
proof −
  have ‹b ∝ x1 ! xa ∈# all-lits-st (a, b, None, d, e, NEk, UEk, NS, US, N0, U0, f, ga)›
    using dom fx' Sa by (auto simp: ran-m-def all-lits-of-mm-add-mset x' f twl-st-wl
        dest!: multi-member-split
        simp: all-lits-def
        intro!: in-clause-in-all-lits-of-m)
  moreover have ‹b ∝ x1 ! xa ∈ set (b ∝ x1)›
    using dom fx' Sa by (auto simp: ran-m-def all-lits-of-mm-add-mset x' f twl-st-wl
        dest!: multi-member-split
        intro!: in-clause-in-all-lits-of-m)

  moreover have ‹b ∝ x1 ! xa ≠ L›
    using X2 L-def[OF unit-T] S-S' SLw fx' x' that Sa
    by (auto simp: polarity-def split: if-splits)
  moreover have ‹correctly-marked-as-binary b (x1, b ∝ x1 ! xa, x2a)›
    using correctly-marked-as-binary unit-T dom Sa SLw
    by (auto simp: correctly-marked-as-binary.simps ‹ga L ! w = (x1, x1a, x2a)›)
  ultimately show ?thesis
    by (rule correct-watching-except-update-blit[OF corr])
qed
moreover have in-all: ‹get-clauses-wl S ∝ x1 ! xa ∈# all-lits-st S›
  using dom fx' Sa unfolding x'
  by (auto dest!: multi-member-split[of x1]
    simp: ran-m-def all-lits-of-mm-add-mset all-lits-of-mm-union
      in-clause-in-all-lits-of-m all-lits-def)
moreover have w: ‹watched-by S L ! w = (x1, x1a, x2a)›
  using x' by auto
ultimately have u: ‹update-blit-wl L x1 x2a j w Ka Sa
≤ SPEC(λ(i, j, T'). correct-watching-except i j L T')›
  using X2 confl Sa SLw alien-L' unit-T Ka dom  j-w w-le unfolding T-def[symmetric]
    update-blit-wl-def
  apply (cases S; cases Sa, hypsubst) apply simp
  apply  (auto simp: keep-watch-def state-wl-l-def
      simp flip: all-lits-alt-def2 all-lits-def
      intro!: ASSERT-leI)
  done
have b: ‹update-blit-wl L x1 x2a j w Ka Sa
≤ SPEC(λ(i, j, T'). blits-in-\mathcal{L}_in T')›
 using X2 confl Sa SLw alien-L' in-all unit-T Ka blit-in-lit j-w w-le dom unfolding T-def[symmetric]
    update-blit-wl-def
  apply (cases S; cases Sa, hypsubst) apply simp
  apply  (auto simp: keep-watch-def state-wl-l-def blits-in-\mathcal{L}_in-keep-watch'
        blits-in-\mathcal{L}_in-propagate all-lits-def ac-simps
      intro!: ASSERT-leI blits-in-\mathcal{L}_in-keep-watch')
  done
have ‹get-conflict-wl S = None›
  using S-S' loop-inv cond unfolding unit-propagation-inner-loop-l-inv-def prod.case apply −
  by normalize-goal+ auto
```

moreover have ‹n = *size* {#(i, -) ∈# *mset* (*drop* (*Suc* w) (*watched-by* S L)). i ∉# *dom-m* (*get-clauses-wl* S)#}›
using n *dom X2 w-le S-S′ SLw Sa*
by (*auto simp*: *Cons-nth-drop-Suc*[*symmetric*] w)
ultimately have ‹*update-blit-wl L x1 x2a j w Ka Sa*
≤ ⇓{((i, j, T'), T, n).
(T', T) ∈ *state-wl-l* (*Some*(L, j)) ∧
j ≤ *length* (*watched-by* T' L) ∧
length (*watched-by* S L) = *length* (*watched-by* T' L) ∧
i ≤ j ∧
(*get-conflict-wl* T' = *None* ⟶
n =
size
{#(i, -)∈# *mset* (*drop* j (*watched-by* T' L)).
i ∉# *dom-m* (*get-clauses-wl* T')#}) ∧
(*get-conflict-wl* T' ≠ *None* ⟶ n = 0)}
(*RETURN*(*fst X2*, *if get-conflict-l* (*fst X2*) = *None then n else 0*))›
using *j-w w-le S-S′ X2 alien-L′ unit-T in-all Sa in-all Ka dom* **unfolding** *T-def*[*symmetric*]
by (*cases* S)
(*auto simp*: *update-blit-wl-def keep-watch-def state-wl-l-def drop-map w*
simp flip: *all-lits-alt-def2 all-lits-def*
intro!: *ASSERT-leI*)

then show *?thesis*
apply (*rule nres-add-unrelated2*[*OF SPEC-rule-conjI*[*OF u b*] -, *THEN order-trans*])
apply (*rule conc-fun-R-mono*)
apply *fastforce*
done
qed

have *find-is-nat-rel*: ‹x ∈ *?find a b c* ⟹ x ∈ ⟨*nat-rel*⟩*option-rel*› **for** x a b c
by *auto*
have *update-clause-wl*: ‹*update-clause-wl L L′ x1 x2a j w ia xa Sa*
≤ ⇓*?unit*
(*do*{
T ← *update-clause-l* (*snd X2*) *va x′b* (*fst X2*);
RETURN(T, *if get-conflict-l* T = *None then n else 0*)
})›
if
cond: ‹*clauses-to-update-l* S' ≠ {#} ∨ 0 < n› **and**
loop-inv: ‹*unit-propagation-inner-loop-l-inv* L (S', n)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre* L (j, w, S)› **and**
‹*inres* (*mop-watched-by-at* S L w) x'› **and**
x': ‹(x', ())
∈ *?Slw*› **and**
[*simp*]: ‹*x2* =(*x1a*, *x2a*)› ‹x' =(*x1*, *x2*)› **and**
‹(*x1* ∉# *dom-m* (*get-clauses-wl* S), b) ∈ *bool-rel*› **and**
‹*inres* (*mop-keep-watch L j w S*) *Sa*› **and**
Sa: ‹(*Sa*, S') ∈{(T, T'). T = *keep-watch L j w S* ∧ T' = S'}› **and**
dom: ‹¬ *x1* ∉# *dom-m* (*get-clauses-wl Sa*)› **and**
‹¬ b› **and**
‹*clauses-to-update-l* S' ≠ {#}› **and**
X2: ‹(*Sa*, *X2*)
∈{(T, T', C).
(T, T') ∈ *state-wl-l* (*Some*(L, *Suc* w)) ∧
C = C' ∧

$T' = \textit{set-clauses-to-update-l}\ (\textit{remove1-mset}\ C\ (\textit{clauses-to-update-l}\ S'))\ S'\}\rangle$ **and**
pre: ‹*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)› **and**
‹(*K*, *x*) ∈ *Id*› **and**
‹*K* ∈ *Collect* ((=) *x1a*)› **and**
‹*x* ∈{*K*. *K* ∈ *set* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)}› **and**
‹(*val-K*, *v*) ∈ *Id*› **and**
‹*val-K* ≠ *Some True*› **and**
‹*v* ≠ *Some True*› **and**
wl-inv: ‹*unit-prop-body-wl-inv Sa j w L*› **and**
i: ‹(*ia*, *va*) ∈{(*j*, *j'*). *j* = *j'* ∧ *j* < *2* ∧ *j* = *i*}› **and**
L': ‹(*L'*, *vaa*)
∈{(*L*, *L'*). *L* = *L'* ∧ *L* = *get-clauses-wl Sa* ∝ *x1* ! (*1* − *ia*)}› **and**
‹(*val-L'*, *vaaa*) ∈ *Id*› **and**
‹*val-L'* ≠ *Some True*› **and**
‹*vaaa* ≠ *Some True*› **and**
fx': ‹(*f*, *x'a*) ∈ *?find* (*get-trail-wl Sa*) (*get-clauses-wl Sa*) *x1*› **and**
‹*unit-prop-body-wl-find-unwatched-inv f x1 Sa*› **and**
[*simp*]: ‹*f* = *Some xa*› ‹*x'a* = *Some x'b*› **and**
‹(*xa*, *x'b*) ∈ *nat-rel*› **and**
‹*x'b* < *length* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)› **and**
Ka: ‹(*Ka*, *Kb*) ∈{(*L*, *L'*). *L* = *L'* ∧ *L* = *get-clauses-wl Sa* ∝ *x1* ! *xa*}› **and**
‹(*val-L'a*, *val-Ka*) ∈ *Id*› **and**
‹*val-L'a* ≠ *Some True*› **and**
‹*val-Ka* ≠ *Some True*›
for *x' x1 x2 x1a x2a b Sa X2 K x val-K v ia va L' vaa val-L' vaaa f x'a xa*
x'b Ka Kb val-L'a val-Ka
proof −
have [*simp*]: ‹*xa* = *x'b*› ‹*Ka* = *Kb*› ‹*Kb* = *get-clauses-wl Sa* ∝ *x1* ! *xa*› ‹*ia* = *va*›
using *fx' Ka i* **by** *auto*
have *update-clause-l-alt-def*:
‹*update-clause-l* = (λ*C i f* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT (*update-clause-l-pre C i f* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
K ← *RETURN* (*op-clauses-at N C f*);
N' ← *mop-clauses-swap N C i f*;
RETURN (*M*, *N'*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
})›
unfolding *update-clause-l-def* **by** (*auto intro*!: *ext*)
have *unit-T*: ‹*unit-propagation-inner-loop-body-l-inv L C' T*›
using *that* **by** (*auto simp*: *remove-one-lit-from-wq-def*)
have *w*: ‹*watched-by S L* ! *w* = (*x1*, *x1a*, *x2a*)›
using *x'* **by** *auto*
have *alien-K*: ‹*Ka* ∈# *all-lits-st Sa*›
using *Sa dom fx'* **unfolding** *all-lits-def*
by (*cases S*; *cases Sa*)
(*auto simp*: *keep-watch-def ran-m-def all-lits-of-mm-add-mset*
intro!: *in-clause-in-all-lits-of-m*
dest!: *multi-member-split*)
have *L-in-watched*: ‹*L* ∈ *set* (*watched-l* (*get-clauses-wl Sa* ∝ *x1*))›
using *corr-w alien-L' Sa X2 w-le w dom* **by** (*cases S*)
(*clarsimp simp*: *correct-watching-except-alt-def keep-watch-def*
simp flip: *Cons-nth-drop-Suc all-lits-alt-def2 all-lits-def*
dest!: *multi-member-split*)
then have *LKa*: ‹*L* ≠ *Ka*›
using *fx' Ka* **by** (*auto simp*: *in-set-conv-nth nth-eq-iff-index-eq*)
have *L*: ‹*get-clauses-wl Sa* ∝ *x1* ! *i* = *L*› ‹*get-clauses-wl Sa* ∝ *x1* ! (*Suc 0* − *i*) ≠ *L*›

```
    ‹get-clauses-wl Sa ∝ x1 ! (Suc 0 − i) ≠ Ka›
   using i Sa w L-in-watched fx′ Ka unfolding i-def
   by (auto simp: in-set-conv-nth nth-eq-iff-index-eq)
  have corr: ‹correct-watching-except (Suc j) (Suc w) L Sa›
   using wl-inv dom w Sa unfolding unit-prop-body-wl-inv-def by auto
  have confl: ‹get-conflict-wl S = None›
   using S-S′ loop-inv cond unfolding unit-propagation-inner-loop-l-inv-def prod.case apply −
   by normalize-goal+ auto
 have
  alien-L′′:
    ‹L′ ∈# all-lits-st Sa›
  using L′ i-le2[OF unit-T] i dom Sa w
  by (auto intro!: nth-in-all-lits-stI)

  show ?thesis
   supply RETURN-as-SPEC-refine[refine2 del]
   unfolding update-clause-l-alt-def update-clause-wl-def
   apply (cases X2; cases ‹fst X2›; cases Sa; hypsubst; unfold fst-conv; hypsubst;
     unfold prod.case snd-conv nres-monad3)
   apply (refine-rcg mop-clauses-at-op-clauses-at-spec2
      mop-clauses-swap-itself-spec2)
   subgoal using alien-L′ Sa X2 dom by (cases S; auto simp: keep-watch-def)
   subgoal using alien-L′ Sa X2 j-w w-le dom by (cases S; auto simp: keep-watch-def)
   subgoal using alien-L′ Sa X2 j-w w-le dom by (cases S; auto simp: keep-watch-def)
   subgoal using alien-L′ Sa X2 j-w w-le dom corr by (cases S; auto simp: keep-watch-def)
   subgoal using alien-L′ Sa X2 by (cases S; auto simp: keep-watch-def
       simp flip: all-lits-alt-def2 all-lits-def)
   subgoal using alien-L′′ by auto
   subgoal using dom Sa X2 w S-S′ by (cases S; cases S′; auto simp: state-wl-l-def keep-watch-def)
  subgoal using fx′ Sa S-S′ X2 w dom by (cases S; cases S′; auto simp: state-wl-l-def keep-watch-def)
    subgoal using fx′ Sa S-S′ X2 w alien-K by (cases S; cases S′; auto simp: state-wl-l-def
keep-watch-def all-lits-def)
   subgoal using fx′ Sa S-S′ X2 w alien-K by (cases S; cases S′; auto simp: state-wl-l-def
     keep-watch-def all-lits-def ac-simps)
   subgoal using LKa Ka by (auto simp: state-wl-l-def keep-watch-def)
   subgoal using fx′ Sa S-S′ X2 w by (cases S; cases S′; auto simp: state-wl-l-def keep-watch-def)

   subgoal supply [[goals-limit=1]]
     using Sa S-S′ X2 w w-le j-w n corr LKa i L-in-watched Ka L fx′ blit-in-lit alien-L′ L′
       alien-K confl x′ unfolding all-lits-def[symmetric] all-lits-alt-def[symmetric]
     apply (cases S; cases S′)
    apply (clarsimp simp add: state-wl-l-def keep-watch-def op-clauses-swap-def blits-in-\mathcal{L}_{in}-keep-watch′
          blits-in-\mathcal{L}_{in}-propagate correct-watching-except-correct-watching-except-update-clause
       simp flip: Cons-nth-drop-Suc
       intro!: blits-in-\mathcal{L}_{in}-keep-watch′′)
     apply (intro conjI correct-watching-except-correct-watching-except-update-clause
       blits-in-\mathcal{L}_{in}-keep-watch′′)
     apply (clarsimp intro!: correct-watching-except-correct-watching-except-update-clause)[]
     apply (auto simp: state-wl-l-def keep-watch-def op-clauses-swap-def blits-in-\mathcal{L}_{in}-keep-watch′
          blits-in-\mathcal{L}_{in}-propagate all-lits-def ac-simps
       simp flip: Cons-nth-drop-Suc dest: in-set-takeD
       intro!: correct-watching-except-correct-watching-except-update-clause
          blits-in-\mathcal{L}_{in}-keep-watch′′)
     done
   done
 qed
```

have *in-watched-keepD*: ‹$x \in set\ (watched\text{-}by\ (keep\text{-}watch\ L\ j\ w\ S)\ L') \Longrightarrow x \in set\ (watched\text{-}by\ S\ L')$›
  for $x\ L'$
  using *j-w w-le* **by** (*cases S*) (*clarsimp simp*: *keep-watch-def elim*!: *in-set-upd-cases split*: *if-splits*)

have *is-nondeleted-clause-pre*: ‹*is-nondeleted-clause-pre x1 L Sa*›
  if
    ‹*clauses-to-update-l S'* $\neq \{\#\} \vee 0 < n$› **and**
    ‹*unit-propagation-inner-loop-l-inv L* (S', n)› **and**
    ‹*unit-propagation-inner-loop-wl-loop-pre L* (j, w, S)› **and**
    ‹*inres* (*mop-watched-by-at S L w*) *x'*› **and**
    x': ‹$(x', ()) \in$ *?Slw*› **and**
    [*simp*]: ‹*x2* = (*x1a*, *x2a*)›
      ‹*x'* = (*x1*, *x2*)› **and**
    ‹$(x1 \notin\# dom\text{-}m\ (get\text{-}clauses\text{-}wl\ S), b) \in bool\text{-}rel$› **and**
    ‹*inres* (*mop-keep-watch L j w S*) *Sa*› **and**
    Sa: ‹$(Sa, S') \in \{(T, T').\ T = keep\text{-}watch\ L\ j\ w\ S \wedge T' = S'\}$›
  for *x' x1 x2 x1a x2a b Sa*
proof −
  have ‹*watched-by Sa L* ! *w* = *x'*› ‹*w* < *length* (*watched-by Sa L*)›
    using *that* **using** *j-w w-le* **by** *auto*
  then have ‹*x'* ∈ *set* (*watched-by Sa L*)›
    using *j-w w-le* **by** *force*
  then show *?thesis* **using** *j-w w-le x' alien-L' Sa*
    by (*auto simp*: *is-nondeleted-clause-pre-def eq-commute*[*of* ‹(-, -)› ‹- ! *w*›]
      simp flip: *all-lits-alt-def2*)
qed

have *other-watched-wl-itself-spec2*[*refine*]:
  ‹$((N, C, i, j), (N', C', i', j')) \in Id \Longrightarrow$
    other-watched-wl N C i j $\leq \Downarrow Id$ (*other-watched-wl N' C' i' j'*)›
  for *N i j N' i' j'* **and** *C C'* :: ‹*'v literal*›
  by (*auto intro*!: *frefI nres-relI ASSERT-refine-right simp*: *mop-clauses-swap-def*
    op-clauses-swap-def)

show *1*: *?propa*
  (**is** ‹- $\leq \Downarrow$ *?unit* -›)
  supply *trail-keep-w*[*simp*]
  unfolding *unit-propagation-inner-loop-body-wl-int-alt-def*
    i-def[*symmetric*] *i-def'*[*symmetric*] *unit-propagation-inner-loop-body-l-with-skip-alt-def*
    unit-propagation-inner-loop-body-l-def
  unfolding *i-def*[*symmetric*] *SLw prod.case all-lits-alt-def2*[*symmetric*] *all-lits-def*[*symmetric*]
  apply (*rewrite* **in** ‹*if* (¬-) *then ASSERT* - >>= - *else* -› *if-not-swap*)
  supply *RETURN-as-SPEC-refine*[*refine2 del*]
  apply (*refine-rcg f*
     mop-polarity-wl-mop-polarity-l[**where** *b* = ‹*Some*(*L*, *Suc w*)›, *THEN fref-to-Down-curry*]
     mop-clauses-at-itself-spec pos-of-watched
     other-watched-wl-other-watched-l-spec-itself[*of* - - - - - - - - ‹*Some*(*L*, *Suc w*)›])
  subgoal using *inner-loop-inv w-le j-w blit-in-lit*
    unfolding *unit-propagation-inner-loop-wl-loop-pre-def* **by** *auto*
  subgoal
    using *j-w w-le S-S'*
    by (*cases S*; *cases S'*)
     (*auto simp*: *n state-wl-l-def unit-propagation-inner-loop-wl-loop-pre-def*
         unit-propagation-inner-loop-l-inv-def twl-st-l-def

simp flip: *Cons-nth-drop-Suc*)
subgoal by (*rule is-nondeleted-clause-pre*)
subgoal by *simp*
subgoal for *x' x1 x2 x1a x2a b Sa*
using *assms j-w w-le n-d blit-in-lit* **unfolding** *unit-prop-body-wl-inv-def mop-polarity-wl-def*
nres-monad3 unit-propagation-inner-loop-wl-loop-pre-def
using *S-S' w-le j-w n confl-S* **apply** −
apply *refine-vcg*
apply (*auto simp*: *keep-watch-state-wl assert-bind-spec-conv Let-def twl-st-wl*
Cons-nth-drop-Suc[*symmetric*] *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
blits-in-$\mathcal{L}_{in}$-def
corr-w correct-watching-except-correct-watching-except-Suc-notin
simp flip: *all-lits-alt-def2*
split: *if-splits dest*: *multi-member-split dest*!: *in-watched-keepD*)
done
apply (*rule keep-watch*; *assumption*+)
subgoal by *auto* — selection of K
subgoal by *fast*
subgoal by *auto*
subgoal by *auto* — If condition
subgoal for *x' x1 x2 x1a x2a b Sa*
using *j-w w-le S-S' blit-in-lit*
by (*auto simp*: *n keep-watch-state-wl assert-bind-spec-conv Let-def twl-st-wl*
Cons-nth-drop-Suc[*symmetric*] *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
blits-in-$\mathcal{L}_{in}$-def
corr-w correct-watching-except-correct-watching-except-Suc-notin
simp flip: *all-lits-def*
split: *if-splits dest*: *multi-member-split dest*!: *in-watched-keepD*)
subgoal for *x' x1 x2 x1a x2a b Sa X2 K x val-K v*
using *w-le j-w corr-w alien-L' S-S' blit-in-lit* **unfolding** *unit-prop-body-wl-inv-def*
all-lits-def[*symmetric*] *all-lits-alt-def*[*symmetric*] *all-lits-alt-def2*[*symmetric*]
apply (*intro conjI impI*)
apply (*auto simp*: *blits-in-$\mathcal{L}_{in}$-keep-watch2*)[*4*]
apply (*rule exI*[*of* - ‹*fst X2*›])
apply (*auto simp*: *remove-one-lit-from-wq-def*
correct-watching-except-correct-watching-except-Suc-Suc-keep-watch simp flip: *all-lits-def*)
done
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal by *simp*
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal by *fast*
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal by *simp* — polarity
subgoal for *x' x1 x2 x1a x2a b Sa X2 K x val-K v ia va L' vaa val-L' vaaa*
by (*rule update-blit-wl*)
subgoal using *S-S'* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal unfolding *unit-prop-body-wl-find-unwatched-inv-def* **by** *fast*
apply (*rule find-is-nat-rel*; *assumption*)
subgoal by *auto*
apply (*rule set-conflict-wl*; *assumption*)
subgoal for *x' x1 x2 x1a x2a b Sa X2 K x val-K v ia va L' vaa val-L' vaaa f x'a*
by *force*
apply (*rule propagate-lit-wl-general*; *assumption*)

subgoal
by *auto*
subgoal using *S-S′* **by** (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal by *fast*
subgoal by *auto*
subgoal for *x′ x1 x2 x1a x2a b Sa X2 K x val-K v ia va L′ vaa val-L′ vaaa f x′a xa*
using *S-S′* **by** (*cases* ‹*X2*›) (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal by *fast*
subgoal for *x′ x1 x2 x1a x2a b Sa X2 K x val-K v ia va L′ vaa val-L′ vaaa f x′a xa*
using *S-S′* **by** (*cases* ‹*X2*›) (*auto simp*: *eq-commute*[*of* - ‹- ! *w*›])
subgoal by *simp* — polarity
subgoal for *x′ x1 x2 x1a x2a b Sa X2 K x val-K v ia va L′ vaa val-L′ vaaa f x′a xa*
x′b Ka Kb val-L′a val-Ka
by (*rule update-blit*)
subgoal by (*rule update-clause-wl*)
done

have [*simp*]: ‹*add-mset a* (*remove1-mset a M*) = *M* ⟷ *a* ∈# *M*› **for** *a M*
by (*metis ab-semigroup-add-class.add.commute add.left-neutral multi-self-add-other-not-self*
remove1-mset-eqE union-mset-add-mset-left)

show *?eq* **if** *inv*: ‹*unit-propagation-inner-loop-body-l-inv L C′ T*›
using *i-le*[*OF inv*] *i-le2*[*OF inv*] *C′-dom*[*OF inv*] *S-S′*
unfolding *i-def*[*symmetric*]
by (*auto simp*: *ran-m-clause-upd image-mset-remove1-mset-if*)
qed

lemma
fixes *S* :: ‹*′v twl-st-wl*› **and** *S′* :: ‹*′v twl-st-l*› **and** *L* :: ‹*′v literal*› **and** *w* :: *nat*
defines [*simp*]: ‹*C′* ≡ *fst* (*watched-by S L* ! *w*)›
defines
[*simp*]: ‹*T* ≡ *remove-one-lit-from-wq C′ S′*›

defines
[*simp*]: ‹*C″* ≡ *get-clauses-l S′* ∝ *C′*›
assumes
S-S′: ‹(*S*, *S′*) ∈ *state-wl-l* (*Some* (*L*, *w*))› **and**
w-le: ‹*w* < *length* (*watched-by S L*)› **and**
j-w: ‹*j* ≤ *w*› **and**
corr-w: ‹*correct-watching-except j w L S*› **and**
blit-in-lit: ‹*blits-in-*$\mathcal{L}_{in}$ *S*› **and**
inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)› **and**
n: ‹*n* = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl S*)) (*mset* (*drop w* (*watched-by S L*))))›
and
confl-S: ‹*get-conflict-wl S* = *None*›
shows *unit-propagation-inner-loop-body-wl-spec*: ‹*unit-propagation-inner-loop-body-wl L j w S* ≤
⇓{((*i*, *j*, *T′*), (*T*, *n*)).
(*T′*, *T*) ∈ *state-wl-l* (*Some* (*L*, *j*)) ∧
correct-watching-except i j L T′ ∧ *blits-in-*$\mathcal{L}_{in}$ *T′* ∧
j ≤ *length* (*watched-by T′ L*) ∧
length (*watched-by S L*) = *length* (*watched-by T′ L*) ∧
i ≤ *j* ∧
(*get-conflict-wl T′* = *None* ⟶
n = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl T′*)) (*mset* (*drop j* (*watched-by T′ L*))))) ∧
(*get-conflict-wl T′* ≠ *None* ⟶ *n* = *0*)}

(*unit-propagation-inner-loop-body-l-with-skip L* (*S'*, *n*))›
apply (*rule order-trans*)
apply (*rule unit-propagation-inner-loop-body-wl-wl-int*[*OF S-S' w-le j-w corr-w inner-loop-inv n confl-S*])
apply (*subst Down-id-eq*)
apply (*rule unit-propagation-inner-loop-body-wl-int-spec*[*OF S-S' w-le j-w corr-w blit-in-lit inner-loop-inv n confl-S*])
done

definition *unit-propagation-inner-loop-wl-loop*
:: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ (*nat* × *nat* × '*v twl-st-wl*) *nres*› **where**
‹*unit-propagation-inner-loop-wl-loop L* S_0 = *do* {
ASSERT(*L* ∈# *all-lits-st* S_0);
let n = *length* (*watched-by* S_0 *L*);
$WHILE_T$*unit-propagation-inner-loop-wl-loop-inv L*
(λ(*j*, *w*, *S*). *w* < *n* ∧ *get-conflict-wl S* = *None*)
(λ(*j*, *w*, *S*). *do* {
unit-propagation-inner-loop-body-wl L j w S
})
(*0*, *0*, S_0)
}›

lemma *blits-in-*$\mathcal{L}_{in}$*-cut-watch*:
assumes *corr*: ‹*blits-in-*$\mathcal{L}_{in}$ (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
shows ‹*blits-in-*$\mathcal{L}_{in}$ (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*(*L* := *take j* (*g L*) @ *drop w* (*g L*)))›
using *assms* **by** (*auto simp*: *blits-in-*$\mathcal{L}_{in}$*-def dest*!: *in-set-takeD in-set-dropD*)

lemma *correct-watching-except-correct-watching-cut-watch*:
assumes *corr*: ‹*correct-watching-except j w L* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
shows ‹*correct-watching* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*(*L* := *take j* (*g L*) @ *drop w* (*g L*)))›
proof −
let *?*$\mathcal{L}$ = ‹*all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*(*L* := *take j* (*g L*) @ *drop w* (*g L*)))›
have $\mathcal{L}$: ‹*?*$\mathcal{L}$ = *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
by (*auto simp*: *all-lits-st-def*)
have
Heq:
‹⋀*La i K b'*. *La* ∈# *?*$\mathcal{L}$ ⟹
(*La* = *L* ⟶
distinct-watched (*take j* (*g La*) @ *drop w* (*g La*)) ∧
((*i*, *K*, *b'*)∈#*mset* (*take j* (*g La*) @ *drop w* (*g La*)) ⟶
i ∈# *dom-m b* ⟶ *K* ∈ *set* (*b* ∝ *i*) ∧ *K* ≠ *La* ∧ *correctly-marked-as-binary b* (*i*, *K*, *b'*)) ∧
((*i*, *K*, *b'*)∈#*mset* (*take j* (*g La*) @ *drop w* (*g La*)) ⟶
b' ⟶ *i* ∈# *dom-m b*) ∧
{#*i* ∈# *fst* ‘# *mset* (*take j* (*g La*) @ *drop w* (*g La*)). *i* ∈# *dom-m b*#} =
clause-to-update La (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))› **and**
Hneq:
‹⋀*La i K b'*. *La*∈#*?*$\mathcal{L}$ ⟹
(*La* ≠ *L* ⟶
distinct-watched (*g La*) ∧

```
      ((i, K, b')∈#mset (g La) ⟶ i ∈# dom-m b ⟶ K ∈ set (b ∝ i) ∧ K ≠ La
        ∧ correctly-marked-as-binary b (i, K, b')) ∧
       ((i, K, b')∈#mset (g La) ⟶ b' ⟶ i ∈# dom-m b) ∧
      {#i ∈# fst `# mset (g La). i ∈# dom-m b#} =
      clause-to-update La (a, b, c, d, e, NEk, UEk, NS, US, N0, U0, {#}, {#}))›
   using corr
   unfolding correct-watching.simps correct-watching-except.simps ℒ
   by fast+
 have
   ‹((i, K, b')∈#mset ((g(L := take j (g L) @ drop w (g L))) La) ⟹
       i ∈# dom-m b ⟶ K ∈ set (b ∝ i) ∧ K ≠ La ∧ correctly-marked-as-binary b (i, K, b'))› and
   ‹(i, K, b')∈#mset ((g(L := take j (g L) @ drop w (g L))) La) ⟹
       b' ⟶ i ∈# dom-m b› and
   ‹{#i ∈# fst `# mset ((g(L := take j (g L) @ drop w (g L))) La).
       i ∈# dom-m b#} =
     clause-to-update La (a, b, c, d, e, NEk, UEk, NS, US, N0, U0, {#}, {#})›and
   ‹distinct-watched ((g(L := take j (g L) @ drop w (g L))) La)›
 if ‹La∈#?ℒ›
 for La i K b'
   apply (cases ‹La = L›)
   subgoal
     using Heq[of La i K] that by auto
   subgoal
     using Hneq[of La i K] that by auto
   apply (cases ‹La = L›)
   subgoal
     using Heq[of La i K] that by auto
   subgoal
     using Hneq[of La i K] that by auto
   apply (cases ‹La = L›)
   subgoal
     using Heq[of La i K] that by auto
   subgoal
     using Hneq[of La i K] that by auto
   apply (cases ‹La = L›)
   subgoal
     using Heq[of La i K] that by auto
   subgoal
     using Hneq[of La i K] that by auto
   done
 then show ?thesis
   unfolding correct-watching.simps
   by blast
qed

lemma unit-propagation-inner-loop-wl-loop-alt-def:
 ‹unit-propagation-inner-loop-wl-loop L S₀ = do {
   ASSERT(L ∈# all-lits-st S₀);
   let (- :: nat) = (if get-conflict-wl S₀ = None then remaining-nondom-wl 0 L S₀ else 0);
   let n = length (watched-by S₀ L);
   WHILE_T^unit-propagation-inner-loop-wl-loop-inv L
     (λ(j, w, S). w < n ∧ get-conflict-wl S = None)
     (λ(j, w, S). do {
       unit-propagation-inner-loop-body-wl L j w S
     })
     (0, 0, S₀)
```

```
  }
  ›
  unfolding unit-propagation-inner-loop-wl-loop-def Let-def by auto

definition cut-watch-list :: ‹nat ⇒ nat ⇒ 'v literal ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
  ‹cut-watch-list j w L =(λ(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). do {
      ASSERT(j ≤ w ∧ j ≤ length (W L) ∧ w ≤ length (W L) ∧ L ∈# all-lits-st (M, N, D, NE, UE,
NEk, UEk, NS, US, N0, U0, Q, W));
      RETURN (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(L := take j (W L) @ drop w
(W L)))
    })›

definition unit-propagation-inner-loop-wl :: ‹'v literal ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
  ‹unit-propagation-inner-loop-wl L S₀ = do {
    (j, w, S) ← unit-propagation-inner-loop-wl-loop L S₀;
    ASSERT(j ≤ w ∧ w ≤ length (watched-by S L) ∧ L ∈# all-lits-st S);
    cut-watch-list j w L S
  }›

lemma correct-watching-correct-watching-except00:
  ‹correct-watching S ⟹ correct-watching-except 0 0 L S›
  apply (cases S)
  apply (simp only: correct-watching.simps correct-watching-except.simps
    take0 drop0 append.left-neutral)
  by fast

lemma unit-propagation-inner-loop-wl-spec:
  shows ‹(uncurry unit-propagation-inner-loop-wl, uncurry unit-propagation-inner-loop-l) ∈
    {((L', T'::'v twl-st-wl), (L, T::'v twl-st-l)). L = L' ∧ (T', T) ∈ state-wl-l (Some (L, 0)) ∧
      correct-watching T' ∧ blits-in-ℒ_in T'} →
    ⟨{(T', T). (T', T) ∈ state-wl-l None ∧ correct-watching T' ∧ blits-in-ℒ_in T'}⟩ nres-rel
    › (is ‹?fg ∈ ?A → ⟨?B⟩nres-rel› is ‹?fg ∈ ?A → ⟨{(T', T). - ∧ ?P T T'}⟩nres-rel›)
proof −
  {
    fix L :: ‹'v literal› and S :: ‹'v twl-st-wl› and S' :: ‹'v twl-st-l›
    assume
      corr-w: ‹correct-watching S› ‹blits-in-ℒ_in S› and
      SS': ‹(S, S') ∈ state-wl-l (Some (L, 0))›
```

To ease the finding the correspondence between the body of the loops, we introduce following function:

```
    let ?R' = ‹{((i, j, T'), (T, n)).
        (T', T) ∈ state-wl-l (Some (L, j)) ∧
        correct-watching-except i j L T' ∧ blits-in-ℒ_in T' ∧
        j ≤ length (watched-by T' L) ∧
        length (watched-by S L) = length (watched-by T' L) ∧
        i ≤ j ∧
        (get-conflict-wl T' = None ⟶
            n = size (filter-mset (λ(i, -). i ∉# dom-m (get-clauses-wl T')) (mset (drop j (watched-by T'
L))))) ∧
        (get-conflict-wl T' ≠ None ⟶ n = 0) ∧
        blits-in-ℒ_in T'}›
    have inv: ‹unit-propagation-inner-loop-wl-loop-inv L iT'›
      if
        iT'-Tn: ‹(iT', Tn) ∈ ?R'› and
        ‹unit-propagation-inner-loop-l-inv L Tn›
```

```
    for Tn iT'
  proof −
    obtain i j :: nat and T' where iT': ‹iT' = (i, j, T')› by (cases iT')
    obtain T n where Tn[simp]: ‹Tn = (T, n)› by (cases Tn)
    have ‹unit-propagation-inner-loop-l-inv L (T, 0::nat)›
      if ‹unit-propagation-inner-loop-l-inv L (T, n)› and ‹get-conflict-l T ≠ None›
      using that iT'-Tn
      unfolding unit-propagation-inner-loop-l-inv-def iT' prod.case
      apply − apply normalize-goal+
      apply (rule-tac x=x in exI)
      by auto
    then show ?thesis
      unfolding unit-propagation-inner-loop-wl-loop-inv-def iT' prod.simps apply −
      apply (rule exI[of - T])
      using that by (auto simp: iT')
  qed
  have cond: ‹(j < length (watched-by S L) ∧ get-conflict-wl T' = None) =
    (clauses-to-update-l T ≠ {#} ∨ n > 0)›
    if
      iT'-T: ‹(ijT', Tn) ∈ ?R'› and
      [simp]: ‹ijT' = (i, jT')› ‹jT' = (j, T')›  ‹Tn = (T, n)›
      for ijT' Tn i j T' n T jT'
  proof −
    have [simp]: ‹size {#(i, -) ∈# mset (drop j xs). i ∉# dom-m b#} =
      size {#i ∈# fst `# mset (drop j xs). i ∉# dom-m b#}› for xs b
      apply (induction ‹xs› arbitrary: j)
      subgoal by auto
      subgoal premises p for a xs j
        using p[of 0] p
        by (cases j) auto
      done
    have [simp]: ‹size (filter-mset (λi. (i ∈# (dom-m b))) (fst `# (mset (drop j (g L))))) +
        size {#i ∈# fst `# mset (drop j (g L)). i ∉# dom-m b#} =
        length (g L) − j› for g j b
      apply (subst size-union[symmetric])
      apply (subst multiset-partition[symmetric])
      by auto
    have [simp]: ‹A ≠ {#} ⟹ size A > 0› for A
      by (auto dest!: multi-member-split)
    have ‹length (watched-by T' L) = size (clauses-to-update-wl T' L j) + n + j›
      if ‹get-conflict-wl T' = None›
      using that iT'-T
      by (cases ‹get-conflict-wl T'›; cases T')
        (auto simp add: state-wl-l-def drop-map)
    then show ?thesis
      using iT'-T
      by (cases ‹get-conflict-wl T' = None›) auto
  qed
  have remaining: ‹RETURN (if get-conflict-wl S = None then remaining-nondom-wl 0 L S else 0)
≤ SPEC (λ-. True)›
    by auto

  have [intro]: ‹unit-propagation-inner-loop-wl-loop-inv L
      (j, w, T) ⟹ L ∈# all-lits-st T› for j w T
    unfolding unit-propagation-inner-loop-wl-loop-inv-def
      unit-propagation-inner-loop-l-inv-def prod.case
```

```
  apply normalize-goal+
  apply (drule twl-struct-invs-no-alien-in-trail[of - ‹−L›])
  apply (simp-all only: in-all-lits-of-mm-uminus-iff twl-st-wl twl-st-l twl-st
    all-lits-def multiset.map-comp comp-def clause-twl-clause-of ac-simps
    in-all-lits-uminus-iff)
 done
have unit-propagation-inner-loop-l-alt-def: ‹unit-propagation-inner-loop-l L S' = do {
   ASSERT(L ∈# all-lits-of-st-l S');
   n ← SPEC (λ-::nat. True);
   (S, n) ← WHILE_T^unit-propagation-inner-loop-l-inv L
       (λ(S, n). clauses-to-update-l S ≠ {#} ∨ 0 < n)
       (unit-propagation-inner-loop-body-l-with-skip L) (S', n);
   RETURN S}› for L S'
 unfolding unit-propagation-inner-loop-l-def by auto
have unit-propagation-inner-loop-wl-alt-def: ‹unit-propagation-inner-loop-wl L S = do {
 ASSERT(L ∈# all-lits-st S);
 let (n::nat) = (if get-conflict-wl S = None then remaining-nondom-wl 0 L S else 0);
 (j, w, S) ← WHILE_T^unit-propagation-inner-loop-wl-loop-inv L
   (λ(j, w, T). w < length (watched-by S L) ∧ get-conflict-wl T = None)
   (λ(j, x, y). unit-propagation-inner-loop-body-wl L j x y) (0, 0, S);
 ASSERT (j ≤ w ∧ w ≤ length (watched-by S L) ∧ L ∈# all-lits-st S);
 cut-watch-list j w L S}›
 unfolding unit-propagation-inner-loop-wl-loop-alt-def unit-propagation-inner-loop-wl-def
 by auto
have ‹unit-propagation-inner-loop-wl L S ≤
      ⇓ {((T'), T). (T', T) ∈ state-wl-l None ∧ ?P T T'}
        (unit-propagation-inner-loop-l L S')›
 (is ‹- ≤ ⇓ ?R -›)
 unfolding unit-propagation-inner-loop-l-alt-def uncurry-def
   unit-propagation-inner-loop-wl-alt-def
 apply (refine-vcg WHILEIT-refine-genR[where
      R' = ‹?R'› and
      R = ‹{((i, j, T'), (T, n)). ((i, j, T'), (T, n)) ∈ ?R' ∧ i ≤ j ∧
         length (watched-by S L) = length (watched-by T' L) ∧
        (j ≥ length (watched-by T' L) ∨ get-conflict-wl T' ≠ None) ∧
        unit-propagation-inner-loop-wl-loop-inv L (i, j, T')}›]
    remaining)
 subgoal using SS' by (auto simp: all-lits-def ac-simps)
 subgoal using corr-w SS' by (auto simp: correct-watching-correct-watching-except00)
 subgoal by (rule inv)
 subgoal by (rule cond)
 subgoal for n i'w'T' Tn i' w'T' w' T'
   apply (cases Tn)
   apply (rule order-trans)
   apply (rule unit-propagation-inner-loop-body-wl-spec[of - ‹fst Tn›])
   apply (simp only: prod.case in-pair-collect-simp)
   apply normalize-goal+
   apply (auto simp del: twl-st-of-wl.simps intro!: conc-fun-R-mono)
   done
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal by auto
 subgoal for n i'w'T' Tn i' w'T' j L' w' T'
   apply (cases T')
```

by (*auto simp*: *state-wl-l-def cut-watch-list-def all-lits-def intro*: *blits-in-$\mathcal{L}_{in}$-cut-watch*
dest!: *correct-watching-except-correct-watching-cut-watch*)
done
}
note H = *this*

show *?thesis*
unfolding *fref-param1*
apply (*intro frefI nres-relI*)
by (*auto simp*: *intro*!: H)
qed

6.5.2 Outer loop

definition *select-and-remove-from-literals-to-update-wl* :: ‹*'v twl-st-wl* ⇒ (*'v twl-st-wl* × *'v literal*) *nres*›
where
‹*select-and-remove-from-literals-to-update-wl S* = *SPEC*(λ(*S'*, *L*). *L* ∈# *literals-to-update-wl S* ∧
S' = *set-literals-to-update-wl* (*literals-to-update-wl S* − {#*L*#}) *S*)›

definition *unit-propagation-outer-loop-wl-inv* **where**
‹*unit-propagation-outer-loop-wl-inv S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧
correct-watching S ∧ *blits-in-$\mathcal{L}_{in}$ S* ∧
unit-propagation-outer-loop-l-inv S')›

definition *unit-propagation-outer-loop-wl* :: ‹*'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*unit-propagation-outer-loop-wl* S_0 =
WHILE$_T$*unit-propagation-outer-loop-wl-inv*
(λ*S*. *literals-to-update-wl S* ≠ {#})
(λ*S*. *do* {
ASSERT(*literals-to-update-wl S* ≠ {#});
(*S'*, *L*) ← *select-and-remove-from-literals-to-update-wl S*;
ASSERT(*L* ∈# *all-lits-st S'*);
unit-propagation-inner-loop-wl L S'
})
(S_0 :: *'v twl-st-wl*)
›

lemma *blits-in-$\mathcal{L}_{in}$-simps*[*simp*]: ‹*blits-in-$\mathcal{L}_{in}$* (*set-literals-to-update-wl C xa*) ⟷ *blits-in-$\mathcal{L}_{in}$ xa*›
by (*cases xa*; *auto simp*: *blits-in-$\mathcal{L}_{in}$-def*)

lemma *unit-propagation-outer-loop-wl-spec*:
‹(*unit-propagation-outer-loop-wl*, *unit-propagation-outer-loop-l*)
∈ {(*T'*::*'v twl-st-wl*, *T*).
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T' ∧ *blits-in-$\mathcal{L}_{in}$ T'*} →$_f$
⟨{(*T'*, *T*).
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T' ∧ *blits-in-$\mathcal{L}_{in}$ T'*}⟩*nres-rel*›
(**is** ‹*?u* ∈ *?A* →$_f$ ⟨*?B*⟩ *nres-rel*›)
proof −
have *inv*: ‹*unit-propagation-outer-loop-wl-inv T'*›
if
‹(*T'*, *T*) ∈ {(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'* ∧ *blits-in-$\mathcal{L}_{in}$ T'*}› **and**
‹*unit-propagation-outer-loop-l-inv T*›
for *T T'*

```
  unfolding unit-propagation-outer-loop-wl-inv-def
  apply (rule exI[of - T])
  using that by auto

  have select-and-remove-from-literals-to-update-wl:
   ‹select-and-remove-from-literals-to-update-wl S' ≤
     ⇓ {((T', L'), (T, L)). L = L' ∧ (T', T) ∈ state-wl-l (Some (L, 0)) ∧
        T' = set-literals-to-update-wl (literals-to-update-wl S' - {#L#}) S' ∧ L ∈# literals-to-update-wl
S' ∧
         L ∈# all-lits-st S'
       }
       (select-and-remove-from-literals-to-update S)›
    if S: ‹(S', S) ∈ state-wl-l None› and ‹get-conflict-wl S' = None› and
      corr-w: ‹correct-watching S'› and
      inv-l: ‹unit-propagation-outer-loop-l-inv S›
    for S :: ‹'v twl-st-l› and S' :: ‹'v twl-st-wl›
  proof -
    obtain M N D NE UE NEk UEk NS US N0 U0 W Q where
      S': ‹S' = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)›
      by (cases S') auto
    obtain R where
      S-R: ‹(S, R) ∈ twl-st-l None› and
      struct-invs: ‹twl-struct-invs R›
      using inv-l unfolding unit-propagation-outer-loop-l-inv-def by blast
    have [simp]:
      ‹init-clss (state_W-of R) = mset `# (init-clss-lf N) + NE + NEk + NS + N0›
      ‹atm-of ` lits-of-l (get-trail R) = atm-of ` lits-of-l M›
      using S-R S by (auto simp: twl-st S' twl-st-wl simp flip: state_W-of-def)
    have
      no-dup-q: ‹no-duplicate-queued R› and
      alien: ‹cdcl_W-restart-mset.no-strange-atm (state_W-of R)›
      using struct-invs that by (auto simp: twl-struct-invs-def
          cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
        pcdcl-all-struct-invs-def state_W-of-def)
    then have H1: ‹L ∈# all-lits-of-mm (mset `# ran-mf N + (NE + UE) + (NEk + UEk) + (NS +
US) + (N0 + U0))›
      if LQ: ‹L ∈# Q› for L
    proof -
      have [simp]: ‹(f o g) ` I = f ` g ` I› for f g I
        by auto
      obtain K where ‹L = - lit-of K› and ‹K ∈# mset (trail (state_W-of R))›
        using that no-dup-q LQ S-R S
        mset-le-add-mset-decr-left2[of L ‹remove1-mset L Q› Q]
        by (fastforce simp: S' cdcl_W-restart-mset.no-strange-atm-def cdcl_W-restart-mset-state
          all-lits-of-mm-def atms-of-ms-def twl-st-l-def state-wl-l-def uminus-lit-swap
          convert-lit.simps
          dest!: multi-member-split[of L Q] mset-subset-eq-insertD in-convert-lits-lD2)
      from imageI[OF this(2), of ‹atm-of o lit-of›]
      have ‹atm-of L ∈ atm-of ` lits-of-l (get-trail-wl S')› and
        [simp]: ‹atm-of ` lits-of-l (trail (state_W-of R)) = atm-of ` lits-of-l (get-trail-wl S')›
        using S-R S S ‹L = - lit-of K›
        by (simp-all add: twl-st image-image[symmetric]
            lits-of-def[symmetric])
      then have ‹atm-of L ∈ atm-of ` lits-of-l M›
        using S' by auto
      moreover {
```

```
      have ‹atm-of ‘ lits-of-l M
       ⊆ (⋃x∈set-mset (init-clss-lf N). atm-of ‘ set x) ∪
         (⋃x∈set-mset NE. atms-of x) ∪
         (⋃x∈set-mset NEk. atms-of x) ∪
         (⋃x∈set-mset NS. atms-of x) ∪
         (⋃x∈set-mset N0. atms-of x)›
        using that alien unfolding cdcl_W-restart-mset.no-strange-atm-def
        by (auto simp: S' cdcl_W-restart-mset.no-strange-atm-def cdcl_W-restart-mset-state
            all-lits-of-mm-def atms-of-ms-def simp flip: state_W-of-def)
      then have ‹atm-of ‘ lits-of-l M ⊆ (⋃x∈set-mset (init-clss-lf N). atm-of ‘ set x) ∪
       (⋃x∈set-mset NE. atms-of x) ∪
       (⋃x∈set-mset NEk. atms-of x) ∪
       (⋃x∈set-mset NS. atms-of x) ∪
       (⋃x∈set-mset N0. atms-of x)›
      unfolding image-Un[symmetric]
        set-append[symmetric]
        append-take-drop-id
        .
       then have ‹atm-of ‘ lits-of-l M ⊆ atms-of-mm (mset ‘# init-clss-lf N + NE + NEk + NS +
N0)›
        by (smt UN-Un Un-iff append-take-drop-id atms-of-ms-def atms-of-ms-mset-unfold set-append
            set-image-mset set-mset-mset set-mset-union subset-eq)
    }
    ultimately have ‹atm-of L ∈ atms-of-mm (mset ‘# ran-mf N + NE + NEk + NS + N0)›
      using that
      unfolding all-lits-of-mm-union atms-of-ms-union all-clss-lf-ran-m[symmetric]
        image-mset-union set-mset-union
      by auto
    then show ?thesis
      using that by (auto simp: in-all-lits-of-mm-ain-atms-of-iff)
  qed
  have H: ‹clause-to-update L S = {#i ∈# fst ‘# mset (W L). i ∈# dom-m N#}›
    ‹L ∈# all-lits-st ([], N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, W)›
    if ‹L ∈# Q› for L
    using corr-w that S H1[OF that] by (auto simp: correct-watching.simps S' clause-to-update-def
      Ball-def ac-simps all-conj-distrib all-lits-st-def all-lits-def
      dest!: multi-member-split)
  show ?thesis
  unfolding select-and-remove-from-literals-to-update-wl-def select-and-remove-from-literals-to-update-def
    apply (rule RES-refine)
    unfolding Bex-def
    by (rule-tac x=‹(set-clauses-to-update-l (clause-to-update (snd s) S)
          (set-literals-to-update-l
            (remove1-mset (snd s) (literals-to-update-l S)) S), snd s)› in exI)
       (use that S' S in ‹auto simp: correct-watching.simps clauses-def state-wl-l-def
        mset-take-mset-drop-mset' cdcl_W-restart-mset-state all-lits-of-mm-union
        dest: H1 H›)
qed
have conflict-None: ‹get-conflict-wl T = None›
  if
    ‹literals-to-update-wl T ≠ {#}› and
    inv1: ‹unit-propagation-outer-loop-wl-inv T›
    for T
proof −
  obtain T' where
    2: ‹(T, T') ∈ state-wl-l None› and
```

inv2: ‹*unit-propagation-outer-loop-l-inv* T'›
using *inv1* **unfolding** *unit-propagation-outer-loop-wl-inv-def* **by** *blast*
obtain T'' **where**
3: ‹$(T', T'') \in$ *twl-st-l None*› **and**
‹*twl-struct-invs* T''›
using *inv2* **unfolding** *unit-propagation-outer-loop-l-inv-def* **by** *blast*
then have ‹*get-conflict* $T'' \neq$ *None* $\longrightarrow$
clauses-to-update $T'' = \{\#\} \wedge$ *literals-to-update* $T'' = \{\#\}$›
unfolding *twl-struct-invs-def* **by** *fast*
then show *?thesis*
using *that 2 3* **by** (*auto simp*: *twl-st-wl twl-st twl-st-l*)
qed
show *?thesis*
unfolding *unit-propagation-outer-loop-wl-def unit-propagation-outer-loop-l-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg select-and-remove-from-literals-to-update-wl*
unit-propagation-inner-loop-wl-spec[*unfolded fref-param1*, *THEN fref-to-Down-curry*])
subgoal by (*rule inv*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule conflict-None*)
subgoal for T' T **by** (*auto simp*:)
subgoal by (*auto simp*: *twl-st-wl ac-simps*)
subgoal by *auto*
done
qed

6.5.3 Decide or Skip

definition *find-unassigned-lit-wl* :: ‹$'v$ *twl-st-wl* $\Rightarrow$ ($'v$ *twl-st-wl* $\times$ $'v$ *literal option*) *nres*› **where**
‹*find-unassigned-lit-wl* = (λS.
SPEC ($\lambda(T, L)$. $T = S \wedge$
($L \neq$ *None* $\longrightarrow$
undefined-lit (*get-trail-wl* S) (*the* L) $\wedge$
the $L \in\#$ *all-lits-st* S) $\wedge$
($L =$ *None* $\longrightarrow$ ($\nexists L'$. *undefined-lit* (*get-trail-wl* S) $L' \wedge$
$L' \in\#$ *all-lits-st* S)))
)›

definition *decide-wl-or-skip-pre* **where**
‹*decide-wl-or-skip-pre* $S \longleftrightarrow$
($\exists S'$. $(S, S') \in$ *state-wl-l None* $\wedge$
decide-l-or-skip-pre $S' \wedge$ *blits-in-*$\mathcal{L}_{in}$ S
)›

definition *decide-lit-wl* :: ‹$'v$ *literal* $\Rightarrow$ $'v$ *twl-st-wl* $\Rightarrow$ $'v$ *twl-st-wl*› **where**
‹*decide-lit-wl* = ($\lambda L'$ (M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, Q, W).
(*Decided* L' # M, N, D, NE, UE, NEk, UEk, NS, US, $N0$, $U0$, $\{\#- L'\#\}$, W))›

definition *decide-wl-or-skip* :: ‹$'v$ *twl-st-wl* $\Rightarrow$ (*bool* $\times$ $'v$ *twl-st-wl*) *nres*› **where**
‹*decide-wl-or-skip* S = (*do* {
ASSERT(*decide-wl-or-skip-pre* S);
(S, L) $\leftarrow$ *find-unassigned-lit-wl* S;
case L *of*
None $\Rightarrow$ *RETURN* (*True*, S)

```
    | Some L ⇒ RETURN (False, decide-lit-wl L S)
  })
›

lemma decide-wl-or-skip-spec:
  ‹(decide-wl-or-skip, decide-l-or-skip)
    ∈ {(T'':: 'v twl-st-wl, T).
         (T', T) ∈ state-wl-l None ∧
         correct-watching T' ∧ blits-in-ℒ_in T' ∧
         get-conflict-wl T' = None} →
        ⟨{((b', T'), (b, T)). b' = b ∧
         (T', T) ∈ state-wl-l None ∧
         correct-watching T' ∧ blits-in-ℒ_in T'}⟩nres-rel›
proof −
  have find-unassigned-lit-wl: ‹find-unassigned-lit-wl S'
    ≤ ⇓ {((T, L), L'). T = S' ∧ L = L'}
        (find-unassigned-lit-l S)›
    if ‹(S', S) ∈ state-wl-l None›
    for S :: ‹'v twl-st-l› and S' :: ‹'v twl-st-wl›
    using that
    by (auto simp: find-unassigned-lit-l-def find-unassigned-lit-wl-def
      intro!: RES-refine)

  have option: ‹(x, x') ∈ ⟨Id⟩option-rel› if ‹x = x'› for x x'
    using that by (auto)
  show ?thesis
    unfolding decide-wl-or-skip-def decide-l-or-skip-def
    apply (refine-vcg find-unassigned-lit-wl option)
    subgoal unfolding decide-wl-or-skip-pre-def by fast
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal for S S'
      by (cases S) (auto simp: correct-watching.simps clause-to-update-def
          decide-lit-l-def decide-lit-wl-def state-wl-l-def blits-in-ℒ_in-def)
    done
qed
```

6.5.4 Skip or Resolve

```
definition mop-tl-state-wl-pre :: ‹'v twl-st-wl ⇒ bool› where
‹mop-tl-state-wl-pre S ⟷
  (∃ S'. (S, S') ∈ state-wl-l None ∧ correct-watching S ∧ mop-tl-state-l-pre S')›

definition tl-state-wl :: ‹'v twl-st-wl ⇒ 'v twl-st-wl› where
  ‹tl-state-wl = (λ(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q).
    (tl M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q))›

definition mop-tl-state-wl :: ‹'v twl-st-wl ⇒ (bool × 'v twl-st-wl) nres› where
  ‹mop-tl-state-wl = (λS. do {
    ASSERT(mop-tl-state-wl-pre S);
    RETURN(False, tl-state-wl S)})›

definition resolve-cls-wl' :: ‹'v twl-st-wl ⇒ nat ⇒ 'v literal ⇒ 'v clause› where
‹resolve-cls-wl' S C L =
   remove1-mset L (remove1-mset (−L) (the (get-conflict-wl S) ∪# (mset (get-clauses-wl S ∝ C))))›
```

definition *update-confl-tl-wl* :: ‹'v literal ⇒ nat ⇒ 'v twl-st-wl ⇒ bool × 'v twl-st-wl› **where**
‹*update-confl-tl-wl* = (λL C (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q).
let D = *resolve-cls-wl'* (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q) C L *in*
(*False*, (*tl M*, N, *Some D*, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q)))›

definition *update-confl-tl-wl-pre* :: ‹'v literal ⇒ nat ⇒ 'v twl-st-wl ⇒ bool› **where**
‹*update-confl-tl-wl-pre L C S* ⟷
(∃ *S'*. (S, *S'*) ∈ *state-wl-l None* ∧ *update-confl-tl-l-pre L C S'* ∧
correct-watching S ∧ *blits-in-*$\mathcal{L}_{in}$ *S*)›

definition *mop-update-confl-tl-wl* :: ‹'v literal ⇒ nat ⇒ 'v twl-st-wl ⇒ (bool × 'v twl-st-wl) nres› **where**
‹*mop-update-confl-tl-wl* = (λL C S. *do* {
ASSERT(*update-confl-tl-wl-pre L C S*);
RETURN (*update-confl-tl-wl L C S*)
})›

definition *mop-hd-trail-wl-pre* :: ‹'v twl-st-wl ⇒ bool› **where**
‹*mop-hd-trail-wl-pre S* ⟷
(∃ *S'*. (S, *S'*) ∈ *state-wl-l None* ∧ *mop-hd-trail-l-pre S'* ∧
correct-watching S)›

definition *mop-hd-trail-wl* :: ‹'v twl-st-wl ⇒ ('v literal × nat) nres› **where**
‹*mop-hd-trail-wl S* = *do*{
ASSERT(*mop-hd-trail-wl-pre S*);
SPEC(λ(L, C). *Propagated L C* = *hd* (*get-trail-wl S*))
}›

definition *skip-and-resolve-loop-wl-inv* :: ‹'v twl-st-wl ⇒ bool ⇒ 'v twl-st-wl ⇒ bool› **where**
‹*skip-and-resolve-loop-wl-inv* S_0 *brk S* ⟷
(∃ *S' S'*$_0$. (S, *S'*) ∈ *state-wl-l None* ∧
(S_0, *S'*$_0$) ∈ *state-wl-l None* ∧
skip-and-resolve-loop-inv-l S'$_0$ *brk S'* ∧
correct-watching S)›

definition *mop-lit-notin-conflict-wl* :: ‹'v literal ⇒ 'v twl-st-wl ⇒ bool nres› **where**
‹*mop-lit-notin-conflict-wl L S* = *do* {
ASSERT(*get-conflict-wl S* ≠ *None* ∧ −*L* ∉# *the* (*get-conflict-wl S*) ∧ *L* ∈# *all-lits-st S*);
RETURN (*L* ∉# *the* (*get-conflict-wl S*))
}›

definition *mop-maximum-level-removed-wl-pre* :: ‹'v literal ⇒ 'v twl-st-wl ⇒ bool› **where**
‹*mop-maximum-level-removed-wl-pre L S* ⟷
(∃ *S'*. (S, *S'*) ∈ *state-wl-l None* ∧ *mop-maximum-level-removed-l-pre L S'* ∧
correct-watching S)›

definition *mop-maximum-level-removed-wl* :: ‹'v literal ⇒ 'v twl-st-wl ⇒ bool nres› **where**
‹*mop-maximum-level-removed-wl L S* = *do* {
ASSERT (*mop-maximum-level-removed-wl-pre L S*);
RETURN (*get-maximum-level* (*get-trail-wl S*) (*remove1-mset* (−*L*) (*the* (*get-conflict-wl S*))) =
count-decided (*get-trail-wl S*))
}›

definition *skip-and-resolve-loop-wl* :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹*skip-and-resolve-loop-wl* S_0 =

```
  do {
    ASSERT(get-conflict-wl S_0 ≠ None);
    (-, S) ←
      WHILE_T^λ(brk, S). skip-and-resolve-loop-wl-inv S_0 brk S
      (λ(brk, S). ¬brk ∧ ¬is-decided (hd (get-trail-wl S)))
      (λ(-, S).
        do {
          (L, C) ← mop-hd-trail-wl S;
          b ← mop-lit-notin-conflict-wl (−L) S;
          if b then
            mop-tl-state-wl S
          else do {
            b ← mop-maximum-level-removed-wl L S;
            if b
            then
              mop-update-confl-tl-wl L C S
            else
              do {RETURN (True, S)}
          }
        }
      )
      (False, S_0);
    RETURN S
  }
›
```

lemma *tl-state-wl-tl-state-l*:
‹(*S*, *S'*) ∈ *state-wl-l None* ⟹ *correct-watching S* ⟹ *blits-in-$\mathcal{L}_{in}$ S* ⟹
mop-tl-state-wl S ≤ ⇓(*bool-rel* ×$_f$ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*})
(*mop-tl-state-l S'*)›
by (*cases S*) (*auto simp*: *state-wl-l-def tl-state-wl-def tl-state-l-def blits-in-$\mathcal{L}_{in}$-def*
mop-tl-state-wl-def mop-tl-state-l-def mop-tl-state-wl-pre-def
assert-bind-spec-conv correct-watching.simps clause-to-update-def
intro!: *ASSERT-refine-right ASSERT-leI*)

lemma *mop-update-confl-tl-wl-mop-update-confl-tl-l*:
‹(*S*, *S'*) ∈ *state-wl-l None* ⟹ *correct-watching S* ⟹ *blits-in-$\mathcal{L}_{in}$ S* ⟹
((*L*, *C*), (*L'*, *C'*)) ∈ *Id* ⟹
mop-update-confl-tl-wl L C S ≤
⇓(*bool-rel* ×$_f$ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*})
(*mop-update-confl-tl-l L' C' S'*)›
by (*cases S*) (*auto simp*: *state-wl-l-def tl-state-wl-def tl-state-l-def blits-in-$\mathcal{L}_{in}$-def*
mop-update-confl-tl-wl-def mop-update-confl-tl-l-def
update-confl-tl-wl-def update-confl-tl-l-def resolve-cls-l'-def
resolve-cls-wl'-def update-confl-tl-wl-pre-def
assert-bind-spec-conv correct-watching.simps clause-to-update-def
intro!: *ASSERT-refine-right ASSERT-leI*)

lemma *mop-hd-trail-wl-mop-hd-trail-l*:
‹(*S*, *S'*) ∈ *state-wl-l None* ⟹ *correct-watching S* ⟹ *blits-in-$\mathcal{L}_{in}$ S* ⟹
mop-hd-trail-wl S
≤ ⇓ (*Id*)
(*mop-hd-trail-l S'*)›
unfolding *mop-hd-trail-wl-def mop-hd-trail-l-def*

apply *refine-rcg*
subgoal unfolding *mop-hd-trail-wl-pre-def* **by** *blast*
subgoal by *auto*
done

lemma *mop-lit-notin-conflict-wl-mop-lit-notin-conflict-l*:
‹$(S,\ S') \in$ *state-wl-l None* $\Longrightarrow$ *correct-watching* $S \Longrightarrow$ *blits-in-*$\mathcal{L}_{in}$ $S \Longrightarrow$
$L = L' \Longrightarrow$
mop-lit-notin-conflict-wl $L\ S \leq \Downarrow$*bool-rel* (*mop-lit-notin-conflict-l* $L'\ S'$)›
unfolding *mop-lit-notin-conflict-wl-def mop-lit-notin-conflict-l-def*
apply *refine-rcg*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *all-lits-def* **by** (*auto simp*: *ac-simps*)
subgoal by *auto*
done

lemma *mop-maximum-level-removed-wl-mop-maximum-level-removed-l*:
‹$(S,\ S') \in$ *state-wl-l None* $\Longrightarrow$ *correct-watching* $S \Longrightarrow$ *blits-in-*$\mathcal{L}_{in}$ $S \Longrightarrow$
$L = L' \Longrightarrow$
mop-maximum-level-removed-wl $L\ S \leq \Downarrow$*bool-rel* (*mop-maximum-level-removed-l* $L'\ S'$)›
unfolding *mop-maximum-level-removed-wl-def mop-maximum-level-removed-l-def*
apply *refine-rcg*
subgoal unfolding *mop-maximum-level-removed-wl-pre-def* **by** *blast*
subgoal by *auto*
done

lemma *skip-and-resolve-loop-wl-spec*:
‹(*skip-and-resolve-loop-wl*, *skip-and-resolve-loop-l*)
$\in \{(T'::'v\ \textit{twl-st-wl},\ T).$
$(T',\ T) \in$ *state-wl-l None* $\wedge$
correct-watching $T' \wedge$ *blits-in-*$\mathcal{L}_{in}$ $T' \wedge$
$0 <$ *count-decided* (*get-trail-wl* T')$\} \rightarrow$
$\langle\{(T',\ T).$
$(T',\ T) \in$ *state-wl-l None* $\wedge$
correct-watching $T' \wedge$ *blits-in-*$\mathcal{L}_{in}$ $T'\}\rangle$*nres-rel*›
(**is** ‹*?s* $\in$ *?A* $\rightarrow \langle$*?B*$\rangle$*nres-rel*›)
proof −
have *get-conflict-wl*: ‹((*False*, S'), *False*, S)
$\in$ *Id* $\times_r$ $\{(T',\ T).\ (T',\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $T' \wedge$ *blits-in-*$\mathcal{L}_{in}$ $T'\}$›
(**is** ‹- $\in$ *?B*›)
if ‹$(S',\ S) \in$ *state-wl-l None*› **and** ‹*correct-watching* S'› **and** ‹*blits-in-*$\mathcal{L}_{in}$ S'›
for S :: ‹$'v$ *twl-st-l*› **and** S' :: ‹$'v$ *twl-st-wl*›
using *that* **by** (*cases* S') (*auto simp*: *state-wl-l-def*)
have [*simp*]: ‹*correct-watching* (*tl aa*, *ca*, *da*, *ea*, *fa*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ha*, *h*) $\longleftrightarrow$
correct-watching (*aa*, *ca*, *None*, *ea*, *fa*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ha*, *h*)›
for *aa ba ca L da ea fa ha h NS US NEk UEk*
by (*auto simp*: *correct-watching.simps tl-state-wl-def clause-to-update-def*)
have [*simp*]: ‹*NO-MATCH None da* $\Longrightarrow$ *correct-watching* (*aa*, *ca*, *da*, *ea*, *fa*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ha*, *h*) $\longleftrightarrow$
correct-watching (*aa*, *ca*, *None*, *ea*, *fa*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ha*, *h*)›
for *aa ba ca L da ea fa ha h NS US N0 U0*
by (*auto simp*: *correct-watching.simps tl-state-wl-def clause-to-update-def*)

have *inv*: ‹*skip-and-resolve-loop-wl-inv* $S'\ b'\ T'$›

if
‹$(S', S) \in ?A$› **and**
‹*get-conflict-wl* $S' \neq$ *None*› **and**
bt-inv: ‹*case* bT *of* $(x, xa) \Rightarrow$ *skip-and-resolve-loop-inv-l* S x xa› **and**
‹$(b'T', bT) \in ?B$› **and**
$b'T'$: ‹$b'T' = (b', T')$›
for S' S $b'T'$ bT b' T'
proof −
obtain b T **where** bT: ‹$bT = (b, T)$› **by** (*cases* bT)
show *?thesis*
unfolding *skip-and-resolve-loop-wl-inv-def*
apply (*rule exI*[*of* - T])
apply (*rule exI*[*of* - S])
using *that* **by** (*auto simp*: bT $b'T'$)
qed

show H: ‹$?s \in ?A \rightarrow \langle\{(T', T).\ (T', T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $T' \wedge$ *blits-in-*$\mathcal{L}_{in}$ $T'\}\rangle$*nres-rel*›
unfolding *skip-and-resolve-loop-wl-def skip-and-resolve-loop-l-def*
apply (*refine-rcg get-conflict-wl tl-state-wl-tl-state-l*
mop-update-confl-tl-wl-mop-update-confl-tl-l mop-hd-trail-wl-mop-hd-trail-l
mop-lit-notin-conflict-wl-mop-lit-notin-conflict-l
mop-maximum-level-removed-wl-mop-maximum-level-removed-l)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule inv*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto intro*!: *tl-state-wl-tl-state-l*)
subgoal for S' S $b'T'$ bT b' T' **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

6.5.5 Backtrack

definition *find-decomp-wl* :: ‹'v literal ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹*find-decomp-wl* = (λL (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W).
SPEC(λS. ∃ K M2 M1. S = (M1, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) ∧
(*Decided* K # M1, M2) ∈ set (*get-all-ann-decomposition* M) ∧
get-level M K = *get-maximum-level* M (*the* D − {#−L#}) + 1))›

definition *find-lit-of-max-level-wl* :: ‹'v twl-st-wl ⇒ 'v literal ⇒ 'v literal nres› **where**
‹*find-lit-of-max-level-wl* = (λ(M, N, D, NE, UE, NEk, UEk, NS, US, Q, W) L.
SPEC(λL'. L' ∈# *remove1-mset* (−L) (*the* D) ∧ *get-level* M L' = *get-maximum-level* M (*the* D − {#−L#})))›

fun *extract-shorter-conflict-wl* :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹*extract-shorter-conflict-wl* (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) = *SPEC*(λS.
∃ D'. D' ⊆# *the* D ∧ S = (M, N, *Some* D', NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) ∧
clause `# *twl-clause-of* `# *ran-mf* N + NE + NEk + UE + UEk + NS + US ⊨*pm* D' ∧ −*lit-of* (*hd* M) ∈# D')›

declare *extract-shorter-conflict-wl.simps*[*simp del*]
lemmas *extract-shorter-conflict-wl-def* = *extract-shorter-conflict-wl.simps*

definition *backtrack-wl-inv* **where**
‹*backtrack-wl-inv* S ⟷ (∃ S'. (S, S') ∈ *state-wl-l None* ∧ *backtrack-l-inv* S' ∧ *correct-watching* S ∧ *blits-in-*$\mathcal{L}_{in}$ S)
›

Rougly: we get a fresh index that has not yet been used.

definition *get-fresh-index-wl* :: ‹'v clauses-l ⇒ - ⇒ - ⇒ nat nres› **where**
‹*get-fresh-index-wl* N NUE W = *SPEC*(λi. i > 0 ∧ i ∉# *dom-m* N ∧
(∀ L ∈# *all-lits-of-mm* (*mset* `# *ran-mf* N + NUE) . i ∉ *fst* ` *set* (W L)))›

definition (**in** −) *list-of-mset2*
:: ‹'v literal ⇒ 'v literal ⇒ 'v clause ⇒ 'v clause-l nres›
where
‹*list-of-mset2* L L' D =
SPEC (λE. *mset* E = D ∧ E!0 = L ∧ E!1 = L' ∧ *length* E ≥ 2)
›
definition *propagate-bt-wl-pre* **where**
‹*propagate-bt-wl-pre* L L' S ⟷
(∃ S'. (S, S') ∈ *state-wl-l None* ∧ *propagate-bt-l-pre* L L' S')›

definition *propagate-bt-wl* :: ‹'v literal ⇒ 'v literal ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹*propagate-bt-wl* = (λL L' (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). *do* {
ASSERT(*propagate-bt-wl-pre* L L' (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W));
D'' ← *list-of-mset2* (−L) L' (*the* D);
i ← *get-fresh-index-wl* N (NE + UE + NEk + UEk + NS + US + N0 + U0) W;
let b = (*length* ([−L, L'] @ (*remove1* (−L) (*remove1* L' D''))) = 2);
M ← *cons-trail-propagate-l* (−L) i M;
RETURN (M,
fmupd i ([−L, L'] @ (*remove1* (−L) (*remove1* L' D'')), *False*) N,
None, NE, UE, NEk, UEk, NS, US, N0, U0, {#L#}, W(−L:= W (−L) @ [(i, L', b)], L':= W L' @ [(i, −L, b)]))
})›

definition *propagate-unit-bt-wl-pre* **where**
‹*propagate-unit-bt-wl-pre L S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *propagate-unit-bt-l-pre L S'*)›

definition *propagate-unit-bt-wl* :: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*propagate-unit-bt-wl* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(*L* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
ASSERT(*propagate-unit-bt-wl-pre L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
M ← *cons-trail-propagate-l* (−*L*) *0 M*;
RETURN (*M*, *N*, *None*, *NE*, *UE*, *NEk*, *add-mset* (*the D*) *UEk*, *NS*, *US*, *N0*, *U0*, {#*L*#}, *W*)})›

definition *mop-lit-hd-trail-wl-pre* :: ‹'*v twl-st-wl* ⇒ *bool*› **where**
‹*mop-lit-hd-trail-wl-pre S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *mop-lit-hd-trail-l-pre S'*)›

definition *mop-lit-hd-trail-wl* :: ‹'*v twl-st-wl* ⇒ ('*v literal*) *nres*› **where**
‹*mop-lit-hd-trail-wl S* = *do*{
ASSERT(*mop-lit-hd-trail-wl-pre S*);
SPEC(λ*L*. *L* = *lit-of* (*hd* (*get-trail-wl S*)))
}›

definition *backtrack-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*backtrack-wl S* =
do {
ASSERT(*backtrack-wl-inv S*);
L ← *mop-lit-hd-trail-wl S*;
S ← *extract-shorter-conflict-wl S*;
S ← *find-decomp-wl L S*;

if size (*the* (*get-conflict-wl S*)) > *1*
then do {
L' ← *find-lit-of-max-level-wl S L*;
propagate-bt-wl L L' S
}
else do {
propagate-unit-bt-wl L S
}
}›

lemma *all-lits-st-learn-simps*:
assumes
L1: ‹*L1* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
L2: ‹*L2* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
UW: ‹*set-mset* (*all-lits-of-m* (*mset UW*)) ⊆
set-mset (*all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))› **and**
i-dom: ‹*i* ∉# *dom-m N*›
shows
‹*set-mset* (*all-lits-st* (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)) =
set-mset (*all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))›
using *assms*
by (*auto 5 3 simp*: *all-lits-st-def all-lits-def all-lits-of-mm-union all-lits-of-mm-add-mset*
ran-m-mapsto-upd-notin all-lits-of-m-add-mset in-all-lits-of-mm-ain-atms-of-iff
in-all-lits-of-m-ain-atms-of-iff atm-of-eq-atm-of)

lemma *correct-watching-learn2*:
assumes
L1: ‹*L1* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
L2: ‹*L2* ∈# *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
UW: ‹*set-mset* (*all-lits-of-m* (*mset UW*)) ⊆
set-mset (*all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))› **and**
i-dom: ‹*i* ∉# *dom-m N*› **and**
fresh: ‹⋀*L*. *L*∈#*all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟹ *i* ∉ *fst* ‘ *set* (*W L*)› **and**
[*iff*]: ‹*L1* ≠ *L2*› **and**
b: ‹*b* ⟷ *length* (*L1* # *L2* # *UW*) = 2›
shows
‹*correct-watching* (*K* # *M*, *fmupd i* (*L1* # *L2* # *UW*, *b*′) *N*,
D, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W* (*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) ⟷
correct-watching (*M*, *N*, *D*′, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*′, *W*)›
(**is** ‹*?l* ⟷ *?c*› **is** ‹*correct-watching* (-, *?N*, -) = -›)
proof −
let *?S* = ‹(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
let *?T* = ‹(*M*, *fmupd i* (*L1* # *L2* # *UW*, *b*′) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *W*)›
let *?L* = ‹*all-lits-st ?S*›
have *L*-*mono*: ‹*set-mset ?L* = *set-mset* (*all-lits-st ?T*)›
using *assms*(*1*−*3*) *i-dom*
by (*auto 5 3 simp*: *all-lits-st-def all-lits-def all-lits-of-mm-union all-lits-of-mm-add-mset ran-m-mapsto-upd-notin all-lits-of-m-add-mset in-all-lits-of-mm-ain-atms-of-iff in-all-lits-of-m-ain-atms-of-iff atm-of-eq-atm-of*)

have *LT*: ‹*set-mset ?L* = *set-mset* (*all-lits-st* (*K* # *M*, *fmupd i* (*L1* # *L2* # *UW*, *b*′) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W* (*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])))›
using *L*-*mono* **by** (*auto simp*: *all-lits-st-def*)

have [*iff*]: ‹*L2* ≠ *L1*›
using ‹*L1* ≠ *L2*› **by** (*subst eq-commute*)
have [*simp*]: ‹*clause-to-update L1* (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b*′) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}) =
add-mset i (*clause-to-update L1* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›
for *L2 UW*
using *i-dom*
by (*auto simp*: *clause-to-update-def intro*: *filter-mset-cong*)
have [*simp*]: ‹*clause-to-update L2* (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b*′) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}) =
add-mset i (*clause-to-update L2* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›
for *L1 UW*
using *i-dom*
by (*auto simp*: *clause-to-update-def intro*: *filter-mset-cong*)
have [*simp*]: ‹*x* ≠ *L1* ⟹ *x* ≠ *L2* ⟹
clause-to-update x (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b*′) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}) =
clause-to-update x (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})› **for** *x UW*
using *i-dom*
by (*auto simp*: *clause-to-update-def intro*: *filter-mset-cong*)

have *H*′:
‹{#*ia* ∈# *fst* ‘# *mset* (*W x*). *ia* = *i* ∨ *ia* ∈# *dom-m N*#} = {#*ia* ∈# *fst* ‘# *mset* (*W x*). *ia* ∈#

```
dom-m N#}›
    if ‹x ∈# ?L› for x
  using i-dom fresh[of x] that
  by (auto simp: clause-to-update-def intro!: filter-mset-cong)
 have [simp]:
  ‹clause-to-update L1 (K # M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#}) =
    clause-to-update L1 (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})›
  for L1 N D NE UE M K NS US
  by (auto simp: clause-to-update-def)

 show ?thesis
 proof (rule iffI)
  assume corr: ?l
  have
    H: ‹⋀L ia K' b''. (L∈#?L ⟹
    distinct-watched ((W(L1 := W L1 @ [(i, L2, b)], L2 := W L2 @ [(i, L1, b)])) L) ∧
    ((ia, K', b'')∈#mset ((W(L1 := W L1 @ [(i, L2, b)], L2 := W L2 @ [(i, L1, b)])) L) ⟶
      ia ∈# dom-m (fmupd i (L1 # L2 # UW, b') N) ⟶
      K' ∈ set (fmupd i (L1 # L2 # UW, b') N ∝ ia) ∧ K' ≠ L ∧
      correctly-marked-as-binary (fmupd i (L1 # L2 # UW, b') N) (ia, K', b'') ) ∧
    ((ia, K', b'')∈#mset ((W(L1 := W L1 @ [(i, L2, b)], L2 := W L2 @ [(i, L1, b)])) L) ⟶
      b'' ⟶ ia ∈# dom-m (fmupd i (L1 # L2 # UW, b') N)) ∧
    {#ia ∈# fst `#
          mset ((W(L1 := W L1 @ [(i, L2, b)], L2 := W L2 @ [(i, L1, b)])) L).
     ia ∈# dom-m (fmupd i (L1 # L2 # UW, b') N)#} =
    clause-to-update L
      (K # M, fmupd i (L1 # L2 # UW, b') N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#},
{#}))›
    using corr unfolding correct-watching.simps LT
    by fast+

  have ‹x ∈# ?L ⟹
      distinct-watched (W x) ∧
      (xa ∈# mset (W x) ⟶ (((case xa of (i, K, b'') ⇒ i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K
≠ x ∧
        correctly-marked-as-binary N (i, K, b'')) ∧
        (case xa of (i, K, b'') ⇒ b'' ⟶ i ∈# dom-m N)))) ∧
      {#i ∈# fst `# mset (W x). i ∈# dom-m N#} = clause-to-update x (M, N, D, NE, UE, NEk,
UEk, NS, US, N0, U0, {#}, {#})›
    for x xa
    supply correctly-marked-as-binary.simps[simp]
    using H[of x ‹fst xa› ‹fst (snd xa)› ‹snd (snd xa)›] fresh[of x] i-dom
    apply (cases ‹x = L1›; cases ‹x = L2›)
    subgoal
      by (cases xa)
        (auto dest!: multi-member-split simp: H')
    subgoal
      by (cases xa) (force simp add: H' split: if-splits)
    subgoal
      by (cases xa)
        (force simp add: H' split: if-splits)
    subgoal
      by (cases xa)
        (force simp add: H' split: if-splits)
    done
  then show ?c
```

unfolding *correct-watching.simps Ball-def*
by (*auto 5 5 simp add*: *all-lits-of-mm-add-mset all-lits-of-m-add-mset*
all-conj-distrib all-lits-of-mm-union clause-to-update-def
dest: *multi-member-split*)
next
assume *corr*: *?c*
have ‹*?L* = *all-lits-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q'*, *W*)›
by *auto*
then have
H: ‹⋀*L ia K' b''*. (*L*∈# *?L* ⟹
distinct-watched (*W L*) ∧
((*ia*, *K'*, *b''*)∈#*mset* (*W L*) ⟶
ia ∈# *dom-m N* ⟶
K' ∈ *set* (*N* ∝ *ia*) ∧ *K'* ≠ *L* ∧ *correctly-marked-as-binary N* (*ia*, *K'*, *b''*)) ∧
((*ia*, *K'*, *b''*)∈#*mset* (*W L*) ⟶ *b''* ⟶ *ia* ∈# *dom-m N*) ∧
{#*ia* ∈# *fst* '# *mset* (*W L*). *ia* ∈# *dom-m N*#} = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›
using *corr* **unfolding** *correct-watching.simps clause-to-update-def get-clauses-l.simps*
by *auto*
have ‹*x* ∈# *?L* ⟶
distinct-watched ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*) ∧
(*xa* ∈# *mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*) ⟶
(*case xa of* (*ia*, *K*, *b''*) ⇒ *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*) ⟶
K ∈ *set* (*fmupd i* (*L1* # *L2* # *UW*, *b'*) *N* ∝ *ia*) ∧ *K* ≠ *x* ∧
correctly-marked-as-binary (*fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*) (*ia*, *K*, *b''*))) ∧
(*xa* ∈# *mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*) ⟶
(*case xa of* (*ia*, *K*, *b''*) ⇒ *b''* ⟶ *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*))) ∧
{#*ia* ∈# *fst* '# *mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*). *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*)#} =
clause-to-update x (*K* # *M*, *fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
for *x* :: ‹*'a literal*› **and** *xa*
supply *correctly-marked-as-binary.simps*[*simp*]
using *H*[*of x* ‹*fst xa*› ‹*fst* (*snd xa*)› ‹*snd* (*snd xa*)›] *fresh*[*of x*] *i-dom b*
apply (*cases* ‹*x* = *L1*›; *cases* ‹*x* = *L2*›)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H'*)
subgoal
using *L-mono*
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H'*)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H'*)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H'*)
done
then show *?l*
using *L-mono*
unfolding *correct-watching.simps Ball-def*
by *auto*
qed
qed

lemma *all-lits-fmupd-new*[*simp*]:
‹*i* ∉# *dom-m NU* ⟹ *all-lits* (*fmupd i C NU*) *NUE* = *all-lits-of-m* (*mset* (*fst C*)) + *all-lits NU NUE*›
unfolding *all-lits-def* **by** (*auto simp*: *ran-m-mapsto-upd-notin all-lits-of-mm-add-mset*)

lemma *blits-in-$\mathcal{L}_{in}$-keep-watch'''*:
assumes
K': ‹*K'* ∈# *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
‹*L'* ∈# *all-lits-st* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)› **and**
w:‹ *blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g*)›
shows ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *f*, *g* (*K* := (*g K*) @[(*i*, *K'*, *b'*)], *L* := *g L* @ [(*i'*, *L'*, *b''*)]))›
using *assms*
unfolding *blits-in-$\mathcal{L}_{in}$-def*
by (*auto split*: *if-splits elim*!: *in-set-upd-cases*)

lemma *backtrack-wl-spec*:
‹(*backtrack-wl*, *backtrack-l*)
∈ {(*T'*::'*v twl-st-wl*, *T*).
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T' ∧ *blits-in-$\mathcal{L}_{in}$ T'* ∧
get-conflict-wl T' ≠ *None* ∧
get-conflict-wl T' ≠ *Some* {#}} →
⟨{(*T'*, *T*).
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T' ∧ *blits-in-$\mathcal{L}_{in}$ T'*}⟩*nres-rel*›
(**is** ‹*?bt* ∈ *?A* → ⟨*?B*⟩*nres-rel*›)
proof −
have *H*: ‹*A* ⊨*p C* ⟹ ¬*B* ⊨*p C* ⟹ *A* ⊆ *B* ⟹ *False*› **for** *A B C*
by (*meson true-clss-cls-subset*)
have *extract-shorter-conflict-wl*: ‹*extract-shorter-conflict-wl S'*
≤ ⇓ {(*U'*::'*v twl-st-wl*, *U*).
(*U'*, *U*) ∈ *state-wl-l None* ∧ *equality-except-conflict-wl U' S'* ∧
the (*get-conflict-wl U'*) ⊆# *the* (*get-conflict-wl S'*) ∧
get-conflict-wl U' ≠ *None* ∧ *correct-watching U'* ∧ *blits-in-$\mathcal{L}_{in}$ U'*}
(*extract-shorter-conflict-l S*)›
(**is** ‹- ≤ ⇓ *?extract* -›)
if ‹(*S'*, *S*) ∈ *?A*›
for *S' S*
apply (*cases S'*; *cases S*)
apply *clarify*
unfolding *extract-shorter-conflict-wl-def extract-shorter-conflict-l-def*
apply (*refine-rcg*, *rule RES-refine*)
using *that*
by (*auto simp*: *extract-shorter-conflict-wl-def extract-shorter-conflict-l-def*
mset-take-mset-drop-mset' state-wl-l-def correct-watching.simps Un-assoc
clause-to-update-def blits-in-$\mathcal{L}_{in}$-def dest!: *multi-member-split*
dest: *H*)
have *find-decomp-wl*: ‹*find-decomp-wl L T'*
≤ ⇓ {(*U'*::'*v twl-st-wl*, *U*).
(*U'*, *U*) ∈ *state-wl-l None* ∧ *equality-except-trail-wl U' T'* ∧
(∃ *M*. *get-trail-wl T'* = *M* @ *get-trail-wl U'*) ∧
correct-watching U' ∧ *blits-in-$\mathcal{L}_{in}$ U'*} (*find-decomp L' T*)›
(**is** ‹- ≤ ⇓ *?find* -›)

if ‹$(S', S) \in ?A$› ‹$L = L'$› ‹$(T', T) \in ?extract\ S'$›
for $S'\ S\ T\ T'\ L\ L'$
using *that*
apply (*cases* T; *cases* T')
apply *clarify*
unfolding *find-decomp-wl-def find-decomp-def prod.case*
apply (*rule RES-refine*)
apply (*auto 5 5 simp add*: *state-wl-l-def find-decomp-wl-def find-decomp-def correct-watching.simps clause-to-update-def blits-in-$\mathcal{L}_{in}$-def dest*!: *multi-member-split*)
done

have *find-lit-of-max-level-wl*: ‹*find-lit-of-max-level-wl* $T'\ LLK'$
$\leq \Downarrow \{(L', L).\ L = L' \wedge L' \in\# the\ (get\text{-}conflict\text{-}wl\ T') \wedge L' \in\# the\ (get\text{-}conflict\text{-}wl\ T') - \{\#-LLK'\#\}\}$
(*find-lit-of-max-level-l* $T\ L$)›
(**is** ‹- $\leq \Downarrow$ *?find-lit* -›)
if ‹$L = LLK'$› ‹$(T', T) \in ?find\ S'$›
for $S'\ S\ T\ T'\ L\ LLK'$
using *that*
apply (*cases* T; *cases* T'; *cases* S')
apply *clarify*
unfolding *find-lit-of-max-level-wl-def find-lit-of-max-level-l-def prod.case*
apply (*rule RES-refine*)
apply (*auto simp add*: *find-lit-of-max-level-wl-def find-lit-of-max-level-def state-wl-l-def dest*: *in-diffD*)
done
have *empty*: ‹*literals-to-update-wl* $S' = \{\#\}$› **if** *bt*: ‹*backtrack-wl-inv* S'› **for** S'
using *bt* **apply** −
unfolding *backtrack-wl-inv-def backtrack-l-inv-def*
apply *normalize-goal+*
apply (*auto simp*: *twl-struct-invs-def*)
done

have [*refine*]: ‹$(S, S') \in \{(T', T).$
$(T', T) \in$ *state-wl-l None* $\wedge$
correct-watching $T' \wedge$
blits-in-$\mathcal{L}_{in}$ $T' \wedge$
get-conflict-wl $T' \neq$ *None* $\wedge$
get-conflict-wl $T' \neq$ *Some* $\{\#\}\} \Longrightarrow$
mop-lit-hd-trail-wl $S \leq \Downarrow Id$ (*mop-lit-hd-trail-l* S')› **for** $S\ S'$
unfolding *mop-lit-hd-trail-wl-def mop-lit-hd-trail-l-def*
apply *refine-rcg*
subgoal unfolding *mop-lit-hd-trail-wl-pre-def*
by *blast*
subgoal by *auto*
done

have *id*: ‹$(x, x') \in Id \Longrightarrow f\ x \leq \Downarrow Id\ (f\ x')$› **for** $x\ x'\ f$
by *auto*

have *id3*: ‹$((x, y, z), (x', y', z')) \in Id \Longrightarrow f\ x\ y\ z \leq \Downarrow Id\ (f\ x'\ y'\ z')$› **for** $x\ x'\ f\ y\ y'\ z\ z'$
by *auto*

have *cons-trail-propagate-l*: ‹*cons-trail-propagate-l* $L\ i\ M0$
$\leq \Downarrow\{(M, M').\ M = M' \wedge M' = Propagated\ L\ i\ \#\ M0\}$

(*cons-trail-propagate-l L′ i′ M′*)›
if ‹*L* = *L′*› ‹*i* = *i′*› ‹*M0* = *M′*› **for** *L i L′ i′ M′ M0*
using *that* **unfolding** *cons-trail-propagate-l-def*
by (*auto intro*!: *ASSERT-refine-right*)

have *fresh*: ‹*get-fresh-index-wl N* (*NE* + *UE* + *NEk* + *UEk* + *NS* + *US* + *N0* + *U0*) *W* ≤
⇓ {(*i*, *i′*). *i* = *i′* ∧ *i* ∉# *dom-m N* ∧
(∀ *L* ∈# *all-lits-st* (*x1*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*). *i* ∉ *fst* ‘ *set* (*W* *L*))}
(*get-fresh-index N′*)›
if ‹*N* = *N′*› ‹*propagate-bt-wl-pre L L′* (*x1*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)›
for *N N′ NUE W NE UE N0 U0 x1e x2e NS US x1 D L L′ NEk UEk*
unfolding *that get-fresh-index-def get-fresh-index-wl-def all-lits-def all-lits-st-def*
by (*auto intro*: *RES-refine simp*: *ac-simps*)

have *blit*: ‹*i* ∉# *dom-m x1a* ⟹
set-mset (*all-lits-of-m* (*mset C*)) ⊆ *set-mset* (*all-lits-st* (*x1*, *x1a*, *Some xd*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*)) ⟹
blits-in-$\mathcal{L}_{in}$
(*x1*, *x1a*, *Some xd*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹
blits-in-$\mathcal{L}_{in}$
(*x1*,
fmupd i (*C*, *b*) *x1a*,
None, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *x2e*)› **for** *x x1a x1 xd x1d x1c x1e x2e i C b NS US N0 U0 NEk UEk*
by (*auto simp*: *blits-in-*$\mathcal{L}_{in}$*-def dest*!: *multi-member-split simp*: *all-lits-st-def*)
have [*simp*]: ‹*mset* (*unwatched-l* (*x*)) + *mset* (*watched-l* (*x*)) = *mset x*› **for** *x*
by (*metis add.commute mset-take-mset-drop-mset′*)

have *propagate-bt-wl*: ‹(*Sb*, *Sc*)
∈ {(*U′*, *U*).
(*U′*, *U*) ∈ *state-wl-l None* ∧
equality-except-trail-wl U′ S ∧
(∃ *M*. *get-trail-wl S* = *M* @ *get-trail-wl U′*) ∧
correct-watching U′ ∧ *blits-in-*$\mathcal{L}_{in}$ *U′*} ⟹
(*L*, *La*) ∈ *Id* ⟹
(*L′*, *L′a*)
∈ {(*L′*, *La*).
La = *L′* ∧
L′ ∈# *the* (*get-conflict-wl Sb*) ∧
L′ ∈# *remove1-mset* (− *L*)
(*the* (*get-conflict-wl Sb*))} ⟹
propagate-bt-wl L L′ Sb
≤ ⇓ {(*T′*, *T*).
(*T′*, *T*) ∈ *state-wl-l None* ∧
correct-watching T′ ∧ *blits-in-*$\mathcal{L}_{in}$ *T′*}
(*propagate-bt-l La L′a Sc*)› **for** *a a′ L La S Sa Sb Sc L′ L′a*
unfolding *propagate-bt-wl-def propagate-bt-l-def Let-def list-of-mset2-def list-of-mset-def*
apply (*cases Sc*; *hypsubst*)
unfolding *prod.case*
apply (*refine-rcg fresh*)
subgoal unfolding *propagate-bt-wl-pre-def* **by** *fast*
subgoal by (*auto simp*: *propagate-bt-wl-pre-def state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
apply *assumption*
apply (*rule cons-trail-propagate-l*)

subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal
unfolding *propagate-bt-l-pre-def propagate-bt-pre-def*
apply *normalize-goal+*
apply (*simp add*: *eq-commute*[*of* - ‹(-, -)›])
apply (*auto simp*: *state-wl-l-def*)
apply (*auto 5 3 simp*: *ran-m-mapsto-upd-notin mset-take-mset-drop-mset′*
uminus-lit-swap blits-in-$\mathcal{L}_{in}$-propagate all-lits-of-m-add-mset
state-wl-l-def twl-st-l-def
dest: *all-lits-of-m-diffD*
simp flip: *image-mset-union*
intro!: *blit blits-in-$\mathcal{L}_{in}$-keep-watch‴ correct-watching-learn2*[*THEN iffD2*]
all-lits-st-learn-simps[*THEN arg-cong*[*of* - - ‹ λ*x*. - ∈ *x*›], *THEN iffD2*])
done
done

have *correct-watching-unit*: ‹*La* ∈# *all-lits-st* (*x1*, *x1a*, *x1b*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹
correct-watching (*x1*, *x1a*, *x1b*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹
correct-watching
(*Propagated* (− *La*) *0* # *x1*, *x1a*, *None*, *x1c*,
x1d, *NEk*, *add-mset* {#−*La*#} *UEk*, *NS*, *US*, *N0*, *U0*, {#*La*#}, *x2e*)›
‹*La* ∈# *all-lits-st* (*x1*, *x1a*, *x1b*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹
blits-in-$\mathcal{L}_{in}$
(*x1*, *x1a*, *x1b*, *x1c*, *x1d*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *x1e*, *x2e*) ⟹
blits-in-$\mathcal{L}_{in}$
(*Propagated* (− *La*) *0* # *x1*, *x1a*, *None*, *x1c*, *x1d*, *NEk*, *add-mset* {#−*La*#} *UEk*, *NS*, *US*, *N0*, *U0*, {#*La*#},
x2e)›
for *a b c d e f g x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e*
M Ma La NS US N0 U0 NEk UEk
unfolding *all-lits-st-def all-lits-def*
apply (*subst* (*asm*) *in-all-lits-of-mm-uminus-iff*[*symmetric*])
apply (*auto simp*: *correct-watching.simps all-lits-of-mm-add-mset*
all-lits-of-m-add-mset clause-to-update-def blits-in-$\mathcal{L}_{in}$-def
all-lits-st-def all-lits-def
dest: *multi-member-split*)
apply (*auto simp*: *correct-watching.simps all-lits-of-mm-add-mset all-lits-def*
all-lits-of-m-add-mset clause-to-update-def in-all-lits-of-mm-uminus-iff ac-simps
dest: *multi-member-split*)
done

have *propagate-unit-bt-wl*: ‹(*Sb*, *Sc*)
∈ {(*U′*, *U*).
(*U′*, *U*) ∈ *state-wl-l None* ∧
equality-except-trail-wl U′ S ∧
(∃ *M*. *get-trail-wl S* = *M* @ *get-trail-wl U′*) ∧
correct-watching U′ ∧ *blits-in-$\mathcal{L}_{in}$ U′*} ⟹
(*L*, *La*) ∈ *Id* ⟹
propagate-unit-bt-wl L Sb
≤ ⇓ {(*T′*, *T*).
(*T′*, *T*) ∈ *state-wl-l None* ∧
correct-watching T′ ∧ *blits-in-$\mathcal{L}_{in}$ T′*}
(*propagate-unit-bt-l La Sc*)› **for** *a a′ L La S Sa Sb Sc L′ L′a*

unfolding *propagate-unit-bt-wl-def propagate-unit-bt-l-def Let-def list-of-mset-def*
apply (*cases Sc*; *hypsubst*)
unfolding *prod.case*
apply (*refine-rcg fresh*)
subgoal unfolding *in-pair-collect-simp* **by** *normalize-goal+* (*simp add*: *eq-commute*[*of* - ‹(-, -)›])
subgoal using *propagate-unit-bt-wl-pre-def* **by** *blast*
apply (*rule cons-trail-propagate-l*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal for *a b c d e f g x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e*
M Ma
unfolding *propagate-unit-bt-l-pre-def propagate-unit-bt-pre-def*
by *normalize-goal+*
(*auto simp*: *all-lits-def ran-m-mapsto-upd-notin all-lits-of-mm-add-mset state-wl-l-def*
twl-st-l-def all-lits-of-m-add-mset mset-take-mset-drop-mset′ in-all-lits-of-mm-uminus-iff
ac-simps
simp flip: *image-mset-union intro*!: *blit correct-watching-unit dest*!: *all-lits-of-m-diffD*)
done

show *?thesis*
unfolding *st-l-of-wl.simps get-trail-wl.simps list-of-mset-def*
backtrack-wl-def backtrack-l-def
apply (*refine-vcg find-decomp-wl find-lit-of-max-level-wl extract-shorter-conflict-wl*
propagate-bt-wl propagate-unit-bt-wl;
remove-dummy-vars)
subgoal using *backtrack-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

6.5.6 Backtrack, Skip, Resolve or Decide

definition *cdcl-twl-o-prog-wl-pre* **where**
‹*cdcl-twl-o-prog-wl-pre S* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *state-wl-l None* ∧
correct-watching S ∧
cdcl-twl-o-prog-l-pre S′)›

definition *cdcl-twl-o-prog-wl* :: ‹*′v twl-st-wl* ⇒ (*bool* × *′v twl-st-wl*) *nres*› **where**
‹*cdcl-twl-o-prog-wl S* =
do {
ASSERT(*cdcl-twl-o-prog-wl-pre S*);
do {
if get-conflict-wl S = *None*
then decide-wl-or-skip S
else do {
if count-decided (*get-trail-wl S*) > *0*
then do {
T ← *skip-and-resolve-loop-wl S*;
ASSERT(*get-conflict-wl T* ≠ *None* ∧ *get-conflict-wl T* ≠ *Some* {#});
U ← *backtrack-wl T*;
RETURN (*False*, *U*)
}

```
        else do {RETURN (True, S)}
      }
    }
  }
›

lemma [simp]: ‹blits-in-\mathcal{L}_in (decide-lit-wl L S) ⟷ blits-in-\mathcal{L}_in S›
  by (cases S) (auto simp: decide-lit-wl-def blits-in-\mathcal{L}_in-def)

lemma cdcl-twl-o-prog-wl-spec:
  ‹(cdcl-twl-o-prog-wl, cdcl-twl-o-prog-l) ∈ {(S::'v twl-st-wl, S'::'v twl-st-l).
     (S, S') ∈ state-wl-l None ∧
       correct-watching S ∧ blits-in-\mathcal{L}_in S} →_f
   ⟨{((brk::bool, T::'v twl-st-wl), brk'::bool, T'::'v twl-st-l).
     (T, T') ∈ state-wl-l None ∧
     brk = brk' ∧
     correct-watching T ∧ blits-in-\mathcal{L}_in T}⟩nres-rel›
   (is ‹?o ∈ ?A →_f ⟨?B⟩ nres-rel›)
proof −

  have [iff]: ‹correct-watching (decide-lit-wl L S) ⟷ correct-watching S› for L S
    by (cases S; auto simp: decide-lit-wl-def correct-watching.simps clause-to-update-def)
  have [iff]: ‹(decide-lit-wl L S, decide-lit-l L S') ∈ state-wl-l None›
    if ‹(S, S') ∈ state-wl-l None›
    for L S S'
    using that by (cases S; auto simp: decide-lit-wl-def decide-lit-l-def state-wl-l-def)
  have option-id: ‹x = x' ⟹ (x,x') ∈ ⟨Id⟩option-rel› for x x' by auto
  have damn-you-are-stupid-isabelle: ‹(a, a')
      ∈ {(S, S').
         (S, S') ∈ state-wl-l None ∧
         correct-watching S ∧ blits-in-\mathcal{L}_in S} ⟹
       get-conflict-l a' = None ⟹
       (a, a')
       ∈ {(T', T).
          (T', T) ∈ state-wl-l None ∧
          correct-watching T' ∧
          blits-in-\mathcal{L}_in T' ∧ get-conflict-wl T' = None}› for a a'
   by auto
  show cdcl-o: ‹?o ∈ ?A →_f
   ⟨{((brk::bool, T::'v twl-st-wl), brk'::bool, T'::'v twl-st-l).
     (T, T') ∈ state-wl-l None ∧
     brk = brk' ∧
     correct-watching T ∧ blits-in-\mathcal{L}_in T}⟩nres-rel›
    unfolding cdcl-twl-o-prog-wl-def cdcl-twl-o-prog-l-def
      fref-param1[symmetric]
    apply (refine-vcg skip-and-resolve-loop-wl-spec[to-⇓] backtrack-wl-spec[to-⇓]
      decide-wl-or-skip-spec[to-⇓, THEN order-trans]
      option-id)
    subgoal unfolding cdcl-twl-o-prog-wl-pre-def by blast
    subgoal by auto
    apply (rule damn-you-are-stupid-isabelle; assumption)
    subgoal by (auto intro!: conc-fun-R-mono)
    subgoal unfolding decide-wl-or-skip-pre-def by auto
    subgoal by auto
    subgoal by (auto simp: )
    subgoal by auto
```

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

6.5.7 Full Strategy

definition *cdcl-twl-stgy-prog-wl-inv* :: ‹*'v twl-st-wl* ⇒ *bool* × *'v twl-st-wl* ⇒ *bool*› **where**
‹*cdcl-twl-stgy-prog-wl-inv* $S_0 \equiv \lambda(brk,\ T).$
$(\exists\ T'\ S_0'.\ (T,\ T') \in$ *state-wl-l None* $\wedge$
$(S_0,\ S_0') \in$ *state-wl-l None* $\wedge$
cdcl-twl-stgy-prog-l-inv $S_0'\ (brk,\ T'))$›

definition *cdcl-twl-stgy-prog-wl* :: ‹*'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*cdcl-twl-stgy-prog-wl* S_0 =
do {
$(brk,\ T) \leftarrow$ $WHILE_T$*cdcl-twl-stgy-prog-wl-inv* S_0
$(\lambda(brk,\ \text{-}).\ \neg brk)$
$(\lambda(brk,\ S).$ *do* {
$T \leftarrow$ *unit-propagation-outer-loop-wl* S;
cdcl-twl-o-prog-wl T
})
$(False,\ S_0)$;
RETURN T
}›

theorem *cdcl-twl-stgy-prog-wl-spec*:
‹(*cdcl-twl-stgy-prog-wl*, *cdcl-twl-stgy-prog-l*) ∈ {(S::*'v twl-st-wl*, S').
(S, S') ∈ *state-wl-l None* ∧
correct-watching S ∧ *blits-in-*$\mathcal{L}_{in}$ S} →
⟨*state-wl-l None*⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* → ⟨*?B*⟩ *nres-rel*›)
proof −
have H: ‹(($False$, S'), $False$, S) ∈ {((brk', T'), (brk, T)). (T', T) ∈ *state-wl-l None* ∧ $brk' = brk$ ∧
correct-watching T' ∧ *blits-in-*$\mathcal{L}_{in}$ T'}›
if ‹(S', S) ∈ *state-wl-l None*› **and**
‹*correct-watching* S'› ‹*blits-in-*$\mathcal{L}_{in}$ S'›
for S' :: ‹*'v twl-st-wl*› **and** S :: ‹*'v twl-st-l*›
using *that* **by** *auto*

show *?thesis*
unfolding *cdcl-twl-stgy-prog-wl-def cdcl-twl-stgy-prog-l-def*
apply (*refine-rcg H unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
subgoal for S' S **by** (*cases* S') *auto*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-prog-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal for S' S $brk'T'$ $brkT$ brk' T' **by** *auto*
subgoal by *fast*
subgoal by *auto*
done

qed

theorem *cdcl-twl-stgy-prog-wl-spec'*:
‹(*cdcl-twl-stgy-prog-wl*, *cdcl-twl-stgy-prog-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-*$\mathcal{L}_{in}$ *S*} →
⟨{(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-*$\mathcal{L}_{in}$ *S*}⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* → ⟨*?B*⟩ *nres-rel*›)
proof −
have *H*: ‹((*False*, *S'*), *False*, *S*) ∈ {((*brk'*, *T'*), (*brk*, *T*)). (*T'*, *T*) ∈ *state-wl-l None* ∧ *brk'* = *brk* ∧
correct-watching T' ∧ *blits-in-*$\mathcal{L}_{in}$ *T'*}›
if ‹(*S'*, *S*) ∈ *state-wl-l None*› **and**
‹*correct-watching S'*› ‹*blits-in-*$\mathcal{L}_{in}$ *S'*›
for *S'* :: ‹'*v twl-st-wl*› **and** *S* :: ‹'*v twl-st-l*›
using *that* **by** *auto*
thm *unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
show *?thesis*
unfolding *cdcl-twl-stgy-prog-wl-def cdcl-twl-stgy-prog-l-def*
apply (*refine-rcg H unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
subgoal for *S' S* **by** (*cases S'*) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-prog-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal for *S' S brk'T' brkT brk' T'* **by** *auto*
subgoal by *fast*
subgoal by *auto*
done
qed

definition *cdcl-twl-stgy-prog-wl-pre* **where**
‹*cdcl-twl-stgy-prog-wl-pre S U* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *cdcl-twl-stgy-prog-l-pre T U* ∧ *correct-watching S* ∧ *blits-in-*$\mathcal{L}_{in}$ *S*)›

lemma *cdcl-twl-stgy-prog-wl-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-wl S* ≤ ⇓ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-norestart-run S'*)›
proof −
obtain *T* **where** *T*: ‹(*S*, *T*) ∈ *state-wl-l None*› ‹*cdcl-twl-stgy-prog-l-pre T S'*› ‹*correct-watching S*›
‹*blits-in-*$\mathcal{L}_{in}$ *S*›
using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-pre-def* **by** *blast*
show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-wl-spec*[*to-⇓*, *of S T*]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step'*)
apply (*rule cdcl-twl-stgy-prog-l-spec-final*[*of - S'*])
subgoal using *T* **by** *fast*
subgoal unfolding *conc-fun-chain* **by** *auto*
done
done

qed

definition *cdcl-twl-stgy-prog-break-wl* :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹*cdcl-twl-stgy-prog-break-wl* S_0 =
do {
$b \leftarrow$ *SPEC*(λ-. *True*);
(b, *brk*, T) ← *WHILE*$_T$λ(-, S). *cdcl-twl-stgy-prog-wl-inv* S_0 S
(λ(b, *brk*, -). $b \wedge \neg$*brk*)
(λ(-, *brk*, S). *do* {
$T \leftarrow$ *unit-propagation-outer-loop-wl* S;
$T \leftarrow$ *cdcl-twl-o-prog-wl* T;
$b \leftarrow$ *SPEC*(λ-. *True*);
RETURN (b, T)
})
(b, *False*, S_0);
if brk then RETURN T
else cdcl-twl-stgy-prog-wl T
}›

theorem *cdcl-twl-stgy-prog-break-wl-spec'*:
‹(*cdcl-twl-stgy-prog-break-wl*, *cdcl-twl-stgy-prog-break-l*) ∈ {(S::'v *twl-st-wl*, S').
(S, S') ∈ *state-wl-l None* ∧ *correct-watching* S ∧ *blits-in-*$\mathcal{L}_{in}$ S} →$_f$
⟨{(S::'v *twl-st-wl*, S'). (S, S') ∈ *state-wl-l None* ∧ *correct-watching* S ∧ *blits-in-*$\mathcal{L}_{in}$ S}⟩*nres-rel*›
(**is** ‹?o ∈ ?A →$_f$ ⟨?B⟩ *nres-rel*›)
proof −
have H: ‹((b', *False*, S'), b, *False*, S) ∈ {((b', *brk'*, T'), (b, *brk*, T)).
(T', T) ∈ *state-wl-l None* ∧ *brk'* = *brk* ∧ b' = b ∧
correct-watching T' ∧ *blits-in-*$\mathcal{L}_{in}$ T'}›
if ‹(S', S) ∈ *state-wl-l None*› **and**
‹*correct-watching* S'› **and**
‹(b', b) ∈ *bool-rel*› ‹*blits-in-*$\mathcal{L}_{in}$ S'›
for S' :: ‹'v *twl-st-wl*› **and** S :: ‹'v *twl-st-l*› **and** b' b :: *bool*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-prog-break-wl-def cdcl-twl-stgy-prog-break-l-def fref-param1*[*symmetric*]
apply (*refine-rcg H unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*]
cdcl-twl-stgy-prog-wl-spec'[*unfolded fref-param1*, *THEN fref-to-Down*])
subgoal for S' S **by** (*cases* S') *auto*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-prog-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal for S' S *brk'T'* *brkT* *brk'* T' **by** *auto*
subgoal by *fast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *fast*
subgoal by *auto*
done
qed

theorem *cdcl-twl-stgy-prog-break-wl-spec*:
‹(*cdcl-twl-stgy-prog-break-wl*, *cdcl-twl-stgy-prog-break-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧
correct-watching S ∧ *blits-in-*$\mathcal{L}_{in}$ *S*} →$_f$
⟨*state-wl-l None*⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* →$_f$ ⟨*?B*⟩ *nres-rel*›)
using *cdcl-twl-stgy-prog-break-wl-spec'*
apply −
apply (*rule mem-set-trans*)
prefer *2* **apply** *assumption*
apply (*match-fun-rel*, *solves simp*)
apply (*match-fun-rel*; *solves auto*)
done

lemma *cdcl-twl-stgy-prog-break-wl-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-break-wl S* ≤ ⇓ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-norestart-run S'*)›
proof −
obtain *T* **where** *T*: ‹(*S*, *T*) ∈ *state-wl-l None*› ‹*cdcl-twl-stgy-prog-l-pre T S'*› ‹*correct-watching S*› ‹*blits-in-*$\mathcal{L}_{in}$ *S*›
using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-pre-def* **by** *blast*
show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-break-wl-spec*[*unfolded fref-param1*[*symmetric*], *to-*⇓, *of S T*]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step'*)
apply (*rule cdcl-twl-stgy-prog-break-l-spec-final*[*of - S'*])
subgoal using *T* **by** *fast*
subgoal unfolding *conc-fun-chain* **by** *auto*
done
done
qed

definition *cdcl-twl-stgy-prog-early-wl* :: ‹'*v twl-st-wl* ⇒ (*bool* × '*v twl-st-wl*) *nres*› **where**
‹*cdcl-twl-stgy-prog-early-wl S*$_0$ =
do {
b ← *SPEC*(λ-. *True*);
(*b*, *brk*, *T*) ← *WHILE*$_T$λ(-, *S*). *cdcl-twl-stgy-prog-wl-inv S*$_0$ *S*
(λ(*b*, *brk*, -). *b* ∧ ¬*brk*)
(λ(-, *brk*, *S*). *do* {
T ← *unit-propagation-outer-loop-wl S*;
T ← *cdcl-twl-o-prog-wl T*;
b ← *SPEC*(λ-. *True*);
RETURN (*b*, *T*)
})
(*b*, *False*, *S*$_0$);
RETURN (*brk*, *T*)
}›

theorem *cdcl-twl-stgy-prog-early-wl-spec'*:

```
  ‹(cdcl-twl-stgy-prog-early-wl, cdcl-twl-stgy-prog-early-l) ∈ {(S::'v twl-st-wl, S').
      (S, S') ∈ state-wl-l None ∧ correct-watching S ∧ blits-in-ℒ_in S} →_f
    ⟨bool-rel ×_r {(S::'v twl-st-wl, S'). (S, S') ∈ state-wl-l None ∧ correct-watching S ∧ blits-in-ℒ_in
S}⟩nres-rel›
   (is ‹?o ∈ ?A →_f ⟨?B⟩ nres-rel›)
proof −
  have H: ‹((b', False, S'), b, False, S) ∈ {((b', brk', T'), (b, brk, T)).
      (T', T) ∈ state-wl-l None ∧ brk' = brk ∧ b' = b ∧
       correct-watching T' ∧ blits-in-ℒ_in T'}›
    if ‹(S', S) ∈ state-wl-l None› and
      ‹correct-watching S'› and
      ‹(b', b) ∈ bool-rel› ‹blits-in-ℒ_in S'›
    for S' :: ‹'v twl-st-wl› and S :: ‹'v twl-st-l› and b' b :: bool
    using that by auto
  show ?thesis
    unfolding cdcl-twl-stgy-prog-early-wl-def cdcl-twl-stgy-prog-early-l-def fref-param1[symmetric]
    apply (refine-rcg H unit-propagation-outer-loop-wl-spec[THEN fref-to-Down]
      cdcl-twl-o-prog-wl-spec[THEN fref-to-Down]
      cdcl-twl-stgy-prog-wl-spec'[unfolded fref-param1, THEN fref-to-Down])
    subgoal for S' S by (cases S') auto
    subgoal by auto
    subgoal by auto
    subgoal unfolding cdcl-twl-stgy-prog-wl-inv-def by blast
    subgoal by auto
    subgoal by auto
    subgoal for S' S brk'T' brkT brk' T' by auto
    subgoal by fast
    subgoal by auto
    subgoal by auto
    done
qed


theorem cdcl-twl-stgy-prog-early-wl-spec:
  ‹(cdcl-twl-stgy-prog-early-wl, cdcl-twl-stgy-prog-early-l) ∈ {(S::'v twl-st-wl, S').
      (S, S') ∈ state-wl-l None ∧
      correct-watching S ∧ blits-in-ℒ_in S} →_f
    ⟨bool-rel ×_r state-wl-l None⟩nres-rel›
   (is ‹?o ∈ ?A →_f ⟨?B⟩ nres-rel›)
  using cdcl-twl-stgy-prog-early-wl-spec'
  apply −
  apply (rule mem-set-trans)
  prefer 2 apply assumption
  apply (match-fun-rel, solves simp)
  apply (match-fun-rel; solves auto)
  done

lemma cdcl-twl-stgy-prog-early-wl-spec-final:
  assumes
    ‹cdcl-twl-stgy-prog-wl-pre S S'›
  shows
    ‹cdcl-twl-stgy-prog-early-wl S ≤ ⇓ (bool-rel ×_r (state-wl-l None O twl-st-l None)) (partial-conclusive-TWL-norestart-run
S')›
proof −
  obtain T where T: ‹(S, T) ∈ state-wl-l None› ‹cdcl-twl-stgy-prog-l-pre T S'› ‹correct-watching S›
‹blits-in-ℒ_in S›
```

using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-pre-def* **by** *blast*
have *H*: ‹{((*a*, *b*), *a'*, *b'*). (*a*, *a'*) ∈ *bool-rel* ∧ (*b*, *b'*) ∈ *state-wl-l None*} *O*
(*bool-rel* ×$_f$ *twl-st-l None*) = *bool-rel* ×$_r$ *state-wl-l None O twl-st-l None*›
by *fastforce*
show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-early-wl-spec*[*unfolded fref-param1*[*symmetric*], *to-⇓*, *of S T*]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step'*)
apply (*rule cdcl-twl-stgy-prog-early-l-spec-final*[*of - S'*])
subgoal using *T* **by** *fast*
subgoal unfolding *H conc-fun-chain* **by** *auto*
done
done
qed

6.5.8 Shared for restarts and inprocessing

definition *clauses-pointed-to* :: ‹'*v literal set* ⇒ ('*v literal* ⇒ '*v watched*) ⇒ *nat set*›
where
‹*clauses-pointed-to* $\mathcal{A}$ *W* ≡ ⋃(((‘) *fst*) ‘ *set* ‘ *W* ‘ $\mathcal{A}$)›

lemma *clauses-pointed-to-insert*[*simp*]:
‹*clauses-pointed-to* (*insert A* $\mathcal{A}$) *W* =
fst ‘ *set* (*W A*) ∪
clauses-pointed-to $\mathcal{A}$ *W*› **and**
clauses-pointed-to-empty[*simp*]:
‹*clauses-pointed-to* {} *W* = {}›
by (*auto simp*: *clauses-pointed-to-def*)

lemma *clauses-pointed-to-remove1-if*:
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹ *xa* ∈# *dom-m aa* ⟹
xa ∈ *clauses-pointed-to* (*set-mset* (*remove1-mset L* $\mathcal{A}$))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
xa ∈ *clauses-pointed-to* (*set-mset* (*remove1-mset L* $\mathcal{A}$)) *W*›
by (*cases* ‹*L* ∈# $\mathcal{A}$›)
(*fastforce simp*: *clauses-pointed-to-def*
dest!: *multi-member-split*)+

lemma *clauses-pointed-to-remove1-if2*:
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹ *xa* ∈# *dom-m aa* ⟹
xa ∈ *clauses-pointed-to* (*set-mset* ($\mathcal{A}$ − {#*L*, *L'*#}))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
xa ∈ *clauses-pointed-to* (*set-mset* ($\mathcal{A}$ − {#*L*, *L'*#})) *W*›
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹ *xa* ∈# *dom-m aa* ⟹
xa ∈ *clauses-pointed-to* (*set-mset* ($\mathcal{A}$ − {#*L'*, *L*#}))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
xa ∈ *clauses-pointed-to* (*set-mset* ($\mathcal{A}$ − {#*L'*, *L*#})) *W*›
by (*cases* ‹*L* ∈# $\mathcal{A}$›; *fastforce simp*: *clauses-pointed-to-def*
dest!: *multi-member-split*)+

lemma *clauses-pointed-to-remove1-if2-eq*:
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹
set-mset (*dom-m aa*) ⊆ *clauses-pointed-to* (*set-mset* ($\mathcal{A}$ − {#*L*, *L'*#}))

```
    (λa. if a = L then [] else W a) ⟷
   set-mset (dom-m aa) ⊆ clauses-pointed-to (set-mset (A − {#L, L'#})) W›
  ‹∀ L∈set (W L). fst L ∉# dom-m aa ⟹
    set-mset (dom-m aa) ⊆ clauses-pointed-to (set-mset (A − {#L', L#}))
     (λa. if a = L then [] else W a) ⟷
    set-mset (dom-m aa) ⊆ clauses-pointed-to (set-mset (A − {#L', L#})) W›
  by (auto simp: clauses-pointed-to-remove1-if2)

lemma negs-remove-Neg: ‹A ∉# A ⟹ negs A + poss A − {#Neg A, Pos A#} =
   negs A + poss A›
  by (induction A) auto
lemma poss-remove-Pos: ‹A ∉# A ⟹ negs A + poss A − {#Pos A, Neg A#} =
   negs A + poss A›
  by (induction A)  auto

end

theory Watched-Literals-List-Reduce
imports Watched-Literals-List-Restart
  Watched-Literals-List-Inprocessing
begin


end
theory Watched-Literals-List-Simp
  imports
    Watched-Literals-List-Reduce
    Watched-Literals-List-Inprocessing
begin
lemma cdcl-twl-inprocessing-l-count-dec:
  ‹cdcl-twl-inprocessing-l S T ⟹ count-decided (get-trail-l T) = count-decided (get-trail-l S)›
  by (induction rule: cdcl-twl-inprocessing-l.induct)
   (auto simp: cdcl-twl-unitres-l.simps cdcl-twl-unitres-true-l.simps
    cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps
    cdcl-twl-pure-remove-l.simps)

lemma rtranclp-cdcl-twl-inprocessing-l-count-dec:
  ‹cdcl-twl-inprocessing-l** S T ⟹ count-decided (get-trail-l T) = count-decided (get-trail-l S)›
  by (induction rule: rtranclp-induct)
    (auto dest!: cdcl-twl-inprocessing-l-count-dec)

inductive cdcl-twl-restart-l-inp for S T where
  ‹cdcl-twl-restart-l S T ⟹ cdcl-twl-restart-l-inp S T› |
  ‹cdcl-twl-inprocessing-l S T ⟹ cdcl-twl-restart-l-inp S T›

lemma cdcl-twl-restart-l-inp-twl-list-invs:
  ‹cdcl-twl-restart-l-inp S T ⟹ twl-list-invs S ⟹ twl-list-invs T›
  apply (induction rule: cdcl-twl-restart-l-inp.induct)
    using cdcl-twl-restart-l-list-invs apply blast
  using cdcl-twl-inprocessing-l-twl-list-invs by blast

lemma rtranclp-cdcl-twl-restart-l-inp-twl-list-invs:
  ‹cdcl-twl-restart-l-inp** S T ⟹ twl-list-invs S ⟹ twl-list-invs T›
  by (induction rule: rtranclp-induct)
    (auto dest: cdcl-twl-restart-l-inp-twl-list-invs)
```

lemma *rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l*:
  ‹*cdcl-twl-restart-l-inp*** *xa V* ⟹ *clauses-to-update-l xa* = {#} ⟹
  clauses-to-update-l V = {#}›
  apply (*induction rule*: *rtranclp-induct*)
  subgoal by *auto*
  subgoal premises *p*
    using *p*(*2*)
    by (*auto simp*: *cdcl-twl-restart-l-inp.simps*
      cdcl-twl-restart-l.simps cdcl-twl-inprocessing-l.simps
      cdcl-twl-unitres-l.simps cdcl-twl-unitres-true-l.simps
      cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps
      cdcl-twl-pure-remove-l.simps)
  done

lemma *rtranclp-cdcl-twl-inprocessing-l-cdcl-twl-l-inp*:
  ‹*cdcl-twl-inprocessing-l*** *S T* ⟹ *cdcl-twl-restart-l-inp*** *S T*›
  by (*induction rule*: *rtranclp-induct*) (*auto dest*!: *cdcl-twl-restart-l-inp.intros*)

lemma *cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*:
  assumes
    ‹*cdcl-twl-restart-l-inp S U*›
    ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
    ‹*twl-list-invs S*› **and**
    ‹*twl-struct-invs T*›
  obtains *V* **where**
    ‹(*U*, *V*) ∈ *twl-st-l None*›
    ‹*cdcl-twl-inp T V*›
  using *assms*
  apply (*induction rule*: *cdcl-twl-restart-l-inp.induct*)
  subgoal premises *p*
    using *p* **apply** −
    apply (*drule cdcl-twl-restart-l-invs*[*OF assms*(*2*,*3*,*4*), *of U*])
    apply *normalize-goal*+
    apply (*rule p*(*2*))
    apply *assumption*
    by (*auto dest*!: *cdcl-twl-inp.intros*)
  subgoal
    apply (*cases rule*: *cdcl-twl-inprocessing-l.cases*, *assumption*)
    using *cdcl-twl-inp.intros*(*3*) *cdcl-twl-unitres-l-cdcl-twl-unitres* **apply** *blast*
    apply (*meson cdcl-twl-inp.simps cdcl-twl-unitres-true-l-cdcl-twl-unitres-true*)
    apply (*meson cdcl-twl-inp.intros cdcl-twl-inprocessing-l-twl-st-l0*)
    apply (*meson cdcl-twl-inp.simps cdcl-twl-subresolution-l-cdcl-twl-subresolution*)
    using *cdcl-twl-inp.intros*(*6*) *cdcl-twl-pure-remove-l-cdcl-twl-pure-remove* **by** *blast*
  done

lemma *rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*:
  assumes
    ‹*cdcl-twl-restart-l-inp*** *S U*›
    ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
    ‹*twl-list-invs S*› **and**
    ‹*twl-struct-invs T*›
    ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*)›
  obtains *V* **where**
    ‹(*U*, *V*) ∈ *twl-st-l None*› **and**
    ‹*cdcl-twl-inp*** *T V*›
proof −

```
  have ‹∃ V. (U, V) ∈ twl-st-l None ∧ cdcl-twl-inp** T V›
    using assms(1)
    apply (induction arbitrary: rule: rtranclp-induct)
    subgoal
      by (rule exI[of - T]) (use assms in auto)
    subgoal for x y
      apply normalize-goal+
      apply (rule cdcl-twl-restart-l-inp-cdcl-twl-restart-inp[of ], assumption+)
      apply (smt (verit, ccfv-threshold) assms(3) rtranclp-cdcl-twl-restart-l-inp-twl-list-invs)
      using assms(4) assms(5) rtranclp-cdcl-twl-inp-invs(1) apply blast
      by force
    done
  then show ?thesis
    using that by blast
qed

definition mark-to-delete-clauses-l-post where
  ‹mark-to-delete-clauses-l-post S T ⟷
     (∃ S'. (S, S') ∈ twl-st-l None ∧ remove-one-annot-true-clause** S T ∧
       twl-list-invs S ∧ twl-struct-invs S' ∧ get-conflict-l S = None ∧
       clauses-to-update-l S = {#} ∧ get-unkept-learned-clss-l T = {#} ∧
    get-subsumed-learned-clauses-l T = {#} ∧
    get-learned-clauses0-l T = {#})›

definition cdcl-twl-full-restart-l-prog where
  ‹cdcl-twl-full-restart-l-prog S = do {
  ASSERT(mark-to-delete-clauses-l-pre S);
  T ← mark-to-delete-clauses-l S;
  ASSERT (mark-to-delete-clauses-l-post S T);
  RETURN T
  }›

definition mark-duplicated-binary-clauses-as-garbage-l2 where
  ‹mark-duplicated-binary-clauses-as-garbage-l2 T = do {
    if get-conflict-l T ≠ None then RETURN T
    else mark-duplicated-binary-clauses-as-garbage T}›

definition mark-to-delete-clauses-l-GC-pre
  :: ‹'v twl-st-l ⇒ bool›
where
  ‹mark-to-delete-clauses-l-GC-pre S ⟷
  (∃ T. (S, T) ∈ twl-st-l None ∧ twl-struct-invs T ∧ twl-list-invs S ∧
  set (get-all-mark-of-propagated (get-trail-l S)) ⊆ {0})›

definition cdcl-twl-full-restart-inprocess-l where
  ‹cdcl-twl-full-restart-inprocess-l S = do {
  ASSERT(cdcl-twl-full-restart-l-GC-prog-pre S);
  S' ← cdcl-twl-local-restart-l-spec0 S;
  S' ← remove-one-annot-true-clause-imp S';
  S' ← mark-duplicated-binary-clauses-as-garbage S';
  S' ← forward-subsume-l S';
  S' ← pure-literal-eliminate-l S';
  S' ← simplify-clauses-with-units-st S';
  if (get-conflict-l S' ≠ None) then do {
    ASSERT(cdcl-twl-restart-l-inp** S S');
    RETURN S'
```

```
  }
  else do {
     ASSERT(mark-to-delete-clauses-l-GC-pre S');
     U ← mark-to-delete-clauses-l S';
     V ← cdcl-GC-clauses U;
     ASSERT(cdcl-twl-restart-l-inp** S V);
     RETURN V
   }
  }⟩


definition cdcl-twl-full-restart-l-GC-prog where
  ⟨cdcl-twl-full-restart-l-GC-prog S = do {
    ASSERT(cdcl-twl-full-restart-l-GC-prog-pre S);
    S' ← cdcl-twl-local-restart-l-spec0 S;
    T ← remove-one-annot-true-clause-imp S';
    ASSERT(mark-to-delete-clauses-l-GC-pre T);
    U ← mark-to-delete-clauses-l T;
    V ← cdcl-GC-clauses U;
    ASSERT(cdcl-twl-restart-l S V);
    RETURN V
  }⟩


context twl-restart-ops
begin

lemma cdcl-twl-full-restart-l-prog-spec:
  assumes
    ST: ⟨(S, T) ∈ twl-st-l None⟩ and
    list-invs: ⟨twl-list-invs S⟩ and
    struct-invs: ⟨twl-struct-invs T⟩ and
    confl: ⟨get-conflict-l S = None⟩ and
    upd: ⟨clauses-to-update-l S = {#}⟩
  shows ⟨cdcl-twl-full-restart-l-prog S ≤ ⇓ Id (SPEC(remove-one-annot-true-clause^++ S))⟩
proof −
  have mark-to-delete-clauses-l:
    ⟨mark-to-delete-clauses-l x ≤ SPEC (λT. ASSERT (mark-to-delete-clauses-l-post U T) ⤜
          (λ-. RETURN T)
          ≤ SPEC (remove-one-annot-true-clause^++ U))⟩
    if
      Ux: ⟨(x, U) ∈ Id⟩ and
      U: ⟨U ∈ Collect (remove-one-annot-true-clause** S)⟩
      for x U
  proof −
    from U have SU: ⟨remove-one-annot-true-clause** S U⟩ by simp
    have x: ⟨x = U⟩
      using Ux by auto
    obtain V where
      SU': ⟨cdcl-twl-restart-l** S U⟩ and
      UV: ⟨(U, V) ∈ twl-st-l None⟩ and
      TV: ⟨cdcl-twl-restart** T V⟩ and
      struct-invs-V: ⟨twl-struct-invs V⟩
      using rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2[OF SU list-invs
        confl upd ST struct-invs]
      by auto
```

```
    have
      confl-U: ‹get-conflict-l U = None› and
      upd-U: ‹clauses-to-update-l U = {#}›
      using rtranclp-remove-one-annot-true-clause-get-conflict-l[OF SU]
        rtranclp-remove-one-annot-true-clause-clauses-to-update-l[OF SU] confl upd
      by auto
    have list-U: ‹twl-list-invs U›
      using SU' list-invs rtranclp-cdcl-twl-restart-l-list-invs by blast
    have ‹remove-one-annot-true-clause^++ U V' ⟹
      get-unkept-learned-clss-l V' = {#} ∧
      get-subsumed-learned-clauses-l V' = {#} ∧
      get-learned-clauses0-l V' = {#}› for V'
      by (subst (asm)tranclp-unfold-end)
       (auto simp: remove-one-annot-true-clause.simps)

     then have [simp]:
      ‹remove-one-annot-true-clause^++ U V' ⟹ mark-to-delete-clauses-l-post U V'› for V'
      unfolding mark-to-delete-clauses-l-post-def
      using UV struct-invs-V list-U confl-U upd-U
      by (blast dest: tranclp-into-rtranclp)
    show ?thesis
      unfolding x
      by (rule mark-to-delete-clauses-l-spec[OF UV list-U struct-invs-V confl-U upd-U,
        THEN order-trans])
       (auto intro: RES-refine)
  qed
  have 1: ‹SPEC (remove-one-annot-true-clause** S) = do {
      T ← SPEC (remove-one-annot-true-clause** S);
      SPEC (remove-one-annot-true-clause** T)
    }›
  by (auto simp: RES-RES-RETURN-RES)
  have H: ‹mark-to-delete-clauses-l-pre T›
    if
      ‹(T, U) ∈ Id› and
      ‹U ∈ Collect (remove-one-annot-true-clause** S)›
    for T U
  proof −
    show ?thesis
      using rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2[of S U,
        OF - list-invs confl upd ST struct-invs] that list-invs
        rtranclp-remove-one-annot-true-clause-get-all-mark-of-propagated[of S U]
        rtranclp-cdcl-twl-restart-l-list-invs[of S U]
      unfolding mark-to-delete-clauses-l-pre-def
      by (metis Un-absorb1 mem-Collect-eq pair-in-Id-conv)
  qed
  show ?thesis
    unfolding cdcl-twl-full-restart-l-prog-def
    apply (refine-vcg mark-to-delete-clauses-l)
    subgoal
      using assms
      unfolding mark-to-delete-clauses-l-pre-def
      by blast
    subgoal by auto
    subgoal by (auto simp: assert-bind-spec-conv)
    done
qed
```

definition *GC-required-l* :: *'v twl-st-l* ⇒ *nat* ⇒ *nat* ⇒ *bool nres* **where**
‹*GC-required-l S m n* = *do* {
ASSERT(*size* (*get-all-learned-clss-l S*) ≥ *m*);
SPEC (λ*b*. *b* ⟶ *size* (*get-all-learned-clss-l S*) − *m* > *f n*)
}›

definition *restart-required-l* :: *'v twl-st-l* ⇒ *nat* ⇒ *nat* ⇒ *bool nres* **where**
‹*restart-required-l S m n* = *do* {
ASSERT(*size* (*get-all-learned-clss-l S*) ≥ *m*);
SPEC (λ*b*. *b* ⟶ *size* (*get-all-learned-clss-l S*) > *m*)
}›

definition *inprocessing-required-l* :: *'v twl-st-l* ⇒ *bool nres* **where**
‹*inprocessing-required-l S* = *do* {
SPEC (λ*b*. *True*)
}›

lemma *inprocessing-required-l-inprocessing-required*:
‹(*inprocessing-required-l*, *inprocessing-required*) ∈ *twl-st-l None* →*f* ⟨*bool-rel*⟩ *nres-rel*›
by (*intro frefI nres-relI*) (*auto simp*: *inprocessing-required-l-def inprocessing-required-def*)

definition *restart-abs-l*
:: *'v twl-st-l* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ (*'v twl-st-l* × *nat* × *nat* × *nat*) *nres*
where
‹*restart-abs-l S last-GC last-Restart n brk* = *do* {
ASSERT(*restart-abs-l-pre S last-GC last-Restart brk*);
b ← *GC-required-l S last-GC n*;
b2 ← *restart-required-l S last-Restart n*;
if b2 ∧ ¬*brk then do* {
T ← *SPEC*(λ*T*. *cdcl-twl-restart-only-l S T*);
RETURN (*T*, *last-GC*, *size* (*get-all-learned-clss-l T*), *n*)
}
else
if b ∧ ¬*brk then do* {
b ← *inprocessing-required-l S*;
if ¬*b then do* {
T ← *SPEC*(λ*T*. *cdcl-twl-restart-l S T*);
RETURN (*T*, *size* (*get-all-learned-clss-l T*), *size* (*get-all-learned-clss-l T*), *n* + *1*)
} *else do* {
T ← *SPEC*(λ*T*. *cdcl-twl-restart-l-inp*** *S T* ∧ *count-decided* (*get-trail-l T*) = *0*);
RETURN (*T*, *size* (*get-all-learned-clss-l T*), *size* (*get-all-learned-clss-l T*), *n* + *1*)
}
}
else
RETURN (*S*, *last-GC*, *last-Restart*, *n*)
}›

lemma (**in** −)[*twl-st-l*]:
‹(*S*, *S'*) ∈ *twl-st-l None* ⟹ *get-learned-clss S'* = *twl-clause-of* ‘# (*get-learned-clss-l S*)›
by (*auto simp*: *get-learned-clss-l-def twl-st-l-def*)

lemma *restart-required-l-restart-required*:
‹(*uncurry2 restart-required-l*, *uncurry2 restart-required*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S*} ×*f* *nat-rel* ×*f* *nat-rel* →*f*

⟨*bool-rel*⟩ *nres-rel*›
unfolding *restart-required-l-def restart-required-def uncurry-def*
by (*intro frefI nres-relI*) (*refine-rcg*, *auto simp*: *twl-st-l-def get-learned-clss-l-def*)

lemma *GC-required-l-GC-required*:
‹(*uncurry2 GC-required-l*, *uncurry2 GC-required*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S*} ×$_f$ *nat-rel* ×$_f$ *nat-rel* →$_f$
⟨*bool-rel*⟩ *nres-rel*›
unfolding *GC-required-l-def GC-required-def uncurry-def*
by (*intro frefI nres-relI*) (*refine-rcg*, *auto simp*: *twl-st-l-def get-learned-clss-l-def*)

lemma ‹*size* (*get-learned-clss-l T*) = *size* (*learned-clss-l* (*get-clauses-l T*))›
by (*auto simp*: *get-learned-clss-l-def*)

lemma *restart-abs-l-restart-prog*:
‹(*uncurry4 restart-abs-l*, *uncurry4 restart-prog*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×$_f$ *nat-rel* ×$_f$ *nat-rel* ×$_f$ *nat-rel* ×$_f$ *bool-rel*
→$_f$
⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*⟩ *nres-rel*›
proof −
have [*refine*]: ‹*RETURN T*
≤ ⇓ ({(*S*, *T*). (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None*})
(*SPEC*
(λ*U*. *cdcl-twl-stgy*** *Ta U* ∧
clauses-to-update U = {#} ∧ *get-conflict U* = *None*))›
if ‹(*T*, *Ta*) ∈ *twl-st-l None*› ‹*clauses-to-update-l T* = {#}›
‹*get-conflict-l T* = *None*› ‹*twl-list-invs T*›
for *T Ta*
using *that* **apply** −
apply (*rule RETURN-RES-refine*)
apply (*rule-tac x*=*Ta* **in** *exI*)
apply (*auto intro*!: *RETURN-RES-refine*)
done
have [*refine0*]: ‹*RETURN False* ≤ ⇓ {(*a*,*b*). ¬*a* ∧ ¬*b*} (*inprocessing-required S*)› **for** *S*
by (*auto simp*: *inprocessing-required-def intro*!: *RETURN-RES-refine*)
have *cdcl-twl-restart-l-inp*: ‹(*x*, *y*)
∈ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×$_f$ *nat-rel* ×$_f$
nat-rel ×$_f$
nat-rel ×$_f$
bool-rel ⟹
x1b = (*x1c*, *x2*) ⟹
x1a = (*x1b*, *x2a*) ⟹
x1 = (*x1a*, *x2b*) ⟹
y = (*x1*, *x2c*) ⟹
x1f = (*x1g*, *x2d*) ⟹
x1e = (*x1f*, *x2e*) ⟹
x1d = (*x1e*, *x2f*) ⟹
x = (*x1d*, *x2g*) ⟹
restart-prog-pre x1c x2 x2a x2c ⟹
restart-abs-l-pre x1g x2d x2e x2g ⟹
SPEC (*cdcl-twl-restart-l-inp*** *x1g*)
≤ ⇓{(*S*, *T*). (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}}

(*SPEC* (*cdcl-twl-inp*** *x1c*))›
for *x y x1 x1a x1b x1c x2 x2a x2b x2c x1d x1e x1f x1g x2d x2e x2f x2g b ba b2 b2a bb bc*
supply [[*goals-limit=1*]]
apply (*rule RES-refine*)
apply (*simp only*: *mem-Collect-eq prod.simps prod-rel-iff restart-abs-l-pre-def restart-prog-pre-def*)
apply (*rule rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*)
apply *assumption*+
apply *normalize-goal*+
apply *assumption*+
apply *normalize-goal*+
apply *assumption*+
apply *normalize-goal*+
apply *assumption*+
apply *normalize-goal*+
apply *assumption*+
apply (*rule-tac x*=*V* **in** *bexI*)
apply *simp*
apply (*intro conjI*)
apply (*meson rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*)
apply (*meson rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l*)
apply *simp*
done
have *inprocess-refine*:
‹*SPEC* (λ*T*. *cdcl-twl-restart-l-inp*** *x1g T* ∧ *count-decided* (*get-trail-l T*) = *0*)
≤ ⇓ {(*x*,*y*). (*x*,*y*) ∈ *twl-st-l None* ∧ *twl-list-invs x* ∧ *twl-struct-invs y*}
(*SPEC* (λ*T*. *cdcl-twl-inp*** *x1c T* ∧ *count-decided* (*get-trail T*) = *0*))›
if
True **and**
‹(*x*, *y*)
∈ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×*f* *nat-rel* ×*f* *nat-rel* ×*f* *nat-rel* ×*f* *bool-rel*› **and**
‹*x1b* = (*x1c*, *x2*)› **and**
‹*x1a* = (*x1b*, *x2a*)› **and**
‹*x1* = (*x1a*, *x2b*)› **and**
‹*y* = (*x1*, *x2c*)› **and**
‹*x1f* = (*x1g*, *x2d*)› **and**
‹*x1e* = (*x1f*, *x2e*)› **and**
‹*x1d* = (*x1e*, *x2f*)› **and**
‹*x* = (*x1d*, *x2g*)› **and**
‹*restart-prog-pre x1c x2 x2a x2c*› **and**
‹*restart-abs-l-pre x1g x2d x2e x2g*›
for *x y x1 x1a x1b x1c x2 x2a x2b x2c x1d x1e x1f x1g x2d x2e x2f x2g b ba b2 b2a bb bc*
using *that*
apply (*simp only*: *in-pair-collect-simp prod-rel-iff restart-prog-pre-def*)
apply (*rule RES-refine*)
unfolding *mem-Collect-eq*
apply *normalize-goal*+
apply (*rule rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*)
apply *assumption*+
apply (*rule-tac x*=*V* **in** *bexI*)
apply (*auto intro*: *rtranclp-cdcl-twl-restart-l-inp-twl-list-invs rtranclp-cdcl-twl-inp-twl-struct-invs*)

```
    done
  show ?thesis
   supply [[goals-limit=1]]
    unfolding restart-abs-l-def restart-prog-def uncurry-def
    apply (intro frefI nres-relI)
    apply (refine-rcg
      restart-required-l-restart-required[THEN fref-to-Down-curry2]
      GC-required-l-GC-required[THEN fref-to-Down-curry2]
      cdcl-twl-restart-only-l-cdcl-twl-restart-only
      cdcl-twl-restart-l-cdcl-twl-restart
      cdcl-twl-restart-only-l-cdcl-twl-restart-only-spec
      cdcl-twl-restart-l-inp
      inprocessing-required-l-inprocessing-required[THEN fref-to-Down]
      )
    subgoal for Snb Snb'
      unfolding restart-abs-l-pre-def
      by (rule exI[of - ‹fst (fst (fst (Snb')))›])
        simp
    subgoal by auto
    subgoal by auto
    subgoal by auto  — If condition
    subgoal by simp
    subgoal unfolding restart-prog-pre-def by auto
    subgoal by (auto simp: get-learned-clss-l-def)
    subgoal by auto
    subgoal by auto
    subgoal for x y x1 x1a x1b x1c x2 x2a x2b x2c x1d x1e x1f x1g x2d x2e x2f x2g b ba
      b2 b2a bb bc
      by auto
    subgoal by auto
    subgoal by auto
    subgoal unfolding restart-prog-pre-def by auto
    subgoal by (auto simp: get-learned-clss-l-def)
      apply (rule inprocess-refine; assumption)
    subgoal by (auto simp: get-learned-clss-l-def
        rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l)
    subgoal by (auto simp: get-learned-clss-l-def)
    done
qed


definition cdcl-twl-stgy-restart-abs-l-inv :: ‹'v twl-st-l ⇒ bool × 'v twl-st-l × nat × nat × nat ⇒ bool›
where
  ‹cdcl-twl-stgy-restart-abs-l-inv S₀ ≡ (λ(brk, T, last-GC, last-Restart, n).
    (∃S₀' T' n'.
      (T, T') ∈ twl-st-l None ∧
      (S₀, S₀') ∈ twl-st-l None ∧
      cdcl-twl-stgy-restart-prog-inv (S₀', n') (brk, T', last-GC, last-Restart, n) ∧
      clauses-to-update-l T = {#} ∧
      twl-list-invs T))›

definition cdcl-twl-stgy-restart-abs-l :: 'v twl-st-l ⇒ 'v twl-st-l nres where
  ‹cdcl-twl-stgy-restart-abs-l S₀ =
  do {
    (brk, T, -, -, -) ← WHILE_T^cdcl-twl-stgy-restart-abs-l-inv S₀
      (λ(brk, -). ¬brk)
```

```
    (λ(brk, S, m, p, n).
    do {
      T ← unit-propagation-outer-loop-l S;
      (brk, T) ← cdcl-twl-o-prog-l T;
      (T, m, p, n) ← restart-abs-l T m p n brk;
      RETURN (brk ∨ get-conflict-l T ≠ None, T, m, p, n)
    })
    (False, S₀, size (get-all-learned-clss-l S₀), size (get-all-learned-clss-l S₀), 0);
  RETURN T
}›

lemma (in −)prod-rel-fst-snd-iff: ‹(x, y) ∈ A ×r B ⟷ (fst x, fst y) ∈ A ∧ (snd x, snd y) ∈ B›
  by (cases x; cases y) auto

lemma cdcl-twl-stgy-restart-abs-l-cdcl-twl-stgy-restart-abs-l:
  ‹(cdcl-twl-stgy-restart-abs-l, cdcl-twl-stgy-restart-prog) ∈
    {(S :: 'v twl-st-l, S'). (S, S') ∈ twl-st-l None ∧ twl-list-invs S ∧
      clauses-to-update-l S  = {#}} →f
    ⟨{(S, S'). (S, S') ∈ twl-st-l None ∧ twl-list-invs S}⟩ nres-rel›
  unfolding cdcl-twl-stgy-restart-abs-l-def cdcl-twl-stgy-restart-prog-def uncurry-def
  apply (intro frefI nres-relI)
  apply (refine-rcg WHILEIT-refine[where R =
    ‹bool-rel ×r {(S :: 'v twl-st-l, S'). (S, S') ∈ twl-st-l None ∧ twl-list-invs S ∧
      clauses-to-update-l S  = {#}} ×r nat-rel ×r nat-rel ×r nat-rel›]
    unit-propagation-outer-loop-l-spec[THEN fref-to-Down]
    cdcl-twl-o-prog-l-spec[THEN fref-to-Down]
    restart-abs-l-restart-prog[THEN fref-to-Down-curry4])
  subgoal by (auto simp add: get-learned-clss-l-def)
  subgoal for x y xa x'
    unfolding cdcl-twl-stgy-restart-abs-l-inv-def case-prod-beta
    apply (rule-tac x=y in exI)
    apply (rule-tac x=‹fst (snd x')› in exI)
    apply (rule-tac x=0 in exI)
    by (auto simp: prod-rel-fst-snd-iff)
  subgoal by auto
  subgoal
    by auto
  subgoal
    by auto
  subgoal
    by auto
  subgoal
    by auto
  subgoal
    by auto
  done

end

lemma cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l:
  assumes
    ST: ‹(S, S') ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs S› and
    struct-invs: ‹twl-struct-invs S'› and
```

```
    confl: ‹get-conflict-l S = None› and
    upd: ‹clauses-to-update-l S = {#}› and
    stgy-invs: ‹twl-stgy-invs S'› and
    abs-pre: ‹restart-prog-pre S' last-GC last-Restart brk›
  shows ‹cdcl-twl-full-restart-l-GC-prog S ≤ ⇓ Id (SPEC (λT. cdcl-twl-restart-l S T))›
proof −
  let ?f = ‹(λS T. cdcl-twl-restart-l S T)›
  let ?f1 = ‹λS S'. (?f S S' ∨ S = S') ∧ count-decided (get-trail-l S') = 0›
  let ?f1' = ‹λS S'. (?f S S') ∧ count-decided (get-trail-l S') = 0›
  let ?f2 = ‹λS S'. ?f S S' ∧ (∀ L ∈ set (get-trail-l S'). mark-of L = 0) ∧
    length (get-trail-l S) = length (get-trail-l S')›
  let ?f3 = ‹λS S'. ?f1 S S' ∧ (∀ L ∈ set (get-trail-l S'). mark-of L = 0) ∧
    length (get-trail-l S) = length (get-trail-l S')›
  have n-d: ‹no-dup (get-trail-l S)›
    using struct-invs ST unfolding twl-struct-invs-def cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
      cdcl_W-restart-mset.cdcl_W-M-level-inv-def pcdcl-all-struct-invs-def
    by (simp add: twl-st)
  then have alt-def: ‹SPEC (λT. cdcl-twl-restart-l S T) ≥ do {
    S' ← SPEC (λS'. ?f1 S S');
    T ← SPEC (?f2 S');
    U ← SPEC (?f3 T);
    V ← SPEC (λV. ?f3 U V);
    RETURN V
    }›
    unfolding RETURN-def RES-RES-RETURN-RES apply −
    apply (rule RES-rule)
    unfolding UN-iff
    apply (elim bexE)+
    unfolding mem-Collect-eq
    by (metis (full-types) cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l singletonD)

  have 1: ‹remove-one-annot-true-clause-imp T ≤ SPEC (λV. ?f2 U V)›
    if
      ‹(T, U) ∈ Id› and
      ‹U ∈ Collect (λS'. ?f1 S S')›
    for T U
  proof −
    have ‹T = U› and ‹?f S T ∨ S = T› and count-0: ‹count-decided (get-trail-l T) = 0›
      using that by auto
    have confl: ‹get-conflict-l T = None›
      using ‹?f S T ∨ S = T› confl
      by (auto simp: cdcl-twl-restart-l.simps)
    obtain T' where
      TT': ‹(T, T') ∈ twl-st-l None› and
      list-invs: ‹twl-list-invs T› and
      struct-invs: ‹twl-struct-invs T'› and
      clss-upd: ‹clauses-to-update-l T = {#}› and
      ‹cdcl-twl-restart S' T' ∨ S' = T'›
      using cdcl-twl-restart-l-invs[OF assms(1−3), of T] assms
        ‹?f S T ∨ S = T›
      by blast
   show ?thesis
      unfolding ‹T = U›[symmetric]
      by (rule remove-one-annot-true-clause-imp-spec-lev0[OF TT' list-invs struct-invs confl
          clss-upd, THEN order-trans])
       (use count-0 remove-one-annot-true-clause-cdcl-twl-restart-l-spec[OF TT' list-invs struct-invs
```

confl clss-upd] *n-d* ‹*?f S T* ∨ *S* = *T*›
remove-one-annot-true-clause-map-mark-of-same-or-0[*of T*] **in**
‹*auto dest*: *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
simp: *rtranclp-remove-one-annot-true-clause-count-dec*›)
qed

have *mark-to-delete-clauses-l-pre*: ‹*mark-to-delete-clauses-l-GC-pre U*›
if
‹(*T*, *T'*) ∈ *Id*› **and**
‹*T'* ∈ *Collect* (*?f1 S*)› **and**
‹(*U*, *U'*) ∈ *Id*› **and**
‹*U'* ∈ *Collect* (*?f2 T'*)›
for *T T' U U'*
proof −
have ‹*T* = *T'*› ‹*U* = *U'*› **and** ‹*?f T U*› **and** ‹*?f S T* ∨ *S* = *T*›
using *that* **by** *auto*
then have ‹*?f S U* ∨ *S* = *U*›
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
by *blast*
have *confl*: ‹*get-conflict-l U* = *None*›
using ‹*?f T U*› ‹*?f S T* ∨ *S* = *T*› *confl*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
obtain *U'* **where**
TT': ‹(*U*, *U'*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs U*› **and**
struct-invs: ‹*twl-struct-invs U'*› **and**
clss-upd: ‹*clauses-to-update-l U* = {#}› **and**
‹*cdcl-twl-restart S' U'* ∨ *S'* = *U'*›
using *cdcl-twl-restart-l-invs*[*OF assms*(*1*−*3*), *of U*] ‹*?f S U* ∨ *S* = *U*› *assms that*[*of S'*]
by *blast*
moreover have ‹*set* (*get-all-mark-of-propagated* (*get-trail-l U*)) ⊆ {*0*}›
using *that rtranclp-remove-one-annot-true-clause-get-all-mark-of-propagated*[*of S U*]
apply *simp*
by (*metis annotated-lit.sel*(*3*)
cdcl$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail singletonI subset-code*(*1*))
ultimately show *?thesis*
unfolding *mark-to-delete-clauses-l-GC-pre-def*
by *blast*
qed
have *2*: ‹*mark-to-delete-clauses-l U* ≤ *SPEC* (λ*V*. *?f3 U' V*)›
if
‹(*T*, *T'*) ∈ *Id*› **and**
‹*T'* ∈ *Collect* (*?f1 S*)› **and**
UU': ‹(*U*, *U'*) ∈ *Id*› **and**
U: ‹*U'* ∈ *Collect* (*?f2 T'*)› **and**
pre: ‹*mark-to-delete-clauses-l-GC-pre U*›
for *T T' U U'*
proof −
have ‹*T* = *T'*› ‹*U* = *U'*› **and** ‹*?f T U*› **and** ‹*?f S T* ∨ *S* = *T*›
using *that* **by** *auto*
then have *SU*: ‹*?f S U*›
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
by *blast*
obtain *V* **where**
TV: ‹(*U*, *V*) ∈ *twl-st-l None*› **and**
struct: ‹*twl-struct-invs V*› **and**

list-invs: ‹*twl-list-invs U*›
using *pre* **unfolding** *mark-to-delete-clauses-l-GC-pre-def*
by *auto*
have *confl*: ‹*get-conflict-l U = None*› **and**
upd: ‹*clauses-to-update-l U = {#}*› **and**
UU[*simp*]: ‹*U' = U*›
using *U UU'* ‹*?f T U*› *confl* ‹*?f S T ∨ S = T*› *assms*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
have *annU*: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l U*)) ⊆ {*0*}›
using *that rtranclp-remove-one-annot-true-clause-get-all-mark-of-propagated*[*of S U*]
apply *simp*
by (*metis annotated-lit.sel*(*3*)
cdcl$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail singletonI subset-code*(*1*))
show *?thesis*
by (*rule mark-to-delete-clauses-l-spec*[*OF TV list-invs struct confl upd*, *THEN order-trans*],
subst Down-id-eq)
(*use remove-one-annot-true-clause-cdcl-twl-restart-l-spec*[*OF TV list-invs struct confl upd*]
cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[*OF - - n-d*, *of T*] *that*
ST **in** ‹*auto dest*!: *cdcl-twl-restart-l-count-dec*›)
qed
have *3*: ‹*cdcl-GC-clauses V ≤ SPEC* (*?f3 V'*)›
if
‹(*T*, *T'*) ∈ *Id*› **and**
‹*T' ∈ Collect* (*?f1 S*)› **and**
‹(*U*, *U'*) ∈ *Id*› **and**
‹*U' ∈ Collect* (*?f2 T'*)› **and**
‹*mark-to-delete-clauses-l-GC-pre U*› **and**
‹(*V*, *V'*) ∈ *Id*› **and**
‹*V' ∈ Collect* (*?f3 U'*)›
for *T T' U U' V V'*
proof −
have *eq*: ‹*U' = U*›
using *that* **by** *auto*
have *st*: ‹*T = T'*› ‹*U = U'*› ‹*V = V'*› **and** ‹*?f S T ∨ S = T*› **and** ‹*?f T U*› **and**
‹*?f U V ∨ U = V*› **and**
le-UV: ‹*length* (*get-trail-l U*) = *length* (*get-trail-l V*)› **and**
mark0: ‹∀ *L*∈*set* (*get-trail-l V'*). *mark-of L = 0*› **and**
count-dec: ‹*count-decided* (*get-trail-l V'*) = *0*›
using *that* **by** (*auto dest*!: *cdcl-twl-restart-l-count-dec*)
then have ‹*?f S V*›
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
by *blast*
have *mark*: ‹*mark-of* (*get-trail-l V* ! *i*) = *0*› **if** ‹*i* < *length* (*get-trail-l V*)› **for** *i*
using *that*
by (*use st that le-UV count-dec mark0* **in**
‹*auto simp*: *count-decided-0-iff is-decided-no-proped-iff*›)
then have *count-dec*: ‹*count-decided* (*get-trail-l V'*) = *0*› **and**
mark: ‹⋀*L*. *L* ∈ *set* (*get-trail-l V'*) ⟹ *mark-of L = 0*›
using *cdcl-twl-restart-l-count-dec*[*of U V*] *that* ‹*?f U V ∨ U = V*›
by (*auto dest*!: *cdcl-twl-restart-l-count-dec*)
obtain *W* **where**
UV: ‹(*V*, *W*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs V*› **and**
clss: ‹*clauses-to-update-l V = {#}*› **and**
‹*cdcl-twl-restart S' W*› **and**
struct: ‹*twl-struct-invs W*›

```
    using cdcl-twl-restart-l-invs[OF assms(1,2,3) ‹?f S V›] unfolding eq by blast
  have confl: ‹get-conflict-l V = None›
    using ‹?f S V› unfolding eq
    by (auto simp: cdcl-twl-restart-l.simps)
  show ?thesis
    unfolding eq
    by (rule cdcl-GC-clauses-cdcl-twl-restart-l[OF UV list-invs struct confl clss, THEN order-trans])
     (use count-dec cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[OF - - n-d, of U']
       ‹?f S V› eq mark in ‹auto simp: ‹V = V'››)
qed
have cdcl-twl-restart-l: ‹cdcl-twl-restart-l S W›
  if
    ‹(T, T') ∈ Id› and
    ‹T' ∈ Collect (?f1 S)› and
    ‹(U, U') ∈ Id› and
    ‹U' ∈ Collect (?f2 T')› and
    ‹mark-to-delete-clauses-l-GC-pre U› and
    ‹(V, V') ∈ Id› and
    ‹V' ∈ Collect (?f3 U')› and
    ‹(W, W') ∈ Id› and
    ‹W' ∈ Collect (?f3 V')›
  for T T' U U' V V' W W'
  using n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[of S T U]
    cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[of S U V]
    cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[of S V W] that
  by fast
have abs-pre: ‹restart-abs-l-pre S last-GC last-Restart False›
  using assms unfolding cdcl-twl-full-restart-l-GC-prog-pre-def restart-abs-l-pre-def
    restart-prog-pre-def apply −
  apply (rule exI[of - S'])
  by auto

show ?thesis
  unfolding cdcl-twl-full-restart-l-GC-prog-def
  apply (rule order-trans)
  prefer 2 apply (rule ref-two-step')
  apply (rule alt-def)
  apply refine-rcg
  subgoal
    using assms unfolding cdcl-twl-full-restart-l-GC-prog-pre-def restart-prog-pre-def
    by fastforce
  subgoal
    by (rule cdcl-twl-local-restart-l-spec0-cdcl-twl-restart-l[THEN order-trans, OF abs-pre])
      auto
  subgoal
    by (rule 1)
  subgoal for T T' U U'
    by (rule mark-to-delete-clauses-l-pre)
  subgoal for T T' U U'
    by (rule 2)
  subgoal for T T' U U' V V'
    by (rule 3)
  subgoal for T T' U U' V V' W W'
    by (rule cdcl-twl-restart-l)
  done
qed
```

lemma *cdcl-twl-full-restart-inprocess-l-cdcl-twl-restart-l*:
assumes
ST: ‹$(S, S') \in$ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs S'*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs S'*› **and**
abs-pre: ‹*restart-prog-pre S' last-GC last-Restart brk*›
shows ‹*cdcl-twl-full-restart-inprocess-l S* ≤ ⇓ *Id* (*SPEC* (λ*T*. *cdcl-twl-restart-l-inp*** *S T* ∧ *count-decided* (*get-trail-l T*) = *0*))› (**is** ‹- ≤ ⇓ - *?P*›)
proof −
let *?f* = ‹(λ*S T*. *cdcl-twl-restart-l S T*)›
let *?f1* = ‹λ*S S'*. (*?f S S'* ∨ *S* = *S'*) ∧ *count-decided* (*get-trail-l S'*) = *0*›
let *?f1'* = ‹λ*S S'*. (*?f S S'*) ∧ *count-decided* (*get-trail-l S'*) = *0*›
let *?finp* = ‹λ*S S'*. *cdcl-twl-restart-l-inp*** *S S'* ∧ *count-decided* (*get-trail-l S'*) = *0*
∧ *set* (*get-all-mark-of-propagated* (*get-trail-l S'*)) ⊆ {*0*}›
let *?f2* = ‹λ*S S'*. *?f S S'* ∧ (∀ *L* ∈ *set* (*get-trail-l S'*). *mark-of L* = *0*) ∧
length (*get-trail-l S*) = *length* (*get-trail-l S'*) ∧
set (*get-all-mark-of-propagated* (*get-trail-l S'*)) ⊆ {*0*}›
let *?f3* = ‹λ*S S'*. *?f1 S S'* ∧ (∀ *L* ∈ *set* (*get-trail-l S'*). *mark-of L* = *0*) ∧
length (*get-trail-l S*) = *length* (*get-trail-l S'*)›
have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *struct-invs ST* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def pcdcl-all-struct-invs-def
by (*simp add*: *twl-st*)
then have *alt-def*: ‹*?P* ≥ *do* {
S' ← *SPEC* (*?f1 S*);
T ← *SPEC* (*?f2 S'*);
T ← *SPEC* (*?finp T*);
T ← *SPEC* (*?finp T*);
U ← *SPEC* (*?finp T*);
U ← *SPEC* (*?finp U*);
if(*get-conflict-l U* ≠ *None*) *then*
RETURN U
else do {
U ← *SPEC* (*?f3 U*);
V ← *SPEC* (λ*V*. *?f3 U V*);
RETURN V
}
}›
unfolding *RETURN-def RES-RES-RETURN-RES* **apply** −
apply *refine-vcg*
apply (*metis* (*no-types*, *lifting*) *cdcl-twl-restart-l-inp.intros*(*1*) *converse-rtranclp-into-rtranclp rtranclp-trans singletonD*)
apply *simp*
apply (*elim UN-E*)+
apply (*auto dest*!: *cdcl-twl-restart-l-inp.intros*)
done

have *1*: ‹*remove-one-annot-true-clause-imp T* ≤ *SPEC* (*?f2 T'*)›
if
‹*cdcl-twl-full-restart-l-GC-prog-pre S*› **and**
‹*T'* ∈ *Collect* (*?f1 S*)› **and**

```
    ‹(T, T') ∈ Id›
    for T T' U U'
proof −
  have ‹T = T'› and ‹?f S T ∨ S = T› and
    count-0: ‹count-decided (get-trail-l T') = 0›
    using that by auto
  have confl: ‹get-conflict-l T' = None›
    using ‹?f S T ∨ S = T› confl
    by (auto simp: cdcl-twl-restart-l.simps ‹T = T'›)
  obtain T'' where
    TT': ‹(T', T'') ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs T'› and
    struct-invs: ‹twl-struct-invs T''› and
    clss-upd: ‹clauses-to-update-l T' = {#}› and
    ‹cdcl-twl-restart S' T'' ∨ S' = T''›
    using cdcl-twl-restart-l-invs[OF assms(1−3), of T] assms
      ‹?f S T ∨ S = T› unfolding ‹T = T'›
    by blast

  have ent: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of T'')›
    using ‹cdcl-twl-restart S' T'' ∨ S' = T''› abs-pre cdcl-twl-inp.intros(5)
      cdcl-twl-inp-invs(3) restart-prog-pre-def by blast

  show ?thesis
   unfolding ‹T = T'›
    by (rule remove-one-annot-true-clause-imp-spec-lev0[OF TT' list-invs struct-invs confl
      clss-upd, THEN order-trans])
     (use count-0 remove-one-annot-true-clause-cdcl-twl-restart-l-spec[OF TT' list-invs
      struct-invs - clss-upd] n-d ‹?f S T ∨ S = T› count-0 confl
        remove-one-annot-true-clause-map-mark-of-same-or-0[of T] in
       ‹auto dest: cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l
        simp: rtranclp-remove-one-annot-true-clause-count-dec
      get-all-mark-of-propagated-alt-def›)
qed

have mark-to-delete-clauses-l-pre: ‹mark-to-delete-clauses-l-GC-pre V› (is ?A) and
  2: ‹mark-to-delete-clauses-l V ≤ SPEC (?f3 V')› (is ?B) and
  3: ‹W' ∈ Collect (?f3 V') ⟹ (W,W') ∈ Id ⟹ cdcl-GC-clauses W ≤ SPEC (?f3 W')› (is ‹- ⟹
- ⟹ ?C›)
  and
  cdcl-twl-restart-l: ‹W' ∈ Collect (?f3 V') ⟹ (W,W') ∈ Id ⟹
  X' ∈ Collect (?f3 W') ⟹ (X,X') ∈ Id ⟹ cdcl-twl-restart-l-inp** S X›
  (is ‹- ⟹ - ⟹ - ⟹ - ⟹ ?D›)
  if
    ‹T' ∈ Collect (?f1 S)› and
    ‹U' ∈ Collect (?f2 T')› and
    ‹U_1' ∈ Collect (?finp U')› and
    ‹U_2' ∈ Collect (?finp U_1')› and
    ‹V_0' ∈ Collect (?finp U_2')› and
    ‹V' ∈ Collect (?finp V_0')› and
    confl-U': ‹¬get-conflict-l V' ≠ None› and
    ‹(V, V') ∈ Id› and
    ‹(T, T') ∈ Id› and
    ‹(U, U') ∈ Id› and
    ‹(U_1, U_1') ∈ Id› and
    ‹(U_2, U_2') ∈ Id›
```

for $T\ T'\ U\ U'\ V\ V'\ W\ W'\ X\ X'\ U_0\ U_0'\ V_0'\ U_1'\ U_1\ U_2\ U_2'$
proof −
have ‹$T = T'$› ‹$U{=}U'$› ‹$V'{=}V$› ‹$U_1' = U_1$› ‹$U_2' = U_2$› **and** ‹*?f S T* ∨ $S = T$› **and**
count-0: ‹*count-decided* (*get-trail-l T*) = *0*› **and**
$T'U'$: ‹*cdcl-twl-restart-l T' U'*› **and**
count-0-V: ‹*count-decided* (*get-trail-l V'*) = *0*› **and**
confl-V': ‹*get-conflict-l V'* = *None*› **and**
UV: ‹*cdcl-twl-restart-l-inp*** *U V*› **and**
‹∀ *L*∈*set* (*get-trail-l U'*). *mark-of L* = *0*› **and**
mark: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l V'*)) ⊆ {*0*}›
using *that* **by** *auto*
have *confl*: ‹*get-conflict-l T* = *None*›
using ‹*?f S T* ∨ $S = T$› *confl*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
obtain T'' **where**
TT': ‹(T', T'') ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs T'*› **and**
struct-invs: ‹*twl-struct-invs T''*› **and**
clss-upd: ‹*clauses-to-update-l T'* = {#}› **and**
‹*cdcl-twl-restart S' T''* ∨ $S' = T''$›
using *cdcl-twl-restart-l-invs*[*OF assms*(*1*−*3*), *of T*] *assms*
‹*?f S T* ∨ $S = T$› **unfolding** ‹$T = T'$›
by *blast*

have *ent*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T''*)›
using ‹*cdcl-twl-restart S' T''* ∨ $S' = T''$› *abs-pre cdcl-twl-inp.intros*(*5*)
cdcl-twl-inp-invs(*3*) *restart-prog-pre-def* **by** *blast*

obtain U'' **where**
UU'': ‹(U, U'') ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs U*› **and**
‹*clauses-to-update-l U* = {#}› **and**
‹*cdcl-twl-restart T'' U''*› **and**
struct-invs: ‹*twl-struct-invs U''*›
using *cdcl-twl-restart-l-invs*[*OF TT' list-invs struct-invs T'U'*] **unfolding** ‹$U{=}U'$›
by *blast*
have *ent*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of U''*)›
by (*metis* ‹*cdcl-twl-restart T'' U''*› *cdcl-twl-restart-entailed-init ent state*$_W$*-of-def*)

obtain V'' **where**
VV'': ‹(V, V'') ∈ *twl-st-l None*› **and**
$U''V''$: ‹*cdcl-twl-inp*** *U'' V''*›
using *rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*[*OF UV UU'' list-invs struct-invs ent*] **unfolding** ‹$V'{=}V$›
by *blast*
then have
list-invs: ‹*twl-list-invs V*› **and**
struct-invs: ‹*twl-struct-invs V''*› **and**
clss-upd: ‹*clauses-to-update-l V* = {#}›
using $T'U'$ *ent UV* ‹$V' = V$› *list-invs rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
‹*cdcl-twl-inp*** *U'' V''*› *ent* ‹*clauses-to-update-l U* = {#}› *struct-invs*
rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l[*OF UV*]
by (*blast intro*: *rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
rtranclp-cdcl-twl-inp-invs)+
have *H*: ‹*L* ∈ *set* (*get-trail-l V'*) ⟹ *mark-of L* = *0*› **for** *L*
using *mark count-0-V* **by** (*cases L*) (*auto dest*!: *split-list*)

have *annV*: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l V*)) ⊆ {*0*}›
using *that rtranclp-remove-one-annot-true-clause-get-all-mark-of-propagated*[*of S V*]
by *simp*
then show *?A*
using *VV'' U''V'' mark list-invs struct-invs clss-upd*
unfolding *mark-to-delete-clauses-l-GC-pre-def* ‹*V'*=*V*› ‹*V'*=*V*›
by *blast*
have *confl*: ‹*get-conflict-l V* = *None*›
using *U''V'' VV'' confl-V' UU''*
by (*auto simp*: *cdcl-twl-restart.simps* ‹*U*=*U'*› ‹*V'*=*V*›
twl-st-l-def)

show *?B*
unfolding ‹*V'*=*V*›
by (*rule mark-to-delete-clauses-l-spec*[*OF VV'' list-invs struct-invs confl clss-upd*,
THEN order-trans], *subst Down-id-eq*)
(*use confl remove-one-annot-true-clause-cdcl-twl-restart-l-spec*[*OF VV'' list-invs*
struct-invs - clss-upd] *H*
cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[*OF - - n-d*, *of T*] *that*
ST **in** ‹*auto dest*!: *cdcl-twl-restart-l-count-dec simp*: ›)
assume *W'*: ‹*W'* ∈ *Collect* (*?f3 V'*)› **and**
‹(*W*,*W'*) ∈ *Id*›
then have ‹*W'* = *W*› **and**
VW: ‹*?f V W* ∨ *V* = *W*› **and**
mark-W: ‹∀ *L*∈*set* (*get-trail-l W*). *mark-of L* = *0*›
using ‹*V'*=*V*›
by *auto*

obtain *W''* **where**
VW'': ‹(*W*, *W''*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs W*› **and**
upd: ‹*clauses-to-update-l W* = {#}› **and**
U''W'': ‹*cdcl-twl-restart V'' W''* ∨ *V* = *W*› **and**
struct-invs: ‹*twl-struct-invs W''*›
using *cdcl-twl-restart-l-invs*[*OF VV'' list-invs struct-invs*, *of W*]
list-invs struct-invs clss-upd VV''
VW **unfolding** ‹*W'* = *W*›
by *blast*
have *confl-W*: ‹*get-conflict-l W* = *None*› **and**
count-0-W: ‹*count-decided* (*get-trail-l W*) = *0*›
by (*use confl U''W'' VW'' VV'' count-0-V* **in**
‹*auto simp*: ‹*W'* = *W*› ‹*V'*=*V*› *twl-st-l-def cdcl-twl-restart.simps*›)[]
(*use VW VV'' VW'' count-0-V* **in** ‹*auto dest*!: *cdcl-twl-restart-l-count-dec*
simp: ‹*W'* = *W*› ‹*V'*=*V*› ›)

show *?C*
unfolding ‹*W'* = *W*›
by (*rule cdcl-GC-clauses-cdcl-twl-restart-l*[*OF VW'' list-invs struct-invs confl-W upd*,
THEN order-trans])
(*use count-0-W mark-W*
cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[*OF - - n-d*, *of U'*]
in ‹*auto simp*: ‹*V'* = *V*››)

assume *W'*: ‹*X'* ∈ *Collect* (*?f3 W'*)› **and**
‹(*X*,*X'*) ∈ *Id*›
then have ‹*X'* = *X*› **and**

```
    WX: ‹?f W X ∨ W = X›
    using ‹V'=V› ‹W'=W›
    by auto

  show ‹cdcl-twl-restart-l-inp** S X›
    using that WX VW unfolding mem-Collect-eq
    by (auto dest!: cdcl-twl-restart-l-inp.intros)
 qed

 have abs-l-pre: ‹restart-abs-l-pre S last-GC last-Restart False›
   using assms unfolding restart-abs-l-pre-def
     restart-prog-pre-def apply -
   apply (rule exI[of - S'])
   by auto

 have  mark-duplicated-binary-clauses-as-garbage:
   ‹mark-duplicated-binary-clauses-as-garbage U ≤ SPEC (?finp U')›
   if
     pre: ‹cdcl-twl-full-restart-l-GC-prog-pre S› and
     ‹T' ∈ Collect (?f1 S)›
     ‹U' ∈ Collect (?f2 T')› and
     ‹(T, T') ∈ Id› and
     ‹(U, U') ∈ Id›
  for T T' U U'
proof -
   have st: ‹T = T'› ‹U=U'› and ‹?f S T ∨ S = T› and
     count-0: ‹count-decided (get-trail-l T) = 0› and
     T'U': ‹cdcl-twl-restart-l T' U'› and
     mark: ‹∀ L∈set (get-trail-l U'). mark-of L = 0› and
     lev0: ‹count-decided (get-trail-l T) = 0›
     using that by auto
   have confl: ‹get-conflict-l T = None›
     using ‹?f S T ∨ S = T› confl
     by (auto simp: cdcl-twl-restart-l.simps)
   obtain T'' where
     TT': ‹(T', T'') ∈ twl-st-l None› and
     list-invs: ‹twl-list-invs T'› and
     struct-invs: ‹twl-struct-invs T''› and
     clss-upd: ‹clauses-to-update-l T' = {#}› and
     ‹cdcl-twl-restart S' T'' ∨ S' = T''›
     using cdcl-twl-restart-l-invs[OF assms(1−3), of T] assms
       ‹?f S T ∨ S = T› unfolding ‹T = T'›
     by blast

   have ent: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of T'')›
     using ‹cdcl-twl-restart S' T'' ∨ S' = T''› abs-pre cdcl-twl-inp.intros(5)
       cdcl-twl-inp-invs(3) restart-prog-pre-def by blast

   obtain U'' where
     UU'': ‹(U, U'') ∈ twl-st-l None› and
     list-invs: ‹twl-list-invs U› and
     clss: ‹clauses-to-update-l U = {#}› and
     T''U'': ‹cdcl-twl-restart T'' U''› and
     struct-invs: ‹twl-struct-invs U''›
     using cdcl-twl-restart-l-invs[OF TT' list-invs struct-invs T'U'] unfolding ‹U=U'›
     by blast
```

```
  have ent: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of U'')›
    by (metis ‹cdcl-twl-restart T'' U''› cdcl-twl-restart-entailed-init ent state_W-of-def)
  have lev0: ‹count-decided (get-trail-l U) = 0›
    using T''U''
    by (metis T'U' cdcl-twl-restart-l-count-dec le-zero-eq lev0 st(1) st(2))

  have confl: ‹get-conflict-l U = None›
    using upd UU'' clss T'U'
    by (auto simp add: cdcl-twl-full-restart-l-GC-prog-pre-def
      cdcl-twl-restart-l.simps st twl-st-l-def)
  have pre: ‹mark-duplicated-binary-clauses-as-garbage-pre U›
    using ent clss lev0 mark apply -
    unfolding mark-duplicated-binary-clauses-as-garbage-pre-def ‹U= U'›[symmetric]
    by (rule exI[of - U''])
     (auto 4 3 simp: UU'' clss-upd list-invs struct-invs
      cdcl_W-restart-mset.in-get-all-mark-of-propagated-in-trail)
  show ?thesis
    apply (rule mark-duplicated-binary-clauses-as-garbage[THEN order-trans])
    apply (rule pre)
    subgoal
      apply simp
      apply standard
      apply simp
        by (metis Un-absorb1 mark-duplicated-binary-clauses-as-garbage-pre-def pre
          rtranclp-cdcl-twl-inprocessing-l-cdcl-twl-l-inp
          rtranclp-cdcl-twl-inprocessing-l-count-decided
          rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated st(2))
    done
qed


have simplify-clauses-with-unit-st:
  ‹simplify-clauses-with-units-st V ≤ SPEC (?finp V')›
  if
    pre: ‹cdcl-twl-full-restart-l-GC-prog-pre S› and
    ‹T' ∈ Collect (?f1 S)›
    ‹U' ∈ Collect (?f2 T')›
    ‹U_2' ∈ Collect (?finp U')›and
    ‹U_3' ∈ Collect (?finp U_2')›and
    ‹V' ∈ Collect (?finp U_3')›and
    ‹(T, T') ∈ Id› and
    ‹(U, U') ∈ Id› and
    ‹(V, V') ∈ Id› and
    ‹(U_2, U_2') ∈ Id›and
    ‹(U_3, U_3') ∈ Id›
  for T T' U U' V V' U_2' U_2 U_3 U_3'
proof -
  have st: ‹T = T'› ‹U=U'›  ‹V'=V› ‹U_2' = U_2› and ‹?f S T ∨ S = T› and
    count-0: ‹count-decided (get-trail-l T) = 0› and
    T'U': ‹cdcl-twl-restart-l T' U'› and
    mark: ‹∀ L∈set (get-trail-l U'). mark-of L = 0› and
    lev0: ‹count-decided (get-trail-l T) = 0› and
    UV': ‹cdcl-twl-restart-l-inp** U' V'› and
    count-dec: ‹count-decided (get-trail-l V') = 0› and
    annot-V: ‹set (get-all-mark-of-propagated (get-trail-l V')) ⊆ {0}›
    using that by auto
```

have *confl*: ‹*get-conflict-l T = None*›
  using ‹*?f S T ∨ S = T*› *confl*
  by (*auto simp*: *cdcl-twl-restart-l.simps*)
obtain *T''* **where**
  TT': ‹(*T'*, *T''*) ∈ *twl-st-l None*› **and**
  list-invs: ‹*twl-list-invs T'*› **and**
  struct-invs: ‹*twl-struct-invs T''*› **and**
  clss-upd: ‹*clauses-to-update-l T' = {#}*› **and**
  ‹*cdcl-twl-restart S' T'' ∨ S' = T''*›
  using *cdcl-twl-restart-l-invs*[*OF assms*(*1*−*3*), *of T*] *assms*
    ‹*?f S T ∨ S = T*› **unfolding** ‹*T = T'*›
  by *blast*

have *ent*: ‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of T''*)›
  using ‹*cdcl-twl-restart S' T'' ∨ S' = T''*› *abs-pre cdcl-twl-inp.intros*(*5*)
    cdcl-twl-inp-invs(*3*) *restart-prog-pre-def* **by** *blast*

obtain *U''* **where**
  UU'': ‹(*U'*, *U''*) ∈ *twl-st-l None*› **and**
  list-invs: ‹*twl-list-invs U'*› **and**
  clss: ‹*clauses-to-update-l U' = {#}*› **and**
  T''U'': ‹*cdcl-twl-restart T'' U''*› **and**
  struct-invs: ‹*twl-struct-invs U''*›
  using *cdcl-twl-restart-l-invs*[*OF TT' list-invs struct-invs T'U'*] **unfolding** ‹*U=U'*›
  by *blast*
have *ent-U''*: ‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of U''*)›
  by (*metis T''U'' cdcl-twl-restart-entailed-init ent state$_W$-of-def*)
then obtain *V''* **where**
  VV'': ‹(*V'*, *V''*) ∈ *twl-st-l None*› **and**
  U''V'': ‹*cdcl-twl-inp*** *U'' V''*›
  using *rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*[*OF UV' UU'' list-invs struct-invs*]
  by *blast*
then have
  list-invs: ‹*twl-list-invs V'*› **and**
  clss-upd: ‹*clauses-to-update-l V' = {#}*› **and**
  struct-invs: ‹*twl-struct-invs V''*›
  using *T'U' ent UV'* ‹*V' = V*› *list-invs rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
    ‹*cdcl-twl-inp*** *U'' V''*› *ent* ‹*clauses-to-update-l U' = {#}*› *struct-invs*
    rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l[*OF UV'*]
    rtranclp-cdcl-twl-inp-invs[*OF U''V''*] *ent-U''*
    unfolding ‹*V' = V*›
  by (*blast intro*: *rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
    rtranclp-cdcl-twl-inp-invs)+

have *ent-V''*: ‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of V''*)›
  using *ent-U'' U''V'' rtranclp-cdcl-twl-inp-entailed-init struct-invs*
  using *T''U''* ‹*cdcl-twl-restart S' T'' ∨ S' = T''*› *assms*(*3*) *cdcl-twl-restart-twl-struct-invs* **by** *blast*
have *pre*: ‹*mark-duplicated-binary-clauses-as-garbage-pre V*›
  using *annot-V count-dec ent-V''* **apply** −
  unfolding *mark-duplicated-binary-clauses-as-garbage-pre-def* ‹*V' = V*›[*symmetric*]
  by (*rule exI*[*of - V''*])
   (*auto simp*: *VV'' clss-upd list-invs struct-invs*)

show *?thesis*
  apply (*rule simplify-clauses-with-units-st-spec*[*THEN order-trans*, *of - V''*])
  apply (*use lev0 UU'' struct-invs list-invs confl clss ent annot-V count-dec ent-V''*

```
      ‹V' = V›[symmetric] clss-upd VV''
    in auto)[7]
  by (use mark count-dec annot-V in ‹auto 4 4 dest: rtranclp-cdcl-twl-inprocessing-l-cdcl-twl-l-inp
    rtranclp-cdcl-twl-inprocessing-l-count-dec
    simp: st lev0 get-all-mark-of-propagated-alt-def
    simplify-clauses-with-unit-st-inv-def simp flip: ‹U=U'› ‹V' = V››)
qed

have forward-subsumption-all:
  ‹forward-subsume-l V ≤ SPEC (?finp V')›
  if
    pre: ‹cdcl-twl-full-restart-l-GC-prog-pre S› and
    ‹T' ∈ Collect (?f1 S)› and
    ‹U' ∈ Collect (?f2 T')› and
    ‹V' ∈ Collect (?finp U')›and
    ‹(T, T') ∈ Id› and
    ‹(U, U') ∈ Id›and
    ‹(V, V') ∈ Id›
  for T T' U U' V V'
proof −
  have st: ‹T = T'› ‹U=U'› ‹V'=V› and ‹?f S T ∨ S = T› and
    count-0: ‹count-decided (get-trail-l T) = 0› and
    T'U': ‹cdcl-twl-restart-l T' U'› and
    mark: ‹∀ L∈set (get-trail-l U'). mark-of L = 0› and
    lev0: ‹count-decided (get-trail-l T) = 0› and
    UV': ‹cdcl-twl-restart-l-inp** U' V'› and
    count-dec: ‹count-decided (get-trail-l V') = 0› and
    annot-V: ‹set (get-all-mark-of-propagated (get-trail-l V')) ⊆ {0}›
    using that by auto
  have confl: ‹get-conflict-l T = None›
    using ‹?f S T ∨ S = T› confl
    by (auto simp: cdcl-twl-restart-l.simps)
  obtain T'' where
    TT': ‹(T', T'') ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs T'› and
    struct-invs: ‹twl-struct-invs T''› and
    clss-upd: ‹clauses-to-update-l T' = {#}› and
    ‹cdcl-twl-restart S' T'' ∨ S' = T''›
    using cdcl-twl-restart-l-invs[OF assms(1−3), of T] assms
      ‹?f S T ∨ S = T› unfolding ‹T = T'›
    by blast

  have ent: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of T'')›
    using ‹cdcl-twl-restart S' T'' ∨ S' = T''› abs-pre cdcl-twl-inp.intros(5)
      cdcl-twl-inp-invs(3) restart-prog-pre-def by blast

  obtain U'' where
    UU'': ‹(U', U'') ∈ twl-st-l None› and
    list-invs: ‹twl-list-invs U'› and
    clss: ‹clauses-to-update-l U' = {#}› and
    T''U'': ‹cdcl-twl-restart T'' U''› and
    struct-invs: ‹twl-struct-invs U''›
    using cdcl-twl-restart-l-invs[OF TT' list-invs struct-invs T'U'] unfolding ‹U=U'›
    by blast
  have ent-U'': ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of U'')›
    by (metis T''U'' cdcl-twl-restart-entailed-init ent state_W-of-def)
```

then obtain V'' **where**
VV'': ‹$(V', V'') \in$ *twl-st-l None*› **and**
$U''V''$: ‹*cdcl-twl-inp*** U'' V''›
using *rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*[*OF UV' UU'' list-invs struct-invs*]
by *blast*
then have
list-invs: ‹*twl-list-invs* V'› **and**
clss-upd: ‹*clauses-to-update-l* $V' = \{\#\}$› **and**
struct-invs: ‹*twl-struct-invs* V''›
using $T'U'$ *ent* UV' ‹$V' = V$› *list-invs rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
‹*cdcl-twl-inp*** U'' V''› *ent* ‹*clauses-to-update-l* $U' = \{\#\}$› *struct-invs*
rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l[*OF UV'*]
rtranclp-cdcl-twl-inp-invs[*OF U''V''*] *ent-U''*
unfolding ‹$V' = V$›
by (*blast intro*: *rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
rtranclp-cdcl-twl-inp-invs)+

have *ent-V''*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* V'')›
using *ent-U''* $U''V''$ *rtranclp-cdcl-twl-inp-entailed-init struct-invs*
using $T''U''$ ‹*cdcl-twl-restart* S' T'' ∨ $S' = T''$› *assms*(3) *cdcl-twl-restart-twl-struct-invs* **by** *blast*

have *pre*: ‹*forward-subsumption-all-pre* V›
unfolding *forward-subsumption-all-pre-def*
using *annot-V count-dec ent-V''* **apply** −
unfolding *mark-duplicated-binary-clauses-as-garbage-pre-def* ‹$V' = V$›[*symmetric*]
by (*rule exI*[*of* - V''])
(*auto simp*: VV'' *clss-upd list-invs struct-invs*)

show *?thesis*
by (*rule forward-subsume-l*[*THEN order-trans*, *of*])
(*use lev0 UU'' struct-invs list-invs confl clss ent annot-V count-dec ent-V''*
‹$V' = V$›[*symmetric*] *clss-upd* VV'' *pre*
in ‹*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-cdcl-twl-l-inp*
rtranclp-cdcl-twl-inprocessing-l-count-decided
rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated›)
qed

have *pure-literal-elimination-round*:
‹*pure-literal-eliminate-l* $V \leq$ *SPEC* (*?finp* V')›
if
pre: ‹*cdcl-twl-full-restart-l-GC-prog-pre* S› **and**
‹$T' \in$ *Collect* (*?f1* S)›
‹$U' \in$ *Collect* (*?f2* T')›
‹$U_1' \in$ *Collect* (*?finp* U')›
‹$V' \in$ *Collect* (*?finp* U_1')›
and
‹$(T, T') \in$ *Id*› **and**
‹$(U, U') \in$ *Id*›**and**
‹$(U_1, U_1') \in$ *Id*›**and**
‹$(V, V') \in$ *Id*›
for T T' U U' V V' U_1 U_1'
proof −
have *st*: ‹$T = T'$› ‹$U = U'$› ‹$V' = V$› ‹$U_1' = U_1$› **and** ‹*?f* S T ∨ $S = T$› **and**
count-0: ‹*count-decided* (*get-trail-l* T) = 0› **and**
$T'U'$: ‹*cdcl-twl-restart-l* T' U'› **and**
mark: ‹∀ $L \in$ *set* (*get-trail-l* U'). *mark-of* $L = 0$› **and**

lev0: ‹*count-decided* (*get-trail-l* *T*) = *0*› **and**
UV': ‹*cdcl-twl-restart-l-inp*** *U'* *V'*› **and**
count-dec: ‹*count-decided* (*get-trail-l* *V'*) = *0*› **and**
annot-V: ‹*set* (*get-all-mark-of-propagated* (*get-trail-l* *V'*)) ⊆ {*0*}›
using *that* **by** *auto*
have *confl*: ‹*get-conflict-l* *T* = *None*›
using ‹*?f* *S* *T* ∨ *S* = *T*› *confl*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
obtain *T''* **where**
TT': ‹(*T'*, *T''*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs* *T'*› **and**
struct-invs: ‹*twl-struct-invs* *T''*› **and**
clss-upd: ‹*clauses-to-update-l* *T'* = {#}› **and**
‹*cdcl-twl-restart* *S'* *T''* ∨ *S'* = *T''*›
using *cdcl-twl-restart-l-invs*[*OF* *assms*(*1*−*3*), *of* *T*] *assms*
‹*?f* *S* *T* ∨ *S* = *T*› **unfolding** ‹*T* = *T'*›
by *blast*

have *ent*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* *T''*)›
using ‹*cdcl-twl-restart* *S'* *T''* ∨ *S'* = *T''*› *abs-pre* *cdcl-twl-inp.intros*(*5*)
cdcl-twl-inp-invs(*3*) *restart-prog-pre-def* **by** *blast*

obtain *U''* **where**
UU'': ‹(*U'*, *U''*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs* *U'*› **and**
clss: ‹*clauses-to-update-l* *U'* = {#}› **and**
T''U'': ‹*cdcl-twl-restart* *T''* *U''*› **and**
struct-invs: ‹*twl-struct-invs* *U''*›
using *cdcl-twl-restart-l-invs*[*OF* *TT'* *list-invs* *struct-invs* *T'U'*] **unfolding** ‹*U*=*U'*›
by *blast*
have *ent-U''*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* *U''*)›
by (*metis* *T''U''* *cdcl-twl-restart-entailed-init* *ent* *state*$_W$*-of-def*)
then obtain *V''* **where**
VV'': ‹(*V'*, *V''*) ∈ *twl-st-l None*› **and**
U''V'': ‹*cdcl-twl-inp*** *U''* *V''*›
using *rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*[*OF* *UV'* *UU''* *list-invs* *struct-invs*]
by *blast*
then have
list-invs: ‹*twl-list-invs* *V'*› **and**
clss-upd: ‹*clauses-to-update-l* *V'* = {#}› **and**
struct-invs: ‹*twl-struct-invs* *V''*›
using *T'U'* *ent* *UV'* ‹*V'* = *V*› *list-invs* *rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
‹*cdcl-twl-inp*** *U''* *V''*› *ent* ‹*clauses-to-update-l* *U'* = {#}› *struct-invs*
rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l[*OF* *UV'*]
rtranclp-cdcl-twl-inp-invs[*OF* *U''V''*] *ent-U''*
unfolding ‹*V'* = *V*›
by (*blast intro*: *rtranclp-cdcl-twl-restart-l-inp-twl-list-invs*
rtranclp-cdcl-twl-inp-invs)+

have *ent-V''*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* *V''*)›
using *ent-U''* *U''V''* *rtranclp-cdcl-twl-inp-entailed-init* *struct-invs*
using *T''U''* ‹*cdcl-twl-restart* *S'* *T''* ∨ *S'* = *T''*› *assms*(*3*) *cdcl-twl-restart-twl-struct-invs* **by** *blast*

have *pre*: ‹*pure-literal-elimination-l-pre* *V*›
unfolding *pure-literal-elimination-l-pre-def*
using *annot-V* *count-dec* *ent-V''* **apply** −

unfolding *mark-duplicated-binary-clauses-as-garbage-pre-def* ‹$V' = V$›[*symmetric*]
by (*rule exI*[*of* - V''])
(*auto simp*: VV'' *clss-upd list-invs struct-invs*)

show *?thesis*
by (*rule pure-literal-eliminate-l*[*THEN order-trans*, *of*])
(*use lev0* UU'' *struct-invs list-invs confl clss ent annot-V count-dec ent-*V''
‹$V' = V$›[*symmetric*] *clss-upd* VV'' *pre*
in ‹*auto dest*: *rtranclp-cdcl-twl-inprocessing-l-cdcl-twl-l-inp*
rtranclp-cdcl-twl-inprocessing-l-count-decided
rtranclp-cdcl-twl-inprocessing-l-get-all-mark-of-propagated›)
qed

show *?thesis*
unfolding *cdcl-twl-full-restart-inprocess-l-def*
apply (*rule order-trans*)
prefer *2* **apply** (*rule ref-two-step'*)
apply (*rule alt-def*)
apply *refine-rcg*
subgoal
using *assms* **unfolding** *cdcl-twl-full-restart-l-GC-prog-pre-def restart-prog-pre-def*
by *fastforce*
subgoal
by (*rule cdcl-twl-local-restart-l-spec0-cdcl-twl-restart-l*[*THEN order-trans*, *OF abs-l-pre*])
auto
subgoal for T T'
by (*rule 1*)
subgoal
by (*rule mark-duplicated-binary-clauses-as-garbage*)
subgoal
by (*rule forward-subsumption-all*)
subgoal
by (*rule pure-literal-elimination-round*)
subgoal for T T' U U'
by (*rule simplify-clauses-with-unit-st*)
subgoal by *auto*
subgoal
by (*auto 5 3 dest*: *cdcl-twl-restart-l-inp.intros*)
subgoal
by (*rule mark-to-delete-clauses-l-pre*)
subgoal for U U'
by (*rule 2*)
subgoal for U U' V V' W W'
by (*rule 3*)
subgoal for U U' V V' W W'
by (*rule cdcl-twl-restart-l*)
done
qed

context *twl-restart-ops*
begin

definition *restart-prog-l*
:: $'v$ *twl-st-l* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* $\Rightarrow$ ($'v$ *twl-st-l* $\times$ *nat* $\times$ *nat* $\times$ *nat*) *nres*
where

‹*restart-prog-l S last-GC last-Restart n brk* = *do* {
ASSERT(*restart-abs-l-pre S last-GC last-Restart brk*);
b ← *GC-required-l S last-GC n*;
b2 ← *restart-required-l S last-Restart n*;
if b2 ∧ ¬*brk then do* {
T ← *cdcl-twl-restart-l-prog S*;
RETURN (*T*, *last-GC*, *size* (*get-all-learned-clss-l T*), *n*)
}
else if b ∧ ¬*brk then do* {
inp ← *inprocessing-required-l S*;
if ¬*inp then do* {
b ← *SPEC*(λ-. *True*);
T ← (*if b then cdcl-twl-full-restart-l-prog S else cdcl-twl-full-restart-l-GC-prog S*);
RETURN (*T*, *size* (*get-all-learned-clss-l T*), *size* (*get-all-learned-clss-l T*), *n* + *1*)
}
else do {
T ← *cdcl-twl-full-restart-inprocess-l S*;
RETURN (*T*, *size* (*get-all-learned-clss-l T*), *size* (*get-all-learned-clss-l T*), *n* + *1*)
}
}
else
RETURN (*S*, *last-GC*, *last-Restart*, *n*)
}›

lemma *restart-prog-l-alt-def*:
‹*restart-prog-l S last-GC last-Restart n brk* = *do* {
ASSERT(*restart-abs-l-pre S last-GC last-Restart brk*);
b ← *GC-required-l S last-GC n*;
b2 ← *restart-required-l S last-Restart n*;
if b2 ∧ ¬*brk then do* {
T ← *cdcl-twl-restart-l-prog S*;
RETURN (*T*, *last-GC*, *size* (*get-all-learned-clss-l T*), *n*)
}
else if b ∧ ¬*brk then do* {
inp ← *inprocessing-required-l S*;
if ¬*inp then do* {
b ← *SPEC*(λ-. *True*);
T ← (*if b then cdcl-twl-full-restart-l-prog S else cdcl-twl-full-restart-l-GC-prog S*);
RETURN (*T*, *size* (*get-all-learned-clss-l T*), *size* (*get-all-learned-clss-l T*), *n* + *1*)
}
else do {
T ← *cdcl-twl-full-restart-inprocess-l S*;
RETURN (*T*, *size* (*get-all-learned-clss-l T*), *size* (*get-all-learned-clss-l T*), *n* + *1*)
}
}
else
RETURN (*S*, *last-GC*, *last-Restart*, *n*)
}›
by (*auto simp*: *restart-prog-l-def cong*: *if-cong*)

lemma *restart-prog-l-restart-abs-l*:
‹(*uncurry4 restart-prog-l*, *uncurry4 restart-abs-l*)
∈ {(*S*:: *'v twl-st-l*, *S'*). (*S*, *S'*) ∈ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×$_f$ *nat-rel* ×$_f$ *nat-rel*×$_f$ *nat-rel* ×$_f$ *bool-rel* →$_f$
⟨{(*S*:: *'v twl-st-l*, *S'*). (*S*, *S'*) ∈ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*⟩*nres-rel*› (**is** ‹- ∈ *?R* ×$_f$ *nat-rel* ×$_f$ *nat-rel*×$_f$ *nat-rel* ×$_f$ *bool-rel* →$_f$ -›)

proof −
have *cdcl-twl-full-restart-l-prog*:
‹*cdcl-twl-full-restart-l-prog S* ≤ *SPEC* (λ*T*. *cdcl-twl-restart-l S T*)›
if
inv: ‹*restart-abs-l-pre S last-GC last-Restart brk*› **and**
‹(*b*, *ba*) ∈ *bool-rel*› **and**
‹*b* ∈ {*b*. *b* ⟶ *f n* < *size* (*S*)}› **and**
‹*ba* ∈ {*b*. *b* ⟶ *f n* < *size* (*S*)}› **and**
brk: ‹¬*brk*›
for *b ba S brk n last-GC last-Restart*
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*›
using *inv brk* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st*)
show *?thesis*
using *cdcl-twl-full-restart-l-prog-spec*[*OF ST list-invs struct-invs*
confl upd]
remove-one-annot-true-clause-cdcl-twl-restart-l-spec[*OF ST list-invs struct-invs*
confl upd]
by (*rule conc-trans-additional*)
qed
have *cdcl-twl-full-restart-l-GC-prog*:
‹*cdcl-twl-full-restart-l-GC-prog S* ≤ *SPEC* (*cdcl-twl-restart-l S*)›
if
inv: ‹*restart-abs-l-pre S last-GC last-Restart brk*› **and**
brk: ‹*ba* ∧ *b2a* ∧ ¬ *brk*›
for *ba b2a brk S last-GC last-Restart*
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
inv2: ‹*restart-prog-pre T last-GC last-Restart brk*› **and**
ent-init: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of T*)›
using *inv brk* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st*)
show *?thesis*
by (*rule cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l*[*unfolded Down-id-eq*, *OF ST list-invs*
struct-invs confl upd stgy-invs inv2])
qed

have *restart-abs-l-alt-def*:
‹*restart-abs-l S last-GC last-Restart n brk* = *do* {
ASSERT(*restart-abs-l-pre S last-GC last-Restart brk*);
b ← *GC-required-l S last-GC n*;

```
    b2 ← restart-required-l S last-Restart n;
    if b2 ∧ ¬brk then do {
      T ← SPEC(λT. cdcl-twl-restart-only-l S T);
      RETURN (T, last-GC, size (get-all-learned-clss-l T), n)
    }
    else
    if b ∧ ¬brk then do {
      inp ← inprocessing-required-l S;
    if ¬inp then do {
      - ← SPEC(λ-::bool. True);
      T ← SPEC(λT. cdcl-twl-restart-l S T);
      RETURN (T, size (get-all-learned-clss-l T), size (get-all-learned-clss-l T), n + 1)
      }
      else do {
        T ← SPEC (λT. cdcl-twl-restart-l-inp** S T ∧ count-decided (get-trail-l T) = 0);
        RETURN (T, size (get-all-learned-clss-l T), size (get-all-learned-clss-l T), n + 1)
     }
    }
    else
      RETURN (S, last-GC, last-Restart, n)
      }› for  S last-GC last-Restart n brk
    unfolding restart-abs-l-def
    by (auto cong: if-cong)
  have cdcl-twl-full-restart-inprocess-l:
   ‹cdcl-twl-full-restart-inprocess-l S ≤ SPEC (λT. cdcl-twl-restart-l-inp** S T ∧ count-decided (get-trail-l
T) = 0)›
    if
      inv: ‹restart-abs-l-pre S last-GC last-Restart brk› and
      brk: ‹ba ∧ b2a ∧ ¬ brk›
    for ba b2a brk S last-GC last-Restart
  proof −
    obtain T where
      ST: ‹(S, T) ∈ twl-st-l None› and
      struct-invs: ‹twl-struct-invs T› and
      list-invs: ‹twl-list-invs S› and
      upd: ‹clauses-to-update-l S = {#}› and
      stgy-invs: ‹twl-stgy-invs T› and
      confl: ‹get-conflict-l S = None› and
      inv2: ‹restart-prog-pre T last-GC last-Restart brk› and
      ent-init: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of T)›
      using inv brk unfolding restart-abs-l-pre-def restart-prog-pre-def
      apply − apply normalize-goal+
      by (auto simp: twl-st)
    show ?thesis
      by (rule cdcl-twl-full-restart-inprocess-l-cdcl-twl-restart-l[THEN order-trans,
        OF ST list-invs struct-invs confl upd stgy-invs inv2])
        auto
   qed
   have cdcl-twl-full-restart-inprocess-l:
       ‹cdcl-twl-full-restart-inprocess-l S ≤ SPEC (λT. cdcl-twl-restart-l-inp** S T ∧ count-decided
(get-trail-l T) = 0)›
     if
       inv: ‹restart-abs-l-pre S last-GC last-Restart brk› and
       brk: ‹b ∧ ¬ brk›
     for b ba b2 b2a inp inp' S last-GC last-Restart brk
   proof −
```

```
    obtain S' where
      SS': ‹(S, S') ∈ twl-st-l None› and
      struct-invs: ‹twl-struct-invs S'› and
      list-invs: ‹twl-list-invs S› and
      upd: ‹clauses-to-update-l S = {#}› and
      stgy-invs: ‹twl-stgy-invs S'› and
      confl: ‹get-conflict-l S = None› and
      inv2: ‹restart-prog-pre S' last-GC last-Restart brk› and
      ent-init: ‹cdcl_W-restart-mset.cdcl_W-learned-clauses-entailed-by-init (state_W-of S')›
      using inv brk unfolding restart-abs-l-pre-def restart-prog-pre-def
      apply - apply normalize-goal+
      by (auto simp: twl-st)

    show ?thesis
      by (rule cdcl-twl-full-restart-inprocess-l-cdcl-twl-restart-l[unfolded Down-id-eq, OF SS'
        list-invs
        struct-invs confl upd stgy-invs inv2])
  qed
  have [simp]: ‹cdcl-twl-restart-only-l S Ta ⟹ clauses-to-update-l Ta = {#}› for S Ta
    by (auto simp: cdcl-twl-restart-only-l.simps)
  have [simp]: ‹cdcl-twl-restart-l S Ta ⟹ clauses-to-update-l Ta = {#}› for S Ta
    by (auto simp: cdcl-twl-restart-l.simps)

  have ‹restart-prog-l S p m n brk ≤ ⇓ (?R ×_r nat-rel ×_r nat-rel ×_r nat-rel)
      (restart-abs-l S p m n brk)› for S n brk p m
    unfolding restart-prog-l-alt-def restart-abs-l-alt-def restart-required-l-def cdcl-twl-restart-l-prog-def
    apply (refine-vcg)
    subgoal by auto
    subgoal
      by (rule cdcl-twl-local-restart-l-spec-cdcl-twl-restart-l[THEN order-trans])
        (auto simp: conc-fun-RES)
    subgoal by (auto intro: cdcl-twl-restart-only-l-list-invs
      simp: restart-abs-l-pre-def)
    subgoal by auto
    subgoal by auto
    subgoal by (rule cdcl-twl-full-restart-l-prog) auto
    subgoal by (rule cdcl-twl-full-restart-l-GC-prog) auto
    subgoal by (auto simp: cdcl-twl-restart-l-list-invs
      simp: restart-abs-l-pre-def)
    subgoal for b ba b2 b2a inp inp'
      by (rule cdcl-twl-full-restart-inprocess-l)
    subgoal by (auto simp: restart-abs-l-pre-def
      dest: rtranclp-cdcl-twl-restart-l-inp-twl-list-invs
      rtranclp-cdcl-twl-restart-l-inp-clauses-to-update-l)
    subgoal by (auto simp: restart-abs-l-pre-def)
    done
  then show ?thesis
    apply -
    unfolding uncurry-def
    apply (intro frefI nres-relI)
    by force
qed


lemma restart-prog-l-restart-abs-l2:
  ‹(uncurry4 restart-prog-l, uncurry4 restart-abs-l)
```

$\in$ *Id* $\times_f$ *nat-rel* $\times_f$ *nat-rel*$\times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$
$\langle$*Id* $\times_r$ *nat-rel* $\times_r$ *nat-rel* $\times_r$ *nat-rel*$\rangle$*nres-rel*› (**is** ‹- $\in$ *?R* $\times_f$ *nat-rel* $\times_f$ *nat-rel*$\times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$ -›)
proof −
have *cdcl-twl-full-restart-l-prog*:
‹*cdcl-twl-full-restart-l-prog S* $\leq$ *SPEC* (λT. *cdcl-twl-restart-l S T*)›
if
inv: ‹*restart-abs-l-pre S last-GC last-Restart brk*› **and**
‹(*b*, *ba*) $\in$ *bool-rel*› **and**
‹*b* $\in$ {*b*. *b* $\longrightarrow$ *f n* < *size* (*S*)}› **and**
‹*ba* $\in$ {*b*. *b* $\longrightarrow$ *f n* < *size* (*S*)}› **and**
brk: ‹¬*brk*›
for *b ba S brk n last-GC last-Restart*
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) $\in$ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*›
using *inv brk* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st*)
show *?thesis*
using *cdcl-twl-full-restart-l-prog-spec*[*OF ST list-invs struct-invs confl upd*]
remove-one-annot-true-clause-cdcl-twl-restart-l-spec[*OF ST list-invs struct-invs confl upd*]
by (*rule conc-trans-additional*)
qed
have *cdcl-twl-full-restart-l-GC-prog*:
‹*cdcl-twl-full-restart-l-GC-prog S* $\leq$ *SPEC* (*cdcl-twl-restart-l S*)›
if
inv: ‹*restart-abs-l-pre S last-GC last-Restart brk*› **and**
brk: ‹*ba* $\wedge$ *b2a* $\wedge$ ¬ *brk*›
for *ba b2a brk S last-GC last-Restart*
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) $\in$ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
inv2: ‹*restart-prog-pre T last-GC last-Restart brk*›
using *inv brk* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st*)
show *?thesis*
by (*rule cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l*[*unfolded Down-id-eq*, *OF ST list-invs struct-invs confl upd stgy-invs inv2*])
qed
have *cdcl-twl-full-restart-inprocess-l*:
‹*cdcl-twl-full-restart-inprocess-l S* $\leq$ *SPEC* (λT. *cdcl-twl-restart-l-inp*** *S T* $\wedge$
count-decided (*get-trail-l T*) = *0*)›

```
  if
    inv: ‹restart-abs-l-pre S last-GC last-Restart brk› and
    brk: ‹ba ∧ b2a ∧ ¬ brk›
  for ba b2a brk S last-GC last-Restart
proof −
  obtain T where
    ST: ‹(S, T) ∈ twl-st-l None› and
    struct-invs: ‹twl-struct-invs T› and
    list-invs: ‹twl-list-invs S› and
    upd: ‹clauses-to-update-l S = {#}› and
    stgy-invs: ‹twl-stgy-invs T› and
    confl: ‹get-conflict-l S = None› and
    inv2: ‹restart-prog-pre T last-GC last-Restart brk›
    using inv brk unfolding restart-abs-l-pre-def restart-prog-pre-def
    apply − apply normalize-goal+
    by (auto simp: twl-st)
  show ?thesis
    by (rule cdcl-twl-full-restart-inprocess-l-cdcl-twl-restart-l[unfolded Down-id-eq, OF ST
      list-invs struct-invs confl upd stgy-invs inv2])
qed

have restart-abs-l-alt-def:
  ‹restart-abs-l S last-GC last-Restart n brk = do {
  ASSERT(restart-abs-l-pre S last-GC last-Restart brk);
  b ← GC-required-l S last-GC n;
  b2 ← restart-required-l S last-Restart n;
  if b2 ∧ ¬brk then do {
    T ← SPEC(λT. cdcl-twl-restart-only-l S T);
    RETURN (T, last-GC, size (get-all-learned-clss-l T), n)
    }
    else
  if b ∧ ¬brk then do {
    b ← inprocessing-required-l S;
  if ¬b then do {
    - ← SPEC(λb :: bool. True);
    T ← SPEC(λT. cdcl-twl-restart-l S T);
    RETURN (T, size (get-all-learned-clss-l T), size (get-all-learned-clss-l T), n + 1)
    } else do {
    T ← SPEC(λT. cdcl-twl-restart-l-inp** S T ∧ count-decided (get-trail-l T) = 0);
    RETURN (T, size (get-all-learned-clss-l T), size (get-all-learned-clss-l T), n + 1)
    }
    }
    else
    RETURN (S, last-GC, last-Restart, n)
    }› for S last-GC last-Restart n brk
  unfolding restart-abs-l-def
  by (auto cong: if-cong)

 have [simp]: ‹cdcl-twl-restart-only-l S Ta ⟹ clauses-to-update-l Ta = {#}› for S Ta
   by (auto simp: cdcl-twl-restart-only-l.simps)
 have [simp]: ‹cdcl-twl-restart-l S Ta ⟹ clauses-to-update-l Ta = {#}› for S Ta
   by (auto simp: cdcl-twl-restart-l.simps)
 have ‹restart-prog-l S p m n brk ≤ ⇓ (?R ×r nat-rel ×r nat-rel ×r nat-rel)
     (restart-abs-l S p m n brk)› for S n brk p m
  unfolding restart-prog-l-def restart-abs-l-alt-def restart-required-l-def cdcl-twl-restart-l-prog-def
  apply (refine-vcg)
```

subgoal by *auto*
subgoal
by (*rule cdcl-twl-local-restart-l-spec-cdcl-twl-restart-l*[*THEN order-trans*])
(*auto simp*: *conc-fun-RES*)
subgoal by (*auto intro*: *cdcl-twl-restart-only-l-list-invs*
simp: *restart-abs-l-pre-def*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule cdcl-twl-full-restart-l-prog*) *auto*
subgoal by (*rule cdcl-twl-full-restart-l-GC-prog*) *auto*
subgoal by (*auto simp*: *cdcl-twl-restart-l-list-invs*
simp: *restart-abs-l-pre-def*)
subgoal by (*rule cdcl-twl-full-restart-inprocess-l*) *auto*
subgoal by (*auto simp*: *cdcl-twl-restart-l-list-invs*
simp: *restart-abs-l-pre-def*)
subgoal by (*auto simp*: *cdcl-twl-restart-l-list-invs*
simp: *restart-abs-l-pre-def*)
done
then show *?thesis*
apply −
unfolding *uncurry-def*
apply (*intro frefI nres-relI*)
by *force*
qed

definition *cdcl-twl-stgy-restart-abs-early-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**
‹*cdcl-twl-stgy-restart-abs-early-l* S_0 =
do {
ebrk ← *RES UNIV*;
(-, *brk*, *T*, *last-GC*, *last-Restart*, *n*) ← $WHILE_T$*cdcl-twl-stgy-restart-abs-l-inv* S_0 *o snd*
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(-, *brk*, *S*, *last-GC*, *last-Restart*,*n*).
do {
T ← *unit-propagation-outer-loop-l S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-l T*;
(*T*, *last-GC*, *last-Restart*,*n*) ← *restart-abs-l T last-GC last-Restart n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk* ∨ *get-conflict-l T* ≠ *None*, *T*, *last-GC*, *last-Restart*,*n*)
})
(*ebrk*, *False*, S_0, *size* (*get-all-learned-clss-l* S_0), *size* (*get-all-learned-clss-l* S_0), *0*);
if ¬*brk then do* {
(*brk*, *T*, *last-GC*, *last-Restart*, -) ← $WHILE_T$*cdcl-twl-stgy-restart-abs-l-inv T*
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*, *last-GC*, *last-Restart*, *n*).
do {
T ← *unit-propagation-outer-loop-l S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-l T*;
(*T*, *last-GC*, *last-Restart*,*n*) ← *restart-abs-l T last-GC last-Restart n brk*;
RETURN (*brk* ∨ *get-conflict-l T* ≠ *None*, *T*, *last-GC*, *last-Restart*, *n*)
})
(*False*, *T*, *last-GC*, *last-Restart*, *n*);
RETURN T
} *else RETURN T*
}›

definition *cdcl-twl-stgy-restart-abs-bounded-l* :: *'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres* **where**

```
‹cdcl-twl-stgy-restart-abs-bounded-l S₀ =
do {
  ebrk ← RES UNIV;
  (ebrk, brk, T, n) ← WHILE_T^{cdcl-twl-stgy-restart-abs-l-inv S₀ o snd}
    (λ(ebrk, brk, -). ¬brk ∧ ¬ebrk)
    (λ(-, brk, S, last-GC, last-Restart, n).
    do {
      T ← unit-propagation-outer-loop-l S;
      (brk, T) ← cdcl-twl-o-prog-l T;
      (T, last-GC, last-Restart, n) ← restart-abs-l T last-GC last-Restart n brk;
ebrk ← RES UNIV;
      RETURN (ebrk, brk ∨ get-conflict-l T ≠ None, T, last-GC, last-Restart, n)
    })
    (ebrk, False, S₀, size (get-all-learned-clss-l S₀), size (get-all-learned-clss-l S₀), 0);
  RETURN (ebrk, T)
}›

definition cdcl-twl-stgy-restart-prog-l :: 'v twl-st-l ⇒ 'v twl-st-l nres where
‹cdcl-twl-stgy-restart-prog-l S₀ =
do {
  (brk, T, last-GC, last-Restart, n) ← WHILE_T^{cdcl-twl-stgy-restart-abs-l-inv S₀}
    (λ(brk, -). ¬brk)
    (λ(brk, S, last-GC, last-Restart, n).
    do {
T ← unit-propagation-outer-loop-l S;
(brk, T) ← cdcl-twl-o-prog-l T;
(T, last-GC, last-Restart, n) ← restart-prog-l T last-GC last-Restart n brk;
RETURN (brk ∨ get-conflict-l T ≠ None, T, last-GC, last-Restart, n)
    })
    (False, S₀, size (get-all-learned-clss-l S₀), size (get-all-learned-clss-l S₀), 0);
  RETURN T
}›


definition cdcl-twl-stgy-restart-prog-early-l :: 'v twl-st-l ⇒ 'v twl-st-l nres where
‹cdcl-twl-stgy-restart-prog-early-l S₀ =
do {
  ebrk ← RES UNIV;
  (ebrk, brk, T, last-GC, last-Restart, n) ← WHILE_T^{cdcl-twl-stgy-restart-abs-l-inv S₀ o snd}
    (λ(ebrk, brk, -). ¬brk ∧ ¬ebrk)
    (λ(ebrk, brk, S, last-GC, last-Restart, n).
    do {
      T ← unit-propagation-outer-loop-l S;
      (brk, T) ← cdcl-twl-o-prog-l T;
      (T, n) ← restart-prog-l T last-GC last-Restart n brk;
ebrk ← RES UNIV;
      RETURN (ebrk, brk ∨ get-conflict-l T ≠ None, T, n)
    })
    (ebrk, False, S₀, size (get-all-learned-clss-l S₀), size (get-all-learned-clss-l S₀), 0);
  if ¬brk then do {
    (brk, T, n) ← WHILE_T^{cdcl-twl-stgy-restart-abs-l-inv T}
(λ(brk, -). ¬brk)
(λ(brk, S, last-GC, last-Restart, n).
do {
  T ← unit-propagation-outer-loop-l S;
```

```
  (brk, T) ← cdcl-twl-o-prog-l T;
  (T, last-GC, last-Restart, n) ← restart-prog-l T last-GC last-Restart n brk;
  RETURN (brk ∨ get-conflict-l T ≠ None, T, last-GC, last-Restart, n)
 })
 (False, T, last-GC, last-Restart, n);
    RETURN T
   }
   else RETURN T
 }›


lemma cdcl-twl-stgy-restart-prog-early-l-cdcl-twl-stgy-restart-abs-early-l:
 ‹(cdcl-twl-stgy-restart-prog-early-l, cdcl-twl-stgy-restart-abs-early-l) ∈ {(S, S').
  (S, S') ∈ Id ∧  twl-list-invs S ∧  clauses-to-update-l S = {#}} →f ⟨Id⟩ nres-rel›
  (is ‹- ∈ ?R →f -›)
proof −
 have [refine0]: ‹((False, S, 0), (False, T , 0)) ∈ bool-rel ×r ?R ×r nat-rel›
   if ‹(S, T) ∈ ?R›
   for S T
   using that by auto
 have [refine0]: ‹unit-propagation-outer-loop-l x1c  ≤ ⇓ Id (unit-propagation-outer-loop-l x1a)›
   if ‹(x1c, x1a) ∈ Id›
   for x1c x1a
   using that by auto
 have [refine0]: ‹cdcl-twl-o-prog-l x1c  ≤ ⇓ Id (cdcl-twl-o-prog-l x1a)›
   if ‹(x1c, x1a) ∈ Id›
   for x1c x1a
   using that by auto
 show ?thesis
   unfolding cdcl-twl-stgy-restart-prog-early-l-def cdcl-twl-stgy-restart-prog-def uncurry-def
     cdcl-twl-stgy-restart-abs-early-l-def
   apply (intro frefI nres-relI)
   apply (refine-rcg WHILEIT-refine[where R = ‹bool-rel ×r Id ×r nat-rel ×r nat-rel ×r nat-rel›]
WHILEIT-refine[where R = ‹bool-rel ×r bool-rel ×r Id ×r nat-rel ×r nat-rel ×r nat-rel› ]
       unit-propagation-outer-loop-l-spec[THEN fref-to-Down]
       cdcl-twl-o-prog-l-spec[THEN fref-to-Down]
       restart-abs-l-restart-prog[THEN fref-to-Down-curry2]
       restart-prog-l-restart-abs-l2[THEN fref-to-Down-curry4])
   subgoal by auto
   subgoal by auto
   subgoal for x y xa x' x1 x2 x1a x2a
     by fastforce
   subgoal by auto
   subgoal
     by (simp add: twl-st)
   subgoal by (auto simp: twl-st)
   subgoal
     unfolding cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def
     by (auto simp: twl-st)
   subgoal by auto
   subgoal
     unfolding cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def
     by (auto simp: twl-st)
   subgoal by auto
   subgoal by auto
   subgoal by auto
```

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

lemma *cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-abs-l*:
‹(*cdcl-twl-stgy-restart-prog-l*, *cdcl-twl-stgy-restart-abs-l*) ∈ {(*S*, *S'*).
(*S*, *S'*) ∈ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} →$_f$ ⟨*Id*⟩ *nres-rel*›
(**is** ‹- ∈ *?R* →$_f$ -›)
proof −
have [*refine0*]: ‹((*False*, *S*, *0*), (*False*, *T* , *0*)) ∈ *bool-rel* ×$_r$ *?R* ×$_r$ *nat-rel*›
if ‹(*S*, *T*) ∈ *?R*›
for *S T*
using *that* **by** *auto*
have [*refine0*]: ‹*unit-propagation-outer-loop-l x1c* ≤ ⇓ *Id* (*unit-propagation-outer-loop-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
have [*refine0*]: ‹*cdcl-twl-o-prog-l x1c* ≤ ⇓ *Id* (*cdcl-twl-o-prog-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-l-def cdcl-twl-stgy-restart-prog-def uncurry-def*
cdcl-twl-stgy-restart-abs-l-def
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** *R* = ‹*bool-rel* ×$_r$ *Id* ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry4*]
restart-prog-l-restart-abs-l2[*THEN fref-to-Down-curry4*])
subgoal by *auto*
subgoal
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def*
by (*fastforce simp*: *prod-rel-fst-snd-iff*)
subgoal for *x y xa x' x1 x2 x1a x2a*
by *fastforce*
subgoal by *auto*
subgoal
by (*auto simp add*: *twl-st*)
subgoal by (*auto simp*: *twl-st*)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def*
by (*auto simp*: *twl-st*)
done
qed

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-prog*:
‹(*cdcl-twl-stgy-restart-prog-l*, *cdcl-twl-stgy-restart-prog*)
∈ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} →$_f$
⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S*}⟩*nres-rel*›
apply (*intro frefI nres-relI*)
apply (*rule order-trans*)
defer
apply (*rule cdcl-twl-stgy-restart-abs-l-cdcl-twl-stgy-restart-abs-l*[*THEN fref-to-Down*])

apply *fast*
apply *assumption*
apply (*rule cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-abs-l*[*THEN fref-to-Down*, *simplified*])
apply *simp*
done

definition *cdcl-twl-stgy-restart-prog-bounded-l* :: *'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres* **where**
‹*cdcl-twl-stgy-restart-prog-bounded-l* S_0 =
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, *T*, *last-GC*, *last-Restart*, *n*) ← $WHILE_T$*cdcl-twl-stgy-restart-abs-l-inv* S_0 *o snd*
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, *S*, *last-GC*, *last-Restart*, *n*).
do {
T ← *unit-propagation-outer-loop-l S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-l T*;
(*T*, *last-GC*, *last-Restart*, *n*) ← *restart-prog-l T last-GC last-Restart n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk* ∨ *get-conflict-l T* ≠ *None*, *T*, *last-GC*, *last-Restart*, *n*)
})
(*ebrk*, *False*, S_0, *size* (*get-all-learned-clss-l* S_0), *size* (*get-all-learned-clss-l* S_0), *0*);
RETURN (*ebrk*, *T*)
}›

lemma (**in** −) [*simp*]:
‹(*S*, *T*) ∈ *twl-st-l b* ⟹ *size* (*get-learned-clss T*) = *size* (*get-learned-clss-l S*)›
‹(*S*, *T*) ∈ *twl-st-l b* ⟹ (*get-init-learned-clss T*) = (*get-unit-learned-clss-l S*)›
by (*auto simp*: *twl-st-l-def get-learned-clss-l-def*)

lemma (**in** −) *get-all-learned-clss-alt-def*:
‹*get-all-learned-clss S* = *clauses* (*get-learned-clss S*) + *get-init-learned-clss S* +
subsumed-learned-clauses S + *get-learned-clauses0 S*›
by (*cases S*) *auto*

lemma *cdcl-twl-stgy-restart-abs-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*:
‹(*cdcl-twl-stgy-restart-abs-bounded-l*, *cdcl-twl-stgy-restart-prog-bounded*) ∈
{(*S* :: *'v twl-st-l*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}} →$_f$
⟨*bool-rel* ×$_r$ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S*}⟩ *nres-rel*›
unfolding *cdcl-twl-stgy-restart-abs-bounded-l-def cdcl-twl-stgy-restart-prog-bounded-def uncurry-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg*
WHILEIT-refine[**where** *R* = ‹*bool-rel* ×$_r$ *bool-rel* ×$_r$ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry4*])
subgoal by (*auto simp add*: *get-all-learned-clss-alt-def*)
subgoal for *x y ebrk ebrka xa x'*
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def comp-def case-prod-beta*
apply (*rule-tac x*=*y* **in** *exI*)
apply (*rule-tac x*=‹*fst* (*snd* (*snd x'*))› **in** *exI*)
by (*auto simp*: *prod-rel-fst-snd-iff*)
subgoal by (*auto simp*: *prod-rel-fst-snd-iff*)

subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
cdcl-twl-stgy-restart-abs-l-inv-def
apply (*simp only*: *prod.case*)
apply (*normalize-goal*)+
by (*simp add*: *twl-st-l twl-st*)
subgoal by (*auto simp*: *twl-st-l twl-st*)
subgoal by *auto*
subgoal by (*auto simp*: *twl-st-l twl-st*)
subgoal by (*auto simp*: *prod-rel-fst-snd-iff*)
done

lemma *cdcl-twl-stgy-restart-abs-early-l-cdcl-twl-stgy-restart-abs-early*:
‹(*cdcl-twl-stgy-restart-abs-early-l*, *cdcl-twl-stgy-restart-prog-early*)
∈ {(*S* :: 'v *twl-st-l*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}} →$_f$ ⟨*twl-st-l None*⟩*nres-rel*›
proof −
show *?thesis*
unfolding *cdcl-twl-stgy-restart-abs-early-l-def cdcl-twl-stgy-restart-prog-early-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg*
WHILEIT-refine[**where** *R* = ‹*bool-rel* ×$_r$ *bool-rel* ×$_r$ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#}} ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry4*]
WHILEIT-refine[**where** *R* =
‹*bool-rel* ×$_r$ {(*S* :: 'v *twl-st-l*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}} ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›])
subgoal by (*auto simp add*: *get-all-learned-clss-alt-def*)
subgoal for *x y ebrk ebrka xa x'*
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def comp-def case-prod-beta*
apply (*rule-tac x*=*y* **in** *exI*)
apply (*rule-tac x*=‹*fst* (*snd* (*snd x'*))› **in** *exI*)
by (*auto simp*: *prod-rel-fst-snd-iff*)
subgoal by (*auto simp*: *prod-rel-fst-snd-iff*)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
cdcl-twl-stgy-restart-abs-l-inv-def
apply (*simp only*: *prod.case*)
apply (*normalize-goal*)+
by (*simp add*: *twl-st-l twl-st*)
subgoal by (*auto simp*: *twl-st-l twl-st*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *x y ebrk ebrka xa x' x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f*
x1g x2g x1h x2h x1i x2i xb x'a
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def comp-def case-prod-beta*
apply (*rule-tac x*= ‹*x1b*› **in** *exI*)
apply (*rule-tac x*=‹*fst* (*snd x'a*)› **in** *exI*)
apply (*rule-tac x*= ‹*x2d*› **in** *exI*)
by (*auto simp*: *prod-rel-fst-snd-iff*)
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

lemma *cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*:
‹(*cdcl-twl-stgy-restart-prog-bounded-l*, *cdcl-twl-stgy-restart-abs-bounded-l*) ∈ {(*S*, *S'*).
(*S*:: 'v *twl-st-l*, *S'*) ∈ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} →$_f$ ⟨*Id*⟩ *nres-rel*›
(**is** ‹- ∈ *?R* →$_f$ -›)
proof −
have [*refine0*]: ‹((*ebrk*, *False*, *S*, *size* (*get-all-learned-clss-l S*),
size (*get-all-learned-clss-l S*), *0*::*nat*),
ebrka, *False*, *T*, *size* (*get-all-learned-clss-l T*),
size (*get-all-learned-clss-l T*), *0*::*nat*) ∈ *bool-rel* ×$_r$ *bool-rel* ×$_r$ *?R* ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›
if ‹(*S*, *T*) ∈ *?R*› ‹(*ebrk*, *ebrka*) ∈ *bool-rel*›
for *S T ebrk ebrka*
using *that* **by** *auto*
have [*refine0*]: ‹*unit-propagation-outer-loop-l x1c* ≤ ⇓ *Id* (*unit-propagation-outer-loop-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
have [*refine0*]: ‹*cdcl-twl-o-prog-l x1c* ≤ ⇓ *Id* (*cdcl-twl-o-prog-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
show *?thesis*
supply [*simp*] = *prod-rel-fst-snd-iff*
unfolding *cdcl-twl-stgy-restart-prog-bounded-l-def cdcl-twl-stgy-restart-prog-def uncurry-def*
cdcl-twl-stgy-restart-abs-bounded-l-def
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** *R* = ‹*bool-rel* ×$_r$ *bool-rel* ×$_r$ *Id* ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›]
WHILEIT-refine[**where** *R* = ‹*bool-rel* ×$_r$ *Id* ×$_r$ *nat-rel* ×$_r$ *nat-rel* ×$_r$ *nat-rel*›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-prog-l-restart-abs-l2[*THEN fref-to-Down-curry4*])
subgoal
by *auto*
subgoal
by *fastforce*
subgoal by *auto*
subgoal
by (*simp add*: *twl-st*)
subgoal
by (*auto simp*: *twl-st*)
subgoal by *auto*
subgoal by *auto*
done
qed

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-prog-bounded*:
‹(*cdcl-twl-stgy-restart-prog-bounded-l*, *cdcl-twl-stgy-restart-prog-bounded*)

$\in \{(S, S').\ (S, S') \in \textit{twl-st-l None} \wedge \textit{twl-list-invs } S \wedge \textit{clauses-to-update-l } S = \{\#\}\} \rightarrow_f$
$\langle \textit{bool-rel} \times_r \{(S, S').\ (S, S') \in \textit{twl-st-l None} \wedge \textit{twl-list-invs } S\} \rangle \textit{nres-rel}\rangle$
apply (*intro frefI nres-relI*)
apply (*rule order-trans*)
defer
apply (*rule cdcl-twl-stgy-restart-abs-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*[*THEN fref-to-Down*])
apply *fast*
apply *assumption*
apply (*rule cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*[*THEN fref-to-Down*, *simplified*])
apply *simp*
done

end

end
theory *Watched-Literals-Watch-List-Restart*
imports *Watched-Literals-Watch-List*
Watched-Literals-List-Simp
begin

lemma *cdcl-twl-restart-get-all-init-clss*:
assumes ‹*cdcl-twl-restart S T*›
shows ‹*get-all-init-clss T = get-all-init-clss S*›
using *assms* **by** (*induction rule*: *cdcl-twl-restart.induct*) *auto*

lemma *rtranclp-cdcl-twl-restart-get-all-init-clss*:
assumes ‹*cdcl-twl-restart*** *S T*›
shows ‹*get-all-init-clss T = get-all-init-clss S*›
using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto simp*: *cdcl-twl-restart-get-all-init-clss*)

As we have a specialised version of *correct-watching*, we defined a special version for the inclusion of the domain:

definition *all-init-lits* :: ‹(*nat*, *'v literal list* × *bool*) *fmap* ⇒ *'v literal multiset multiset* ⇒
'v literal multiset› **where**
‹*all-init-lits S NUE = all-lits-of-mm* ((λ*C*. *mset C*) ‘# *init-clss-lf S* + *NUE*)›

lemma *all-init-lits-alt-def*:
‹*all-init-lits S* (*NUE* + *NUS* + *N0S*) = *all-lits-of-mm* ((λ*C*. *mset C*) ‘# *init-clss-lf S* + *NUE* + *NUS* + *N0S*)›
‹*all-init-lits b* (*d* + *f* + *g*) = *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *init-clss-l b*#} + *d* + *f* + *g*)›
by (*auto simp*: *all-init-lits-def ac-simps*)

definition *all-init-atms* :: ‹- ⇒ - ⇒ *'v multiset*› **where**
‹*all-init-atms N NUE = atm-of* ‘# *all-init-lits N NUE*›

declare *all-init-atms-def*[*symmetric*, *simp*]

lemma *all-init-atms-alt-def*:
‹*set-mset* (*all-init-atms N NE*) = *atms-of-mm* (*mset* ‘# *init-clss-lf N*) ∪ *atms-of-mm NE*›
unfolding *all-init-atms-def all-init-lits-def*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff*

all-lits-of-mm-def atms-of-ms-def image-UN
atms-of-def
dest!: *multi-member-split*[*of* ‹(-, -)› ‹*ran-m N*›]
dest: *multi-member-split atm-of-lit-in-atms-of*
simp del: *set-image-mset*)

lemma *in-set-all-init-atms-iff*:
‹*y* ∈# *all-init-atms bu bw* ⟷
y ∈ *atms-of-mm* (*mset* `# *init-clss-lf bu*) ∨ *y* ∈ *atms-of-mm bw*›
by (*auto simp*: *all-lits-def atm-of-all-lits-of-mm all-init-atms-alt-def atms-of-def*
in-all-lits-of-mm-ain-atms-of-iff all-lits-of-mm-def atms-of-ms-def image-UN
dest!: *multi-member-split*[*of* ‹(-, -)› ‹*ran-m N*›]
dest: *multi-member-split atm-of-lit-in-atms-of*
simp del: *set-image-mset*)

lemma *all-init-atms-fmdrop-add-mset-unit*:
‹*C* ∈# *dom-m baa* ⟹ *irred baa C* ⟹
all-init-atms (*fmdrop C baa*) (*add-mset* (*mset* (*baa* ∝ *C*)) *da*) =
all-init-atms baa da›
‹*C* ∈# *dom-m baa* ⟹ ¬*irred baa C* ⟹
all-init-atms (*fmdrop C baa*) *da* =
all-init-atms baa da›
by (*auto simp del*: *all-init-atms-def*[*symmetric*]
simp: *all-init-atms-def all-init-lits-def*
init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if)

lemma *all-init-lits-of-wl-simps*[*simp*]:
‹*C* ∈# *dom-m N* ⟹ ¬*irred N C* ⟹
all-init-lits-of-wl (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*NO-MATCH* {#} *US* ⟹
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›
‹*NO-MATCH* [] *M* ⟹
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl ([], *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*C* ∈# *dom-m N* ⟹ *irred N C* ⟹
all-init-lits-of-wl (*M*, *fmdrop C N*, *D*, *add-mset* (*mset* (*N* ∝ *C*)) *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*,
U0, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*all-init-lits-of-wl* (*M*, *N*, *D*, *NE*, *add-mset E UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*NO-MATCH* {#} *UEk* ⟹
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, {#}, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*NO-MATCH* {#} *U0* ⟹
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, {#}, *Q*, *W*)›
‹*NO-MATCH* {#} *UE* ⟹
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-init-lits-of-wl (*M*, *N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
by (*auto simp*: *all-init-lits-of-wl-def all-lits-of-mm-add-mset*
image-mset-remove1-mset-if)

lemma *all-learned-lits-of-wl-simps*[*simp*]:
‹*C* ∈# *dom-m N* ⟹ *irred N C* ⟹

all-learned-lits-of-wl (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-learned-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›

‹*NO-MATCH* [] *M* ⟹
all-learned-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-learned-lits-of-wl ([], *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*all-learned-lits-of-wl* (*M*, *N*, *D*, *add-mset E NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-learned-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*all-learned-lits-of-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *add-mset E NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-learned-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
‹*C* ∈# *dom-m N* ⟹ ¬*irred N C* ⟹
all-learned-lits-of-wl (*M*, *fmdrop C N*, *D*, *NE*, *add-mset* (*mset* (*N* ∝ *C*)) *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) =
all-learned-lits-of-wl (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
by (*auto simp*: *all-learned-lits-of-wl-def all-lits-of-mm-add-mset*
image-mset-remove1-mset-if)

To ease the proof, we introduce the following "alternative" definitions, that only considers variables that are present in the initial clauses (which are never deleted from the set of clauses, but only moved to another component).

fun *correct-watching′* :: ‹′*v twl-st-wl* ⇒ *bool*› **where**
‹*correct-watching′* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-init-lits-of-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
distinct-watched (*W L*) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*).
i ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*).
b ⟶ *i* ∈# *dom-m N*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* '# *mset* (*W L*)) =
clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›

fun *correct-watching′-nobin* :: ‹′*v twl-st-wl* ⇒ *bool*› **where**
‹*correct-watching′-nobin* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-init-lits-of-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
distinct-watched (*W L*) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*).
i ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* '# *mset* (*W L*)) =
clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›

lemma *correct-watching′-correct-watching′*: ‹*correct-watching′ S* ⟹ *correct-watching′ S*›
by (*cases S*) *auto*

declare *correct-watching′-nobin.simps*[*simp del*] *correct-watching′.simps*[*simp del*]

Now comes a weaker version of the invariants on watch lists: instead of knowing that the watch lists are correct, we only know that the clauses appear somewhere in the watch lists. From a conceptual point of view, this is sufficient to specify all operations, but this is not sufficient to derive bounds on the length. Hence, we also add the invariants that each watch list does not contain duplicates.

definition *no-lost-clause-in-WL* :: ‹′*v twl-st-wl* ⇒ *bool*› **where**
‹*no-lost-clause-in-WL S* ≡
set-mset (*dom-m* (*get-clauses-wl S*))
⊆ *clauses-pointed-to* (*set-mset* (*all-init-lits-of-wl S*)) (*get-watched-wl S*) ∧

(∀ L∈# all-init-lits-of-wl S. distinct-watched (watched-by S L))›

definition no-lost-clause-in-WL0 :: ‹'v twl-st-wl ⇒ bool› **where**
‹no-lost-clause-in-WL0 S ≡
set-mset (dom-m (get-clauses-wl S))
⊆ clauses-pointed-to (set-mset (all-init-lits-of-wl S)) (get-watched-wl S)›

definition blits-in-$\mathcal{L}_{in}$' :: ‹'v twl-st-wl ⇒ bool› **where**
‹blits-in-$\mathcal{L}_{in}$' S ⟷
(∀ L ∈# all-init-lits-of-wl S.
∀(i, K, b) ∈ set (watched-by S L). K ∈# all-init-lits-of-wl S)›

definition literals-are-$\mathcal{L}_{in}$' :: ‹'v twl-st-wl ⇒ bool› **where**
‹literals-are-$\mathcal{L}_{in}$' S ≡
set-mset (all-learned-lits-of-wl S) ⊆ set-mset (all-init-lits-of-wl S) ∧
blits-in-$\mathcal{L}_{in}$' S›

definition all-init-atms-st :: ‹'v twl-st-wl ⇒ 'v multiset› **where**
‹all-init-atms-st S ≡ all-init-atms (get-clauses-wl S)
(get-unit-init-clss-wl S + get-subsumed-init-clauses-wl S + get-init-clauses0-wl S)›

lemma all-init-atms-st-alt-def: ‹all-init-atms-st S = atm-of `# all-init-lits-of-wl S›
by (auto simp: all-atms-def all-lits-st-def all-init-atms-st-def all-init-lits-of-wl-def
atm-of-all-lits-of-mm all-init-atms-def all-init-lits-def ac-simps
simp del: all-init-atms-def[symmetric])

lemma $\mathcal{L}_{all}$-all-init-atms:
‹set-mset ($\mathcal{L}_{all}$ (all-init-atms N NU)) = set-mset (all-init-lits N NU)›
‹set-mset ($\mathcal{L}_{all}$ (all-init-atms-st S)) = set-mset (all-init-lits-of-wl S)›
by (simp-all add: $\mathcal{L}_{all}$-atm-of-all-lits-of-mm all-init-atms-def all-init-lits-def
all-init-lits-of-wl-def ac-simps all-init-atms-st-def)

lemma literals-are-$\mathcal{L}_{in}$-cong:
‹set-mset $\mathcal{A}$ = set-mset $\mathcal{B}$ ⟹ literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ S = literals-are-$\mathcal{L}_{in}$ $\mathcal{B}$ S›
using $\mathcal{L}_{all}$-cong[of $\mathcal{A}$ $\mathcal{B}$]
unfolding literals-are-$\mathcal{L}_{in}$-def blits-in-$\mathcal{L}_{in}$-def is-$\mathcal{L}_{all}$-def
by auto

lemma all-learned-lits-of-wl-all-lits-st:
‹set-mset (all-learned-lits-of-wl S) ⊆ set-mset (all-lits-st S)›
unfolding all-learned-lits-of-wl-def all-lits-st-def all-lits-def
apply (subst (2) all-clss-l-ran-m[symmetric])
unfolding image-mset-union
by (cases S) (auto simp: all-lits-of-mm-union)

lemma all-lits-st-init-learned:
‹set-mset (all-lits-st S) = set-mset (all-init-lits-of-wl S) ∪ set-mset (all-learned-lits-of-wl S)›
unfolding all-learned-lits-of-wl-def all-lits-st-def all-lits-def all-init-lits-of-wl-def
apply (subst (1) all-clss-l-ran-m[symmetric])
unfolding image-mset-union
by (cases S) (auto simp: all-lits-of-mm-union)

lemma $\mathcal{L}_{all}$-all-atms:
‹set-mset ($\mathcal{L}_{all}$ (all-atms-st S)) = set-mset (all-lits-st S)›
by (metis $\mathcal{L}_{all}$-atm-of-all-lits-of-mm all-atms-st-alt-def-sym all-lits-def all-lits-st-def)

lemma *literals-are-$\mathcal{L}_{in}$'-literals-are-$\mathcal{L}_{in}$-iff*:
assumes
Sx: ‹(*S*, *x*) ∈ *state-wl-l None*› **and**
x-xa: ‹(*x*, *xa*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs xa*›
shows
‹*literals-are-$\mathcal{L}_{in}$' S* ⟷ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*› (**is** *?A*)
‹*literals-are-$\mathcal{L}_{in}$' S* ⟷ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*› (**is** *?B*)
‹*set-mset* (*all-init-atms-st S*) = *set-mset* (*all-atms-st S*)› (**is** *?C*) **and**
‹*set-mset* (*all-init-lits-of-wl S*) = *set-mset* (*all-lits-st S*)› (**is** *?D*)
proof −
have ‹*cdcl$_W$-restart-mset.no-strange-atm* (*state$_W$-of xa*)›
using *struct-invs* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
pcdcl-all-struct-invs-def state$_W$-of-def
by *fast+*
then have ‹⋀*L*. *L* ∈ *atm-of* ‘ *lits-of-l* (*get-trail-wl S*) ⟹ *L* ∈ *atms-of-ms*
((λ*x*. *mset* (*fst x*)) ‘ {*a*. *a* ∈# *ran-m* (*get-clauses-wl S*) ∧ *snd a*}) ∪
atms-of-mm (*get-unit-init-clss-wl S*) ∪
atms-of-mm (*get-subsumed-init-clauses-wl S*) ∪
atms-of-mm (*get-init-clauses0-wl S*)› **and**
alien-learned: ‹*atms-of-mm* (*learned-clss* (*state$_W$-of xa*))
⊆ *atms-of-mm* (*init-clss* (*state$_W$-of xa*))›
using *Sx x-xa* **unfolding** *cdcl$_W$-restart-mset.no-strange-atm-def*
by (*auto 5 2 simp add*: *twl-st twl-st-l twl-st-wl*)

show *1*: ‹*set-mset* (*all-init-lits-of-wl S*) = *set-mset* (*all-lits-st S*)›
unfolding *all-lits-st-def all-lits-def all-init-lits-of-wl-def*
apply (*subst* (*2*) *all-clss-l-ran-m*[*symmetric*])
using *alien-learned Sx x-xa*
unfolding *image-mset-union all-lits-of-mm-union*
by (*auto simp* : *in-all-lits-of-mm-ain-atms-of-iff get-unit-clauses-wl-alt-def*
twl-st twl-st-l twl-st-wl get-learned-clss-wl-def)
have ‹*set-mset* (*all-init-lits-of-wl S*) = *set-mset* (*$\mathcal{L}_{all}$* (*all-init-atms-st S*))›
unfolding *$\mathcal{L}_{all}$-all-init-atms*(*2*) **..**
show *A*: ‹*literals-are-$\mathcal{L}_{in}$' S* ⟷ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*› **for** *$\mathcal{A}$*
proof −
have *sub*: ‹*set-mset* (*all-lits-st S*) ⊆ *set-mset* (*all-init-lits-of-wl S*) ⟷
is-$\mathcal{L}_{all}$ (*all-init-atms-st S*) (*all-lits-st S*)›
using *all-init-lits-of-wl-all-lits-st*[*of S*]
unfolding *is-$\mathcal{L}_{all}$-def $\mathcal{L}_{all}$-all-init-atms*(*2*) **by** *auto*
have ‹*set-mset* (*all-learned-lits-of-wl S*) ⊆ *set-mset* (*all-lits-st S*) ⟷
is-$\mathcal{L}_{all}$ (*all-atms-st S*) (*all-lits-st S*)›
using *all-init-lits-of-wl-all-lits-st*[*of S*]
unfolding *all-lits-st-init-learned is-$\mathcal{L}_{all}$-def $\mathcal{L}_{all}$-all-atms*
by *auto*
then show *?thesis*
unfolding *literals-are-$\mathcal{L}_{in}$'-def*
literals-are-$\mathcal{L}_{in}$-def blits-in-$\mathcal{L}_{in}$-def blits-in-$\mathcal{L}_{in}$'-def sub
all-init-lits-def[*symmetric*] *all-lits-alt-def2*[*symmetric*]
all-lits-alt-def[*symmetric*] *all-init-lits-alt-def*[*symmetric*]
is-$\mathcal{L}_{all}$-def[*symmetric*] *all-init-atms-def*[*symmetric*] *1*
by *simp*
qed

show *C*: *?C*

```
    using 1 unfolding all-atms-st-alt-def all-init-atms-st-alt-def
    apply (simp add: 1 del: all-init-atms-def[symmetric])
    by (metis all-atms-st-alt-def set-image-mset)

  show ?B
    apply (subst A)
    ..
qed

lemma correct-watching'-nobin-clauses-pointed-to0:
  assumes
    xa-xb: ‹(xa, xb) ∈ state-wl-l None› and
    corr: ‹correct-watching'-nobin xa› and
    L: ‹literals-are-ℒ_in' xa› and
    xb-x: ‹(xb, x) ∈ twl-st-l None› and
    struct-invs: ‹twl-struct-invs x›

  shows ‹set-mset (dom-m (get-clauses-wl xa))
    ⊆ clauses-pointed-to
    (Neg ‘ set-mset (all-init-atms-st xa) ∪
    Pos ‘ set-mset (all-init-atms-st xa))
    (get-watched-wl xa)›
    (is ?G1 is ‹- ⊆ ?A›) and
    ‹no-lost-clause-in-WL xa› (is ?G2)
proof −
  let ?A = ‹all-init-atms (get-clauses-wl xa) (get-unit-init-clss-wl xa)›
  show ?G1
  proof
    fix C
    assume C: ‹C ∈# dom-m (get-clauses-wl xa)›
    obtain M N D NE UE NEk UEk NS US N0 U0 Q W where
      xa: ‹xa = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)›
      by (cases xa)
    have ‹twl-st-inv x›
      using xb-x C struct-invs
      by (auto simp: twl-struct-invs-def
        cdcl_W-restart-mset.cdcl_W-all-struct-inv-def)
    then have le0: ‹get-clauses-wl xa ∝ C ≠ []›
      using xb-x C xa-xb
      by (cases x; cases ‹irred N C›)
        (auto simp: twl-struct-invs-def twl-st-inv.simps
          twl-st-l-def state-wl-l-def xa ran-m-def conj-disj-distribR
          Collect-disj-eq Collect-conv-if
        dest!: multi-member-split)
    then have le: ‹N ∝ C ! 0 ∈ set (watched-l (N ∝ C))›
      by (cases ‹N ∝ C›) (auto simp: xa)
    have eq: ‹set-mset (ℒ_all (all-init-atms N NE)) =
        set-mset (all-lits-of-mm (mset ‘# init-clss-lf N + NE))›
      by (auto simp del: all-init-atms-def[symmetric]
        simp: all-init-atms-def xa ℒ_all-atm-of-all-lits-of-mm[symmetric]
          all-init-lits-def)

    have H: ‹get-clauses-wl xa ∝ C ! 0 ∈# all-init-lits-of-wl xa›
      using L C le0 apply −
      unfolding all-init-atms-def[symmetric] all-init-lits-def[symmetric]
      apply (subst literals-are-ℒ_in'-literals-are-ℒ_in-iff(4)[OF xa-xb xb-x struct-invs])
```

```
    apply (cases ‹N ∝ C›; auto simp: literals-are-\mathcal{L}_in-def all-lits-def ran-m-def eq
        all-lits-of-mm-add-mset is-\mathcal{L}_all-def xa all-lits-of-m-add-mset
        \mathcal{L}_all-all-atms-all-lits all-lits-st-def
      dest!: multi-member-split)
    done
  moreover {
    have ‹{#i ∈# fst `# mset (W (N ∝ C ! 0)). i ∈# dom-m N#} =
          add-mset C {#Ca ∈# remove1-mset C (dom-m N). N ∝ C ! 0 ∈ set (watched-l (N ∝
Ca))#}›
      using corr H C le unfolding xa
      by (auto simp: clauses-pointed-to-def correct-watching'-nobin.simps xa
        simp flip: all-init-atms-def all-init-lits-def all-init-atms-alt-def
          all-init-lits-alt-def
        simp: clause-to-update-def
        simp del: all-init-atms-def[symmetric]
        dest!: multi-member-split)
    from arg-cong[OF this, of set-mset] have ‹C ∈ fst ` set (W (N ∝ C ! 0))›
      using corr H C le unfolding xa
      by (auto simp: clauses-pointed-to-def correct-watching'.simps xa
        simp: all-init-atms-def all-init-lits-def clause-to-update-def
        simp del: all-init-atms-def[symmetric]
        dest!: multi-member-split) }
  ultimately show ‹C ∈ ?A›
    by (cases ‹N ∝ C ! 0›)
      (auto simp: clauses-pointed-to-def correct-watching'.simps xa
        simp flip: all-init-lits-def all-init-atms-alt-def
          all-init-lits-alt-def
        simp: clause-to-update-def all-init-atms-st-def all-init-lits-of-wl-def all-init-atms-def
        simp del: all-init-atms-def[symmetric]
      dest!: multi-member-split)
qed
moreover have ‹set-mset (all-init-lits-of-wl xa) =
  Neg ` set-mset (all-init-atms-st xa) ∪ Pos ` set-mset (all-init-atms-st xa)›
  unfolding all-init-lits-of-wl-def
    all-lits-of-mm-def all-init-atms-st-def all-init-atms-def
  by (auto simp: all-init-atms-def all-init-lits-def all-lits-of-mm-def image-image
    image-Un
    simp del: all-init-atms-def[symmetric])
moreover have ‹distinct-watched (watched-by xa (Pos L))›
  ‹distinct-watched (watched-by xa (Neg L))›
  if ‹L ∈# all-init-atms-st xa› for L
  using that corr
  by (cases xa;
      auto simp: correct-watching'-nobin.simps all-init-lits-of-wl-def all-init-atms-def
      all-lits-of-mm-union all-init-lits-def all-init-atms-st-def literal.atm-of-def
      in-all-lits-of-mm-uminus-iff[symmetric, of ‹Pos -›]
      simp del: all-init-atms-def[symmetric]
      split: literal.splits; fail)+
ultimately show ?G2
  unfolding no-lost-clause-in-WL-def
  by (auto simp del: all-init-atms-def[symmetric])
qed

lemma correct-watching'-clauses-pointed-to2:
  assumes
    xa-xb: ‹(xa, xb) ∈ state-wl-l None› and
```

corr: ‹*correct-watching'-nobin xa*› **and**
pre: ‹*mark-to-delete-clauses-l-GC-pre xb*› **and**
L: ‹*literals-are-$\mathcal{L}_{in}$' xa*›
shows ‹*set-mset* (*dom-m* (*get-clauses-wl xa*))
⊆ *clauses-pointed-to*
(*Neg* ‘ *set-mset* (*all-init-atms-st xa*) ∪
Pos ‘ *set-mset* (*all-init-atms-st xa*))
(*get-watched-wl xa*)›
(**is** *?G1* **is** ‹- ⊆ *?A*›) **and**
‹*no-lost-clause-in-WL xa*› (**is** *?G2*)
using *correct-watching'-nobin-clauses-pointed-to0*[*OF xa-xb corr L*] *pre*
unfolding *mark-to-delete-clauses-l-GC-pre-def*
by *fast+*

definition (**in** −) *restart-abs-wl-pre* :: ‹'*v twl-st-wl* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-abs-wl-pre S last-GC last-Restart brk* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *restart-abs-l-pre S' last-GC last-Restart brk*
∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*)›

definition (**in** −) *cdcl-twl-local-restart-wl-spec* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*cdcl-twl-local-restart-wl-spec* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(∃ *last-GC last-Restart*. *restart-abs-wl-pre* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) *last-GC last-Restart False*);
(*M*, *Q*) ← *SPEC*(λ(*M'*, *Q'*). (∃ *K M2*. (*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q' = {#}) ∨ (*M'* = *M* ∧ *Q'* = *Q*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)
})›

lemma *cdcl-twl-local-restart-wl-spec-cdcl-twl-local-restart-l-spec*:
‹(*cdcl-twl-local-restart-wl-spec*, *cdcl-twl-local-restart-l-spec*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*} →$_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*}⟩*nres-rel*›
proof −
have [*simp*]:
‹*all-lits N* (*NE* + *UE* + (*NS* + *US*) + (*N0* + *U0*)) = *all-lits N* (*NE* + *UE* + *NS* + *US* + *N0* + *U0*)›
‹*all-lits N* ((*NE* + *UE*) + (*NS* + *US*) + (*N0* + *U0*)) = *all-lits N* (*NE* + *UE* + *NS* + *US* + *N0* + *U0*)›
for *NE UE NS US N N0 U0*
by (*auto simp*: *ac-simps*)
have [*refine0*]:
‹⋀*x y x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h x1i x2i x1j x2j x1k x2k x1l x2l x1m x2m x1n*
x2n x1o x2o x1p x2p x1q x2q x1r x2r x1s x2s x1t x2t x1u x2u x1v x2v x1w x2w xa x1x x2x.
(*x*, *y*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*} ⟹
x2j = (*x1k*, *x2k*) ⟹
x2i = (*x1j*, *x2j*) ⟹
x2h = (*x1i*, *x2i*) ⟹
x2g = (*x1h*, *x2h*) ⟹
x2f = (*x1g*, *x2g*) ⟹
x2e = (*x1f*, *x2f*) ⟹
x2d = (*x1e*, *x2e*) ⟹
x2c = (*x1d*, *x2d*) ⟹
x2b = (*x1c*, *x2c*) ⟹

$x2a = (x1b,\ x2b) \Longrightarrow$
$x2 = (x1a,\ x2a) \Longrightarrow$
$y = (x1,\ x2) \Longrightarrow$
$x2v = (x1w,\ x2w) \Longrightarrow$
$x2u = (x1v,\ x2v) \Longrightarrow$
$x2t = (x1u,\ x2u) \Longrightarrow$
$x2s = (x1t,\ x2t) \Longrightarrow$
$x2r = (x1s,\ x2s) \Longrightarrow$
$x2q = (x1r,\ x2r) \Longrightarrow$
$x2p = (x1q,\ x2q) \Longrightarrow$
$x2o = (x1p,\ x2p) \Longrightarrow$
$x2n = (x1o,\ x2o) \Longrightarrow$
$x2m = (x1n,\ x2n) \Longrightarrow$
$x2l = (x1m,\ x2m) \Longrightarrow$
$x = (x1l,\ x2l) \Longrightarrow$
case xa of
$(M',\ Q') \Rightarrow (\exists K\ M2.\ (Decided\ K\ \#\ M',\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ x1l) \wedge Q' = \{\#\}) \vee M' = x1l \wedge Q' = x1w \Longrightarrow$
$xa = (x1x,\ x2x) \Longrightarrow$
$(\exists K\ M2.\ (Decided\ K\ \#\ x1x,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ x1) \wedge x2x = \{\#\}) \vee x1x = x1 \wedge x2x = x2k$›
by (*auto 5 3 simp*: *state-wl-l-def*)
show *?thesis*
unfolding *cdcl-twl-local-restart-wl-spec-def cdcl-twl-local-restart-l-spec-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg*)
subgoal unfolding *restart-abs-wl-pre-def* **by** *fast*
apply *assumption+*
subgoal
by (*fastforce simp*: *state-wl-l-def correct-watching.simps clause-to-update-def*
blits-in-$\mathcal{L}_{in}$-def
simp flip: *all-lits-alt-def2*)
done
qed

definition *cdcl-twl-restart-wl-prog* **where**
‹*cdcl-twl-restart-wl-prog S = do* {
cdcl-twl-local-restart-wl-spec S
}›

lemma *cdcl-twl-restart-wl-prog-cdcl-twl-restart-l-prog*:
‹(*cdcl-twl-restart-wl-prog*, *cdcl-twl-restart-l-prog*)
$\in \{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching S* $\wedge$ *blits-in-$\mathcal{L}_{in}$ S*$\} \rightarrow_f$
$\langle\{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching S* $\wedge$ *blits-in-$\mathcal{L}_{in}$ S*$\}\rangle$*nres-rel*›
unfolding *cdcl-twl-restart-wl-prog-def cdcl-twl-restart-l-prog-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg cdcl-twl-local-restart-wl-spec-cdcl-twl-local-restart-l-spec*[*THEN fref-to-Down*])
done

definition *cdcl-twl-full-restart-wl-GC-prog-post* :: ‹*'v twl-st-wl* $\Rightarrow$ *'v twl-st-wl* $\Rightarrow$ *bool*› **where**
‹*cdcl-twl-full-restart-wl-GC-prog-post S T* $\longleftrightarrow$
$(\exists S'\ T'.\ (S,\ S') \in$ *state-wl-l None* $\wedge\ (T,\ T') \in$ *state-wl-l None* $\wedge$
cdcl-twl-full-restart-l-GC-prog-pre S' $\wedge$
*cdcl-twl-restart-l-inp*** *S' T'* $\wedge$ *correct-watching' T* $\wedge$
set-mset (*all-init-lits-of-wl T*) =
set-mset (*all-lits-st T*) $\wedge$

get-unkept-learned-clss-wl T = {#} ∧
get-subsumed-learned-clauses-wl T = {#} ∧
get-learned-clauses0-wl T = {#}
)›

definition *cdcl-twl-full-restart-wl-GC-prog-post-confl* :: ‹*'v twl-st-wl ⇒ 'v twl-st-wl ⇒ bool*› **where**
‹*cdcl-twl-full-restart-wl-GC-prog-post-confl S T ⟷*
(∃ S' T'. (S, S') ∈ state-wl-l None ∧ (T, T') ∈ state-wl-l None ∧
cdcl-twl-full-restart-l-GC-prog-pre S' ∧
*cdcl-twl-restart-l-inp** S' T' ∧*
set-mset (all-init-lits-of-wl T) =
set-mset (all-lits-st T))›

definition (**in** −) *restart-abs-wl-pre2* :: ‹*'v twl-st-wl ⇒ bool ⇒ bool*› **where**
‹*restart-abs-wl-pre2 S brk ⟷*
(∃ S' last-GC last-Restart. (S, S') ∈ state-wl-l None ∧ restart-abs-l-pre S' last-GC last-Restart brk
∧ correct-watching' S ∧ literals-are-$\mathcal{L}_{in}$' S)›

definition (**in** −) *cdcl-twl-local-restart-wl-spec0* :: ‹*'v twl-st-wl ⇒ 'v twl-st-wl nres*› **where**
‹*cdcl-twl-local-restart-wl-spec0 = (λ(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). do {*
ASSERT(restart-abs-wl-pre2 (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) False);
(M, Q) ← SPEC(λ(M', Q'). (∃ K M2. (Decided K # M', M2) ∈ set (get-all-ann-decomposition
M) ∧
Q' = {#} ∧ count-decided M' = 0) ∨ (M' = M ∧ Q' = Q ∧ count-decided M' = 0));
RETURN (M, N, D, NE, UE, NEk, UEk, NS, {#}, N0, {#}, Q, W)
})›

definition *cdcl-twl-full-restart-wl-GC-prog-pre*
:: ‹*'v twl-st-wl ⇒ bool*›
where
‹*cdcl-twl-full-restart-wl-GC-prog-pre S ⟷*
(∃ T. (S, T) ∈ state-wl-l None ∧ correct-watching' S ∧ literals-are-$\mathcal{L}_{in}$' S ∧ cdcl-twl-full-restart-l-GC-prog-pre
T)›

lemma *blits-in-$\mathcal{L}_{in}$'-restart-wl-spec0*:
‹*NO-MATCH {#} f' ⟹*
literals-are-$\mathcal{L}_{in}$' (a, b, c, d, e, NEk, UEk, NS, US, N0, U0, f', g) ⟷
literals-are-$\mathcal{L}_{in}$' (ah, b, c, d, e, NEk, UEk, NS, US, N0, U0, {#}, g)›
by (*auto simp*: *blits-in-$\mathcal{L}_{in}$'-def literals-are-$\mathcal{L}_{in}$'-def*
all-init-lits-def all-init-lits-of-wl-def all-learned-lits-of-wl-def)

lemma *all-init-lits-of-wl-keepUSD*:
‹*L ∈# all-init-lits-of-wl ([], x1k, x1l, x1m, x1n, NEk, UEk, x1o, {#}, x1q, x1r, {#}, x2s) ⟹*
L ∈# all-init-lits-of-wl ([], x1k, x1l, x1m, x1n, NEk, UEk, x1o, {#}, x1q, x1r, Q, x2s)›
by (*auto simp*: *all-init-lits-of-wl-def all-lits-of-mm-def*)

lemma (**in** −)[*twl-st*,*simp*]: ‹*learned-clss (state$_W$-of S) = get-all-learned-clss S*›
by (*cases S*) *auto*

lemma (**in** −)[*twl-st*,*simp*]: ‹*init-clss (state$_W$-of S) = get-all-init-clss S*›
by (*cases S*) *auto*

lemma *literals-are-$\mathcal{L}_{in}$'-empty*:
‹*NO-MATCH {#} x2m ⟹ literals-are-$\mathcal{L}_{in}$' (x1h, x1p, x1j, x1k, b, NEk, UEk, x', x2l, N0, U0, x2m,*
Q) ⟷
literals-are-$\mathcal{L}_{in}$' (x1h, x1p, x1j, x1k, b, NEk, UEk, x', x2l, N0, U0, {#}, Q)›

‹NO-MATCH {#} x2l ⟹ correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟷
correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, {#}, N0, U0, x2m, Q)›
‹NO-MATCH {#} x2m ⟹ correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟷
correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, {#}, Q)›
‹NO-MATCH {#} U0 ⟹ correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟷
correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, x2l, N0, {#}, x2m, Q)›
‹NO-MATCH {#} b ⟹ correct-watching′ (x1h, x1i, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟷
correct-watching′ (x1h, x1i, x1j, x1k, {#}, NEk, UEk, x′, x2l, N0, U0, x2m, Q)›
‹literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟹
literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j, x1k, b, NEk, UEk, x′, {#}, N0, U0, x2m, Q)›
‹literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟹
literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j, x1k, b, NEk, UEk, x′, x2l, N0, {#}, x2m, Q)›
‹literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟹
literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j, x1k, {#}, NEk, UEk, x′, x2l, N0, U0, x2m, Q)›
by (*auto 5 3 simp*: *literals-are-$\mathcal{L}_{in}$′-def blits-in-$\mathcal{L}_{in}$′-def all-lits-of-mm-union correct-watching′.simps correct-watching′.simps clause-to-update-def all-init-lits-of-wl-def all-learned-lits-of-wl-def*)

lemma *literals-are-$\mathcal{L}_{in}$′-decompD*:
‹(K # x1h′, M2) ∈ set (get-all-ann-decomposition x1h) ⟹
literals-are-$\mathcal{L}_{in}$′ (x1h, x1p, x1j′, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q) ⟹
literals-are-$\mathcal{L}_{in}$′ (x1h′, x1p, x1j, x1k, b, NEk, UEk, x′, x2l, N0, U0, x2m, Q)›
by (*auto 5 3 simp*: *literals-are-$\mathcal{L}_{in}$′-def blits-in-$\mathcal{L}_{in}$′-def all-lits-of-mm-union correct-watching′.simps correct-watching′.simps clause-to-update-def all-init-lits-of-wl-def all-learned-lits-of-wl-def dest*!: *get-all-ann-decomposition-exists-prepend*)

lemma *all-init-learned-lits-simps-Q*:
‹NO-MATCH {#} Q ⟹ all-init-lits-of-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =
all-init-lits-of-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, W)›
‹NO-MATCH {#} U0 ⟹ all-init-lits-of-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =
all-init-lits-of-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, {#}, Q, W)›
‹NO-MATCH {#} Q ⟹ all-learned-lits-of-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =
all-learned-lits-of-wl (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, W)›
by (*auto simp*: *all-init-lits-of-wl-def all-learned-lits-of-wl-def all-lits-of-mm-def*)

lemma *in-all-learned-lits-of-wl-addUS*:
‹x ∈ set-mset (all-learned-lits-of-wl (M, x1k, x1l, x1m, x1n, NEk, UEk, x1o, {#}, x1q, x1r, x1s, x2s)) ⟹
x ∈ set-mset (all-learned-lits-of-wl (M, x1k, x1l, x1m, x1n, NEk, UEk, x1o, x1p, x1q, x1r, x1s, x2s))›
‹x ∈ set-mset (all-learned-lits-of-wl (M, x1k, x1l, x1m, x1n, NEk, UEk, x1o, x1p, x1q, {#}, x1s, x2s)) ⟹
x ∈ set-mset (all-learned-lits-of-wl (M, x1k, x1l, x1m, x1n, NEk, UEk, x1o, x1p, x1q, x1r, x1s, x2s))›
‹x ∈ set-mset (all-learned-lits-of-wl (M, x1k, x1l, x1m, {#}, NEk, UEk, x1o, x1p, x1q, x1r, x1s, x2s)) ⟹
x ∈ set-mset (all-learned-lits-of-wl (M, x1k, x1l, x1m, x1n, NEk, UEk, x1o, x1p, x1q, x1r, x1s, x2s))›
by (*auto simp*: *all-learned-lits-of-wl-def all-lits-of-mm-union*)

lemma *cdcl-twl-local-restart-wl-spec0-cdcl-twl-local-restart-l-spec0*:
  ‹$(x,\ y) \in \{(S,\ S').\ (S,\ S') \in$ *state-wl-l None* $\wedge$ *correct-watching'* $S \wedge$ *literals-are-*$\mathcal{L}_{in}$*'* $S\} \Longrightarrow$
    cdcl-twl-local-restart-wl-spec0 x
    $\leq \Downarrow \{(S,\ S').\ (S,\ S') \in$ *state-wl-l None* $\wedge$ *correct-watching'* $S \wedge$ *literals-are-*$\mathcal{L}_{in}$*'* $S\}$
   (*cdcl-twl-local-restart-l-spec0* y)›
  unfolding *cdcl-twl-local-restart-wl-spec0-def cdcl-twl-local-restart-l-spec0-def curry-def*
  apply *refine-vcg*
  subgoal unfolding *restart-abs-wl-pre2-def* **by** (*rule exI*[*of* - y]) *fast*
  subgoal
    by (*auto simp add*: *literals-are-*$\mathcal{L}_{in}$*'-empty*
      state-wl-l-def image-iff correct-watching'.simps clause-to-update-def
     conc-fun-RES RES-RETURN-RES2 blits-in-$\mathcal{L}_{in}$*'-restart-wl-spec0*
     intro: *literals-are-*$\mathcal{L}_{in}$*'-decompD literals-are-*$\mathcal{L}_{in}$*'-empty*(*4*))
  subgoal
    by (*auto 4 3 simp add*: *literals-are-*$\mathcal{L}_{in}$*'-empty*
      state-wl-l-def image-iff correct-watching'.simps clause-to-update-def
     conc-fun-RES RES-RETURN-RES2 blits-in-$\mathcal{L}_{in}$*'-restart-wl-spec0*
     literals-are-$\mathcal{L}_{in}$*'-def all-init-learned-lits-simps-Q blits-in-*$\mathcal{L}_{in}$*'-def*
     dest: *all-init-lits-of-wl-keepUSD*
     in-all-learned-lits-of-wl-addUS)
  done

lemma *cdcl-twl-full-restart-wl-GC-prog-post-correct-watching*:
  assumes
    pre: ‹*cdcl-twl-full-restart-l-GC-prog-pre* y› **and**
    y-Va: ‹*cdcl-twl-restart-l-inp*** y *Va*› **and**
    ‹$(V,\ Va) \in \{(S,\ S').\ (S,\ S') \in$ *state-wl-l None* $\wedge$ *correct-watching'* $S \wedge$ *literals-are-*$\mathcal{L}_{in}$*'* $S\}$›
  shows ‹$(V,\ Va) \in \{(S,\ S').\ (S,\ S') \in$ *state-wl-l None* $\wedge$ *correct-watching* $S \wedge$ *blits-in-*$\mathcal{L}_{in}$ $S\}$› **and**
    ‹*set-mset* (*all-init-lits-of-wl* V) = *set-mset* (*all-lits-st* V)›
proof −
  obtain x **where**
    y-x: ‹$(y,\ x) \in$ *twl-st-l None*› **and**
    struct-invs: ‹*twl-struct-invs* x› **and**
    list-invs: ‹*twl-list-invs* y› **and**
    ent: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of* x)›
    using *pre* **unfolding** *cdcl-twl-full-restart-l-GC-prog-pre-def* **by** *blast*
  obtain V' **where** ‹*cdcl-twl-inp*** x V'› **and** *Va-V'*: ‹$(Va,\ V') \in$ *twl-st-l None*›
    using *rtranclp-cdcl-twl-restart-l-inp-cdcl-twl-restart-inp*[*OF y-Va y-x list-invs struct-invs ent*]
    by *blast*
  then have ‹*twl-struct-invs* V'›
    using *struct-invs ent rtranclp-cdcl-twl-inp-twl-struct-invs* **by** *blast*
  then show *eq*: ‹*set-mset* (*all-init-lits-of-wl* V) = *set-mset* (*all-lits-st* V)›
    using *assms*(*3*) *Va-V'* ‹*twl-struct-invs* V'› *literals-are-*$\mathcal{L}_{in}$*'-literals-are-*$\mathcal{L}_{in}$*-iff*(*4*) **by** *blast*
  then have ‹*correct-watching'* $V \Longrightarrow$ *correct-watching* V›
    by (*cases* V) (*auto simp*: *correct-watching.simps correct-watching'.simps*)
  moreover
    have ‹*literals-are-*$\mathcal{L}_{in}$*'* $V \Longrightarrow$ *blits-in-*$\mathcal{L}_{in}$ V›
    using *eq* **by** (*cases* V)
      (*clarsimp simp*: *blits-in-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*'-def all-lits-def literals-are-*$\mathcal{L}_{in}$*'-def*
        all-init-lits-def ac-simps)
  ultimately show ‹$(V,\ Va) \in \{(S,\ S').\ (S,\ S') \in$ *state-wl-l None* $\wedge$ *correct-watching* $S \wedge$ *blits-in-*$\mathcal{L}_{in}$ $S\}$›
    using *assms* **by** (*auto simp*: *cdcl-twl-full-restart-wl-GC-prog-post-def*)
qed

lemma $\mathcal{L}_{all}$*-all-init-atms-all-init-lits*:
‹*set-mset* ($\mathcal{L}_{all}$ (*all-init-atms N NE*)) = *set-mset* (*all-init-lits N NE*)›
unfolding $\mathcal{L}_{all}$*-all-init-atms* **..**
end
theory *Watched-Literals-Watch-List-Reduce*
imports *Watched-Literals-List-Simp Watched-Literals-List-Reduce Watched-Literals-Watch-List*
Watched-Literals-Watch-List-Restart
begin
no-notation *funcset* (**infixr** $\rightarrow$ *60*)

lemma *GC-remap-all-init-atmsD*:
‹*GC-remap* (*N*, *x*, *m*) (*N'*, *x'*, *m'*) $\Longrightarrow$
all-init-atms N NE + *all-init-atms m NE* = *all-init-atms N' NE* + *all-init-atms m' NE*›
by (*induction rule*: *GC-remap.induct*[*split-format*(*complete*)])
(*auto simp*: *all-init-atms-def all-init-lits-def init-clss-l-fmdrop-if*
init-clss-l-fmupd-if image-mset-remove1-mset-if
simp del: *all-init-atms-def*[*symmetric*]
simp flip: *image-mset-union all-lits-of-mm-add-mset all-lits-of-mm-union*)

lemma *rtranclp-GC-remap-all-init-atmsD*:
‹*GC-remap*** (*N*, *x*, *m*) (*N'*, *x'*, *m'*) $\Longrightarrow$
all-init-atms N NE + *all-init-atms m NE* = *all-init-atms N' NE* + *all-init-atms m' NE*›
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*: *GC-remap-all-init-atmsD*)

lemma *rtranclp-GC-remap-all-init-atms*:
‹*GC-remap*** (*x1a*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *x1ad*) $\Longrightarrow$
all-init-atms x1ad NE = *all-init-atms x1a NE*›
by (*auto dest*!: *rtranclp-GC-remap-all-init-atmsD*[*of - - - - - - NE*])

lemma *GC-remap-all-init-lits*:
‹*GC-remap* (*N*, *m*, *new*) (*N'*, *m'*, *new'*) $\Longrightarrow$
all-init-lits N NE + *all-init-lits new NE* = *all-init-lits N' NE* + *all-init-lits new' NE*›
by (*induction rule*: *GC-remap.induct*[*split-format*(*complete*)])
(*case-tac* ‹*irred N C*› ; *auto simp*: *all-init-lits-def init-clss-l-fmupd-if image-mset-remove1-mset-if*
simp flip: *all-lits-of-mm-union*)

lemma *rtranclp-GC-remap-all-init-lits*:
‹*GC-remap*** (*N*, *m*, *new*) (*N'*, *m'*, *new'*) $\Longrightarrow$
all-init-lits N NE + *all-init-lits new NE* = *all-init-lits N' NE* + *all-init-lits new' NE*›
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*: *GC-remap-all-init-lits*)

lemma *subsumed-clauses-alt-def*:
‹*subsumed-clauses S* = *subsumed-init-clauses S* + *subsumed-learned-clauses S*›
by (*cases S*) *auto*

definition *drop-clause-add-move-init-wl* :: ‹*'v twl-st-wl* $\Rightarrow$ *nat* $\Rightarrow$ *'v twl-st-wl*› **where**
‹*drop-clause-add-move-init-wl* = (λ(*M*, *N*, *D*, *NE0*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) *C*.
(*M*, *fmdrop C N*, *D*, *add-mset* (*mset* (*N* $\propto$ *C*)) *NE0*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*))›

definition *drop-clause-wl* :: ‹*'v twl-st-wl* $\Rightarrow$ *nat* $\Rightarrow$ *'v twl-st-wl*› **where**
‹*drop-clause-wl* = (λ(*M*, *N*, *D*, *NE0*, *UE*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) *C*.
(*M*, *fmdrop C N*, *D*, *NE0*, *UE*, *NS*, {#}, *N0*, *U0*, *Q*, *W*))›

lemma *reduce-dom-clauses-fmdrop*:

‹reduce-dom-clauses N0 N ⟹ reduce-dom-clauses N0 (fmdrop C N)›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *reduce-dom-clauses-def distinct-mset-remove1-All*)

lemma *correct-watching-fmdrop*:
assumes
irred: ‹¬ *irred N C*› **and**
C: ‹*C* ∈# *dom-m N*› **and**
‹*correct-watching'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
C2: ‹*length* (*N* ∝ *C*) ≠ *2*› **and**
blit: ‹*literals-are-*$\mathcal{L}_{in}$*'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
shows ‹*correct-watching'* (*M*, *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*)› **and**
‹*literals-are-*$\mathcal{L}_{in}$*'* (*M*, *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*)›
proof −
let *?S* = ‹(*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
have
Hdist: ‹⋀*L i K b*. *L*∈#*all-init-lits-of-wl ?S* ⟹
distinct-watched (*W L*)› **and**
H1: ‹⋀*L i K b*. *L*∈#*all-init-lits-of-wl ?S* ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)› **and**
H1': ‹⋀*L i K b*. *L*∈#*all-init-lits-of-wl ?S* ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *b* ⟹ *i* ∈# *dom-m N*› **and**
H2: ‹⋀*L*. *L*∈# *all-init-lits-of-wl ?S* ⟹
{#*i* ∈# *fst* '# *mset* (*W L*). *i* ∈# *dom-m N*#} =
{#*C* ∈# *dom-m* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})).
L ∈ *set* (*watched-l* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})
∝ *C*))#}›
using *assms*
unfolding *correct-watching'.simps clause-to-update-def*
by *fast*+
have *1*: ‹{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =
{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule filter-mset-cong2*)
using *distinct-mset-dom*[*of N*] *C irred*
by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
have *2*: ‹*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} =
removeAll-mset C {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule distinct-mset-remove1-All*)
using *distinct-mset-dom*[*of N*]
by (*auto intro*: *distinct-mset-filter*)
have [*simp*]: ‹*filter-mset* (λ*i*. *i* ≠ *C* ∧ *i* ∈# (*dom-m N*)) *A* =
removeAll-mset C (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)› **for** *A*
by (*induction A*)
(*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
show ‹*correct-watching'* (*M*, *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*)›
unfolding *correct-watching'.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L* **using** *Hdist*[*of L*] *distinct-mset-dom*[*of N*]
H1[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
H1'[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›]
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)

done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
H1'[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›]
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
H1'[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C2*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
have [*dest*!]: ‹*x* ∈# *all-learned-lits-of-wl* ([], *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*) ⟹
x ∈# *all-learned-lits-of-wl* ([], *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **for** *x*
using *assms*(*1*,*2*)
by (*auto dest*: *all-lits-of-mm-diffD simp*: *all-learned-lits-of-wl-def all-lits-of-mm-union image-mset-remove1-mset-if*)

show ‹*literals-are-*$\mathcal{L}_{in}$*'* (*M*, *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*)›
using *assms* **by** (*auto simp*: *blits-in-*$\mathcal{L}_{in}$*'-def image-mset-remove1-mset-if literals-are-*$\mathcal{L}_{in}$*'-def all-init-lits-def all-lits-of-mm-union dest*: *multi-member-split all-lits-of-mm-diffD*)
qed

lemma *correct-watching'-fmdrop*:
assumes
irred: ‹¬ *irred N C*› **and**
C: ‹*C* ∈# *dom-m N*› **and**
‹*correct-watching'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
‹*literals-are-*$\mathcal{L}_{in}$*'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
shows ‹*correct-watching'-nobin* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›**and**
‹*literals-are-*$\mathcal{L}_{in}$*'* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›
proof −
let *?S* = ‹(*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
let *?*$\mathcal{L}$ = ‹*all-init-lits-of-wl ?S*›
have
Hdist: ‹⋀*L i K b*. *L*∈#*?*$\mathcal{L}$ ⟹
distinct-watched (*W L*)› **and**
H1: ‹⋀*L i K b*. *L*∈#*?*$\mathcal{L}$ ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)› **and**
H2: ‹⋀*L*. *L*∈#*?*$\mathcal{L}$ ⟹
{#*i* ∈# *fst* ‘# *mset* (*W L*). *i* ∈# *dom-m N*#} =

{#C ∈# dom-m (get-clauses-l (M', N, D, NE, UE, NEk, UEk, NS, US,N0, U0, {#}, {#})).
L ∈ set (watched-l (get-clauses-l (M', N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, {#})
∝ C))#}›
using assms
unfolding correct-watching'.simps clause-to-update-def
by fast+
have 1: ‹{#Ca ∈# dom-m (fmdrop C N). L ∈ set (watched-l (fmdrop C N ∝ Ca))#} =
{#Ca ∈# dom-m (fmdrop C N). L ∈ set (watched-l (N ∝ Ca))#}› for L
apply (rule filter-mset-cong2)
using distinct-mset-dom[of N] C irred
by (auto simp: image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset
distinct-mset-remove1-All filter-mset-neq-cond dest: all-lits-of-mm-diffD
dest: multi-member-split)
have 2: ‹remove1-mset C {#Ca ∈# dom-m N. L ∈ set (watched-l (N ∝ Ca))#} =
removeAll-mset C {#Ca ∈# dom-m N. L ∈ set (watched-l (N ∝ Ca))#}› for L
apply (rule distinct-mset-remove1-All)
using distinct-mset-dom[of N]
by (auto intro: distinct-mset-filter)
have [simp]: ‹filter-mset (λi. i ≠ C ∧ i ∈# dom-m N) A =
removeAll-mset C (filter-mset (λi. i ∈# dom-m N) A)› for A
by (induction A)
(auto simp: distinct-mset-remove1-All distinct-mset-dom)
show ‹correct-watching'-nobin (M, fmdrop C N, D, NE, UE, NEk, UEk, NS, {#}, N0, U0, Q, W)›
unfolding correct-watching'-nobin.simps clause-to-update-def
apply (intro conjI impI ballI)
subgoal for L using Hdist[of L] distinct-mset-dom[of N]
H1[of L ‹fst iK› ‹fst (snd iK)› ‹snd (snd iK)›] C irred
apply (auto simp: image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: all-lits-of-mm-diffD
dest: multi-member-split)
done
subgoal for L iK
using distinct-mset-dom[of N] H1[of L ‹fst iK› ‹fst (snd iK)› ‹snd (snd iK)›] C irred
apply (auto simp: image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: all-lits-of-mm-diffD
dest: multi-member-split)
done
subgoal for L
using C irred apply −
unfolding get-clauses-l.simps
apply (subst 1)
by (auto 5 1 simp: image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset
distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest: all-lits-of-mm-diffD
dest: multi-member-split)
done
have [dest!]: ‹x ∈# all-learned-lits-of-wl ([], fmdrop C N, D, NE, UE, NEk, UEk, NS, {#}, N0, U0,
Q, W) ⟹
x ∈# all-learned-lits-of-wl ([], N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)› for x
using assms
by (auto dest: all-lits-of-mm-diffD simp: all-learned-lits-of-wl-def
all-lits-of-mm-union image-mset-remove1-mset-if)
show ‹literals-are-$\mathcal{L}_{in}$' (M, fmdrop C N, D, NE, UE, NEk, UEk, NS, {#}, N0, U0, Q, W)›
using assms by (auto simp: blits-in-$\mathcal{L}_{in}$'-def image-mset-remove1-mset-if literals-are-$\mathcal{L}_{in}$'-def
all-init-lits-def all-lits-of-mm-union
dest: multi-member-split all-lits-of-mm-diffD)
qed

lemma *all-init-lits-of-wl-fmdrop-addNEk*[*simp*]:
  ‹*irred N C* ⟹ *C* ∈# *dom-m N* ⟹
  (*all-init-lits-of-wl* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *add-mset* (*mset* (*N* ∝ *C*)) *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)) =
  (*all-init-lits-of-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))›
  using *distinct-mset-dom*[*of N*]
  by (*auto simp*: *all-init-lits-of-wl-def ran-m-def dest*!: *multi-member-split*
    intro!: *arg-cong*[*of* - - *all-lits-of-mm*]
    intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *correct-watching'-fmdrop'*:
  assumes
    irred: ‹*irred N C*› **and**
    C: ‹*C* ∈# *dom-m N*› **and**
    ‹*correct-watching'-nobin* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›**and**
    ‹*literals-are-*$\mathcal{L}_{in}$*'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
  shows ‹*correct-watching'-nobin* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *add-mset* (*mset* (*N* ∝ *C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)› **and**
    ‹*literals-are-*$\mathcal{L}_{in}$*'* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *add-mset* (*mset* (*N* ∝ *C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›
proof −
  let *?S* = ‹(*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
  let *?*$\mathcal{L}$ = ‹*all-init-lits-of-wl ?S*›
  have
    Hdist: ‹⋀*L*. *L*∈#*?*$\mathcal{L}$ ⟹
      distinct-watched (*W L*)› **and**
    H1: ‹⋀*L i K b*. *L*∈#*?*$\mathcal{L}$ ⟹
      (*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹
        K ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)› **and**
    H2: ‹⋀*L*. *L*∈#*?*$\mathcal{L}$ ⟹
      {#*i* ∈# *fst* ‘# *mset* (*W L*). *i* ∈# *dom-m N*#} =
      {#*C* ∈# *dom-m* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})).
       L ∈ *set* (*watched-l* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}) ∝ *C*))#}›
    using *assms*
    unfolding *correct-watching'-nobin.simps clause-to-update-def*
    by *blast*+
  have *1*: ‹{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =
    {#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
    apply (*rule filter-mset-cong2*)
      using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*snd iK*›] *C irred*
    by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
      distinct-mset-remove1-All filter-mset-neq-cond dest: *all-lits-of-mm-diffD*
        dest: *multi-member-split*)
  have *2*: ‹*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} =
     removeAll-mset C {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
    apply (*rule distinct-mset-remove1-All*)
    using *distinct-mset-dom*[*of N*]
    by (*auto intro*: *distinct-mset-filter*)
  have [*simp*]: ‹*filter-mset* (λ*i*. *i* ≠ *C* ∧ *i* ∈# (*dom-m N*)) *A* =
    removeAll-mset C (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)› **for** *A*
    by (*induction A*)
      (*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
  show ‹*correct-watching'-nobin* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *add-mset* (*mset* (*N* ∝ *C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›

unfolding *correct-watching'-nobin.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
Hdist[*of L*] *irred C*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
have [*dest*!]: ‹*x* ∈# *all-learned-lits-of-wl* ([], *N*, *D*, *NE*, *UE*, *add-mset* (*mset* (*N* ∝ *C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*) ⟹
x ∈# *all-learned-lits-of-wl* ([], *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **for** *x*
using *assms*
by (*auto dest*: *all-lits-of-mm-diffD simp*: *all-learned-lits-of-wl-def*
all-lits-of-mm-union image-mset-remove1-mset-if)
have ‹(*N* ∝ *C*, *irred N C*) ∈# (*init-clss-l N*)›
using *assms* **by** (*auto simp*: *ran-m-def dest*!: *multi-member-split*) (*metis prod.collapse*)
from *multi-member-split*[*OF this*]
show ‹*literals-are-*$\mathcal{L}_{in}$*'* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *add-mset* (*mset* (*N* ∝ *C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›
using *distinct-mset-dom*[*of N*]
using *assms* **by** (*auto simp*: *blits-in-*$\mathcal{L}_{in}$*'-def image-mset-remove1-mset-if all-lits-of-mm-add-mset*
all-lits-of-mm-union literals-are-$\mathcal{L}_{in}$*'-def all-init-lits-def*
dest: *multi-member-split all-lits-of-mm-diffD*)
qed

lemma *correct-watching'-fmdrop''*:
assumes
irred: ‹¬*irred N C*› **and**
C: ‹*C* ∈# *dom-m N*› **and**
‹*correct-watching'-nobin* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)› **and**
‹*literals-are-*$\mathcal{L}_{in}$*'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
shows ‹*correct-watching'-nobin* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *add-mset* (*mset* (*N* ∝ *C*)) *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)› **and**
‹*literals-are-*$\mathcal{L}_{in}$*'* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *add-mset* (*mset* (*N* ∝ *C*)) *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›
proof −
let *?S* = ‹(*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
let *?*$\mathcal{L}$ = ‹*all-init-lits-of-wl ?S*›
have
Hdist: ‹$\bigwedge$*L*. *L*∈#*?*$\mathcal{L}$ ⟹

distinct-watched (*W L*)› **and**
H1: ‹⋀*L i K b*. *L*∈#*?*ℒ ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)› **and**
H2: ‹⋀*L*. *L*∈#*?*ℒ ⟹
{#*i* ∈# *fst* ‘# *mset* (*W L*). *i* ∈# *dom-m N*#} =
{#*C* ∈# *dom-m* (*get-clauses-l* (*M′*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#})).
L ∈ *set* (*watched-l* (*get-clauses-l* (*M′*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}) ∝ *C*))#}›
using *assms*
unfolding *correct-watching′-nobin.simps clause-to-update-def*
by *blast+*
have *1*: ‹{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =
{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule filter-mset-cong2*)
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*snd iK*›] *C irred*
by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
have *2*: ‹*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} = *removeAll-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule distinct-mset-remove1-All*)
using *distinct-mset-dom*[*of N*]
by (*auto intro*: *distinct-mset-filter*)
have [*simp*]: ‹*filter-mset* (λ*i*. *i* ≠ *C* ∧ *i* ∈# *dom-m N*) *A* = *removeAll-mset C* (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)› **for** *A*
by (*induction A*)
(*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
show ‹*correct-watching′-nobin* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *add-mset* (*mset* (*N* ∝ *C*)) *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*)›
unfolding *correct-watching′-nobin.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
Hdist[*of L*] *assms*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done

have [*dest*!]: ‹*x* ∈# *all-learned-lits-of-wl* ([], *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *add-mset* (*mset* (*N* ∝ *C*)) *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*) ⟹

```
      x ∈# all-learned-lits-of-wl ([], N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)› for x
    using assms
    by (auto dest: all-lits-of-mm-diffD simp: all-learned-lits-of-wl-def
      all-lits-of-mm-union image-mset-remove1-mset-if)
  have ‹(N ∝ C, irred N C) ∈# (learned-clss-l N)›
    using assms by (auto simp: ran-m-def dest!: multi-member-split) (metis prod.collapse)
  from multi-member-split[OF this]
  show ‹literals-are-ℒ_in' (M, fmdrop C N, D, NE, UE, NEk, add-mset (mset (N ∝ C)) UEk, NS, {#},
N0, U0, Q, W)›
    using distinct-mset-dom[of N]
    using assms by (auto simp: blits-in-ℒ_in'-def image-mset-remove1-mset-if all-lits-of-mm-add-mset
        all-lits-of-mm-union literals-are-ℒ_in'-def all-init-lits-def
      dest: multi-member-split all-lits-of-mm-diffD)
qed

definition remove-one-annot-true-clause-one-imp-wl-pre where
  ‹remove-one-annot-true-clause-one-imp-wl-pre i T ⟷
    (∃ T'. (T, T') ∈ state-wl-l None ∧
      remove-one-annot-true-clause-one-imp-pre i T' ∧
      correct-watching'-nobin T ∧ literals-are-ℒ_in' T)›

definition replace-annot-wl-pre :: ‹'v literal ⇒ nat ⇒ 'v twl-st-wl ⇒ bool› where
‹replace-annot-wl-pre L C S ⟷
  (∃ S'. (S, S') ∈ state-wl-l None ∧ L ∈# all-init-lits-of-wl S ∧
    replace-annot-l-pre L C S' ∧ literals-are-ℒ_in' S ∧
    correct-watching'-nobin S)›

definition replace-annot-wl :: ‹'v literal ⇒ nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
  ‹replace-annot-wl L C =
    (λ(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W). do {
      ASSERT(replace-annot-wl-pre L C (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W));
      RES {(M', N, D, NE, {#}, NEk, UEk, NS, {#}, N0, {#}, Q, W)| M'.
       (∃ M2 M1 C. M = M2 @ Propagated L C # M1 ∧ M' = M2 @ Propagated L 0 # M1)}
    })›

lemma replace-annot-l-pre-replace-annot-wl-pre: ‹(((L, C), S), (L', C'), S')
    ∈ Id ×_f nat-rel ×_f
      {(S, T).
       (S, T) ∈ state-wl-l None ∧
       correct-watching'-nobin S ∧ literals-are-ℒ_in' S} ⟹
    replace-annot-l-pre L' C' S' ⟹
    replace-annot-wl-pre L C S›
    unfolding replace-annot-wl-pre-def replace-annot-l-pre-alt-def
    unfolding replace-annot-l-pre-def[symmetric]
    by (rule exI[of - ‹S'›])
      (auto simp add: ac-simps all-init-lits-of-wl-def)

lemma all-learned-lits-of-wl-all-init-lits-of-wlD[intro]:
  ‹set-mset (all-learned-lits-of-wl (M, ab, ac, ad, ae, NEk, UEk, af, ag, ah, ai, aj, ba))
      ⊆ set-mset (all-init-lits-of-wl (M, ab, ac, ad, {#}, NEk, {#}, af, {#}, ah, {#}, aj, ba)) ⟹
      x ∈# all-learned-lits-of-wl (M, ab, ac, ad, {#}, NEk, UEk, af, {#}, ah, {#}, aj, ba) ⟹
      x ∈# all-init-lits-of-wl (M, ab, ac, ad, {#}, NEk, {#}, af, {#}, ah, {#}, aj, ba)›
  by (auto simp: all-learned-lits-of-wl-def all-init-lits-of-wl-def
    all-lits-of-mm-union)

lemma replace-annot-wl-replace-annot-l:
```

```
  ‹(uncurry2 replace-annot-wl, uncurry2 replace-annot-l) ∈
    Id ×f nat-rel ×f {(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'-nobin S ∧
        literals-are-ℒin' S} →f
    ⟨{(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'-nobin S ∧
        literals-are-ℒin' S}⟩nres-rel›
    unfolding replace-annot-wl-def replace-annot-l-def uncurry-def
    apply (intro frefI nres-relI)
    apply clarify
    apply refine-rcg
    subgoal for a b aa ab ac ad ae af ba ag bb ah ai aj ak al am bc
      by (force intro!: replace-annot-l-pre-replace-annot-wl-pre)
    subgoal
      by (rule RES-refine)
       (force simp: state-wl-l-def literals-are-ℒin'-def ac-simps
          all-lits-of-mm-union
        correct-watching'-nobin.simps clause-to-update-def blits-in-ℒin'-def)
    done

definition remove-and-add-cls-wl :: ‹nat ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres› where
  ‹remove-and-add-cls-wl C =
    (λ(M, N, D, NE, UE, NEk, UEk, NS, US, Q, W). do {
       ASSERT(C ∈# dom-m N);
       RETURN (M, fmdrop C N, D, NE, UE,
        (if irred N C then add-mset (mset (N∝C)) else id) NEk,
  (if ¬irred N C then add-mset (mset (N∝C)) else id) UEk, NS, {#}, Q, W)
    })›

definition remove-one-annot-true-clause-one-imp-wl
  :: ‹nat ⇒ 'v twl-st-wl ⇒ (nat × 'v twl-st-wl) nres›
where
‹remove-one-annot-true-clause-one-imp-wl = (λi S. do {
      ASSERT(remove-one-annot-true-clause-one-imp-wl-pre i S);
      ASSERT(is-proped (rev (get-trail-wl S) ! i));
      (L, C) ← SPEC(λ(L, C). (rev (get-trail-wl S))!i = Propagated L C);
      ASSERT(Propagated L C ∈ set (get-trail-wl S));
      ASSERT(L ∈# all-init-lits-of-wl S);
      if C = 0 then RETURN (i+1, S)
      else do {
        ASSERT(C ∈# dom-m (get-clauses-wl S));
 S ← replace-annot-wl L C S;
 S ← remove-and-add-cls-wl C S;
        RETURN (i+1, S)
      }
  })›

lemma
  assumes
      x2-T: ‹(x2, T) ∈ state-wl-l b› and
      struct: ‹twl-struct-invs U› and
      T-U: ‹(T, U) ∈ twl-st-l b'›
  shows
    literals-are-ℒin-literals-are-ℒin-trail-init:
      ‹literals-are-in-ℒin-trail (all-init-atms-st x2) (get-trail-wl x2)›
     (is ‹?trail›)
proof −
  have
```

alien: ‹$cdcl_W$-*restart-mset.no-strange-atm* ($state_W$-*of U*)› **and**
confl: ‹$cdcl_W$-*restart-mset.cdcl*$_W$-*conflicting* ($state_W$-*of U*)› **and**
M-lev: ‹$cdcl_W$-*restart-mset.cdcl*$_W$-*M-level-inv* ($state_W$-*of U*)› **and**
dist: ‹$cdcl_W$-*restart-mset.distinct-cdcl*$_W$-*state* ($state_W$-*of U*)›
using *struct* **unfolding** *twl-struct-invs-def* $cdcl_W$-*restart-mset.cdcl*$_W$-*all-struct-inv-def*
pcdcl-all-struct-invs-def $state_W$-*of-def*
by *fast+*

show *lits-trail*: ‹*literals-are-in-*$\mathcal{L}_{in}$-*trail* (*all-init-atms-st x2*) (*get-trail-wl x2*)›
using *alien x2-T T-U* **unfolding** *is-*$\mathcal{L}_{all}$-*def*
literals-are-in-$\mathcal{L}_{in}$-*trail-def* $cdcl_W$-*restart-mset.no-strange-atm-def*
literals-are-$\mathcal{L}_{in}$-*def all-init-lits-of-wl-def all-atms-def* $\mathcal{L}_{all}$-*all-init-atms*
by
(*auto 5 2*
simp del: *all-clss-l-ran-m union-filter-mset-complement*
simp: *twl-st twl-st-l twl-st-wl all-lits-of-mm-union lits-of-def*
convert-lits-l-def image-image in-all-lits-of-mm-ain-atms-of-iff
get-unit-clauses-wl-alt-def $\mathcal{L}_{all}$-*all-init-atms all-init-lits-def*)
qed

lemma *remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp*:
‹(*uncurry remove-one-annot-true-clause-one-imp-wl*, *uncurry remove-one-annot-true-clause-one-imp*)
∈ *nat-rel* ×$_f$ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching'-nobin S* ∧ *literals-are-*$\mathcal{L}_{in}$*' S*}
→$_f$
⟨*nat-rel* ×$_f$ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching'-nobin S* ∧ *literals-are-*$\mathcal{L}_{in}$*' S*}⟩*nres-rel*›
(**is** ‹- ∈ - ×$_f$ *?A* →$_f$ -›)
proof −

have [*refine0*]: ‹*remove-and-add-cls-wl C S* ≤⇓ *?A* (*remove-and-add-cls-l C' S'*)›
if ‹(*C*, *C'*) ∈ *Id*› **and** ‹(*S*, *S'*) ∈ *?A*›
for *C C' S S'*
using *that* **unfolding** *remove-and-add-cls-l-def remove-and-add-cls-wl-def*
by *refine-rcg*
(*auto intro*!: *RES-refine simp*: *state-wl-l-def*
intro: *correct-watching'-fmdrop correct-watching'-fmdrop''*
correct-watching'-fmdrop')

show *?thesis*
unfolding *remove-one-annot-true-clause-one-imp-wl-def remove-one-annot-true-clause-one-imp-def*
uncurry-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg replace-annot-wl-replace-annot-l*[*THEN fref-to-Down-curry2*])
subgoal for *x y* **unfolding** *remove-one-annot-true-clause-one-imp-wl-pre-def*
by (*rule exI*[*of* - ‹*snd y*›]) *auto*
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal for *x y x1 x2 x1a x2a xa x' x1b x2b x1c x2c*
unfolding *remove-one-annot-true-clause-one-imp-wl-pre-def*
remove-one-annot-true-clause-one-imp-pre-def
apply *normalize-goal+*

subgoal for *U S T*
using *literals-are-*$\mathcal{L}_{in}$*-literals-are-*$\mathcal{L}_{in}$*-trail-init*[*of x2a U None T None*]
by (*auto simp add*: *literals-are-in-*$\mathcal{L}_{in}$*-trail-def lits-of-def image-image*

```
      \mathcal{L}_{all}-all-init-atms)
    done
  subgoal by simp
  subgoal by (simp add: state-wl-l-def)
  subgoal by (simp add: state-wl-l-def)
  subgoal by (simp add: state-wl-l-def)
  subgoal by simp
  subgoal by (simp add: state-wl-l-def)
  done
qed

definition remove-one-annot-true-clause-imp-wl-inv where
  ‹remove-one-annot-true-clause-imp-wl-inv S = (λ(i, T).
    (∃ S' T'. (S, S') ∈ state-wl-l None ∧ (T, T') ∈ state-wl-l None ∧
      correct-watching'-nobin S ∧ correct-watching'-nobin T ∧ literals-are-\mathcal{L}_{in}' S ∧
      literals-are-\mathcal{L}_{in}' T ∧
      remove-one-annot-true-clause-imp-inv S' (i, T')))›

definition remove-all-learned-subsumed-clauses-wl :: ‹'v twl-st-wl ⇒ ('v twl-st-wl) nres› where
‹remove-all-learned-subsumed-clauses-wl = (λ(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W).
  RETURN (M, N, D, NE, {#}, NEk, UEk, NS, {#}, N0, {#}, Q, W))›

definition remove-one-annot-true-clause-imp-wl :: ‹'v twl-st-wl ⇒ ('v twl-st-wl) nres›
where
‹remove-one-annot-true-clause-imp-wl = (λS. do {
  k ← SPEC(λk. (∃ M1 M2 K. (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (get-trail-wl
S)) ∧
      count-decided M1 = 0 ∧ k = length M1)
    ∨ (count-decided (get-trail-wl S) = 0 ∧ k = length (get-trail-wl S)));
  start ← SPEC (λi. i ≤ k ∧ (∀ j < i. is-proped (rev (get-trail-wl S) ! j) ∧ mark-of (rev (get-trail-wl
S) ! j) = 0));
  (i, T) ← WHILE_T^{remove-one-annot-true-clause-imp-wl-inv S}
    (λ(i, S). i < k)
    (λ(i, S). remove-one-annot-true-clause-one-imp-wl i S)
    (start, S);
  ASSERT (remove-one-annot-true-clause-imp-wl-inv S (i, T));
  remove-all-learned-subsumed-clauses-wl T
})›

lemma remove-all-learned-subsumed-clauses-wl-remove-all-learned-subsumed-clauses:
  ‹(remove-all-learned-subsumed-clauses-wl, remove-all-learned-subsumed-clauses)
  ∈ {(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'-nobin S ∧ literals-are-\mathcal{L}_{in}' S} →_f
    ⟨{(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'-nobin S ∧ literals-are-\mathcal{L}_{in}' S}⟩nres-rel›
  apply (auto intro!: frefI nres-relI
    simp: remove-all-learned-subsumed-clauses-wl-def remove-all-learned-subsumed-clauses-def
      literals-are-\mathcal{L}_{in}'-def correct-watching'-nobin.simps state-wl-l-def all-init-learned-lits-simps-Q
    clause-to-update-def all-lits-of-mm-union blits-in-\mathcal{L}_{in}'-def)
  by (meson basic-trans-rules(31) in-all-learned-lits-of-wl-addUS)

lemma remove-one-annot-true-clause-imp-wl-remove-one-annot-true-clause-imp:
  ‹(remove-one-annot-true-clause-imp-wl, remove-one-annot-true-clause-imp)
    ∈ {(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'-nobin S ∧ literals-are-\mathcal{L}_{in}' S} →_f
      ⟨{(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'-nobin S ∧ literals-are-\mathcal{L}_{in}' S}⟩nres-rel›
proof −
  show ?thesis
    unfolding remove-one-annot-true-clause-imp-wl-def remove-one-annot-true-clause-imp-def
```

uncurry-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
WHILEIT-refine[**where**
$R = \langle$*nat-rel* $\times_f$ $\{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching'-nobin S* $\wedge$
literals-are-$\mathcal{L}_{in}$*' S*$\}\rangle$]
remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp[*THEN*
fref-to-Down-curry]
remove-all-learned-subsumed-clauses-wl-remove-all-learned-subsumed-clauses[*THEN*
fref-to-Down])
subgoal by *force*
subgoal by *auto*
subgoal by *force*
subgoal by *force*
subgoal by *force*
subgoal for *x y k ka xa start start' x'*
unfolding *remove-one-annot-true-clause-imp-wl-inv-def*
apply (*subst case-prod-beta*)
apply (*rule-tac x=*$\langle$*y*$\rangle$ **in** *exI*)
apply (*rule-tac x=*$\langle$*snd x'*$\rangle$ **in** *exI*)
apply (*subst* (*asm*)(*23*) *surjective-pairing*)
apply (*subst* (*asm*)(*28*) *surjective-pairing*)
unfolding *prod-rel-iff* **by** *simp*
subgoal by *auto*
subgoal by *auto*
subgoal for *x y k ka xa start start' x'*
unfolding *remove-one-annot-true-clause-imp-wl-inv-def*
apply (*subst case-prod-beta*)
apply (*rule-tac x=*$\langle$*y*$\rangle$ **in** *exI*)
apply (*rule-tac x=*$\langle$*snd x'*$\rangle$ **in** *exI*)
apply (*subst* (*asm*)(*23*) *surjective-pairing*)
apply (*subst* (*asm*)(*28*) *surjective-pairing*)
unfolding *prod-rel-iff* **by** *simp*
subgoal by *auto*
done
qed

definition *collect-valid-indices-wl* :: $\langle$*'v twl-st-wl* $\Rightarrow$ *nat list nres*$\rangle$ **where**
$\langle$*collect-valid-indices-wl S* = *SPEC* $(\lambda N.\ True)\rangle$

definition *mark-to-delete-clauses-wl-inv*
:: $\langle$*'v twl-st-wl* $\Rightarrow$ *nat list* $\Rightarrow$ *nat* $\times$ *'v twl-st-wl*$\times$ *nat list* $\Rightarrow$ *bool*$\rangle$
where
$\langle$*mark-to-delete-clauses-wl-inv* $= (\lambda S\ xs0\ (i,\ T,\ xs).$
$\exists S'\ T'.\ (S,\ S') \in$ *state-wl-l None* $\wedge$ $(T,\ T') \in$ *state-wl-l None* $\wedge$
mark-to-delete-clauses-l-inv S' xs0 $(i,\ T',\ xs)$ $\wedge$
correct-watching' S$\wedge$ *literals-are-*$\mathcal{L}_{in}$*' S* $\wedge$ *literals-are-*$\mathcal{L}_{in}$*' T*$\rangle$

definition *mark-to-delete-clauses-wl-pre* :: $\langle$*'v twl-st-wl* $\Rightarrow$ *bool*$\rangle$
where
$\langle$*mark-to-delete-clauses-wl-pre S* $\longleftrightarrow$
$(\exists\ T.\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *mark-to-delete-clauses-l-pre T* $\wedge$ *literals-are-*$\mathcal{L}_{in}$*' S*$)\rangle$

definition *mark-garbage-wl*:: $\langle$*nat* $\Rightarrow$ *'v twl-st-wl* $\Rightarrow$ *'v twl-st-wl*$\rangle$ **where**
$\langle$*mark-garbage-wl* $= (\lambda C\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ WS,\ Q).$
$(M,$ *fmdrop C N*$,\ D,\ NE, \{\#\},\ NEk,\ UEk,\ NS, \{\#\},\ N0, \{\#\},\ WS,\ Q))\rangle$

definition *mark-to-delete-clauses-wl* :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹*mark-to-delete-clauses-wl* = (λS. *do* {
  ASSERT(*mark-to-delete-clauses-wl-pre S*);
  xs ← *collect-valid-indices-wl S*;
  l ← *SPEC*(λ-:: *nat*. *True*);
  (-, *S*, -) ← *WHILE*$_T$*mark-to-delete-clauses-wl-inv S xs*
    (λ(*i*, *S*, *xs*). *i* < *length xs*)
    (λ(*i*, *T*, *xs*). *do* {
      if(*xs*!*i* ∉# *dom-m* (*get-clauses-wl T*)) *then RETURN* (*i*, *T*, *delete-index-and-swap xs i*)
      else do {
        ASSERT(*0* < *length* (*get-clauses-wl T*∝(*xs*!*i*)));
  ASSERT (*get-clauses-wl T* ∝ (*xs* ! *i*) ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T*));
        can-del ← *SPEC*(λb. *b* ⟶
          (*Propagated* (*get-clauses-wl T*∝(*xs*!*i*)!*0*) (*xs*!*i*) ∉ *set* (*get-trail-wl T*)) ∧
           ¬*irred* (*get-clauses-wl T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl T* ∝ (*xs*!*i*)) ≠ *2*);
        ASSERT(*i* < *length xs*);
        if can-del
        then
          RETURN (*i*, *mark-garbage-wl* (*xs*!*i*) *T*, *delete-index-and-swap xs i*)
        else
          RETURN (*i*+*1*, *T*, *xs*)
      }
    })
    (*l*, *S*, *xs*);
  remove-all-learned-subsumed-clauses-wl S
})›

lemma *mark-to-delete-clauses-wl-invD1*:
  assumes ‹*mark-to-delete-clauses-wl-inv S xs* (*i*, *T*, *ys*)› **and**
    ‹*C* ∈# *dom-m* (*get-clauses-wl T*)› **and**
    ‹*0* < *length* (*get-clauses-wl T* ∝ *C*)›
  shows
    ‹*get-clauses-wl T* ∝ *C* ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T*)›
proof −
  have ‹*literals-are-*$\mathcal{L}_{in}$' *T*›
    using *assms* **unfolding** *mark-to-delete-clauses-wl-inv-def* **by** *blast*
  then have ‹*get-clauses-wl T* ∝ *C* ! *0* ∈# *all-init-lits-of-wl T*›
    using *nth-in-all-lits-stI*[*of C T 0*]
    using *assms*(*2*,*3*) **by** (*auto simp del*: *nth-in-all-lits-stI*
      simp: *all-lits-st-init-learned literals-are-*$\mathcal{L}_{in}$'*-def*)
  then show ‹*?thesis*›
    by (*simp add*: $\mathcal{L}_{all}$*-all-init-atms*)
qed

lemma *remove-all-learned-subsumed-clauses-wl-remove-all-learned-subsumed-clauses2*:
  ‹(*remove-all-learned-subsumed-clauses-wl*, *remove-all-learned-subsumed-clauses*)
   ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching'* *S* ∧ *literals-are-*$\mathcal{L}_{in}$' *S*} →$_f$
     ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching'* *S* ∧ *literals-are-*$\mathcal{L}_{in}$' *S*}⟩*nres-rel*›
  by (*auto intro*!: *frefI nres-relI*
    simp: *remove-all-learned-subsumed-clauses-wl-def remove-all-learned-subsumed-clauses-def*
      literals-are-$\mathcal{L}_{in}$'*-def correct-watching'.simps state-wl-l-def all-init-learned-lits-simps-Q*
    clause-to-update-def all-lits-of-mm-union blits-in-$\mathcal{L}_{in}$'*-def*)
    (*meson basic-trans-rules*(*31*) *in-all-learned-lits-of-wl-addUS*)

lemma *mark-to-delete-clauses-wl-mark-to-delete-clauses-l*:

```
  ‹(mark-to-delete-clauses-wl, mark-to-delete-clauses-l)
    ∈ {(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching′ S ∧ literals-are-ℒ_in′ S} →_f
      ⟨{(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching′ S ∧ literals-are-ℒ_in′ S}⟩nres-rel›
proof −
  have [refine0]: ‹collect-valid-indices-wl S ≤ ⇓ Id (collect-valid-indices S′)›
    if ‹(S, S′) ∈ {(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching′ S ∧
          mark-to-delete-clauses-wl-pre S}›
    for S S′
    using that by (auto simp: collect-valid-indices-wl-def collect-valid-indices-def)
  have if-inv: ‹(if A then RETURN P else RETURN Q) = RETURN (if A then P else Q)› for A P Q
    by auto
  have Ball-range[simp]: ‹(∀ x∈range f ∪ range g. P x)⟷(∀ x. P (f x) ∧ P (g x))› for P f g
    by auto
  show ?thesis
    unfolding mark-to-delete-clauses-wl-def mark-to-delete-clauses-l-def
      uncurry-def
    apply (intro frefI nres-relI)
    apply (refine-vcg
      WHILEIT-refine[where
        R = ‹{((i, S, xs), (j, T, ys)). i = j ∧ (S, T) ∈ state-wl-l None ∧ correct-watching′ S ∧
          xs = ys ∧ literals-are-ℒ_in′ S}›]
      remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp[THEN
        fref-to-Down-curry]
      remove-all-learned-subsumed-clauses-wl-remove-all-learned-subsumed-clauses2[THEN
        fref-to-Down])
    subgoal unfolding mark-to-delete-clauses-wl-pre-def by blast
    subgoal by auto
    subgoal by (auto simp: state-wl-l-def)
    subgoal unfolding mark-to-delete-clauses-wl-inv-def by fast
    subgoal by auto
    subgoal by (force simp: state-wl-l-def)
    subgoal by auto
    subgoal by (force simp: state-wl-l-def)
    subgoal for x y xs xsa l to-keep xa x′ x1 x2 x1a x2a x1b x2b x1c x2c
      by (auto simp: mark-to-delete-clauses-wl-invD1)
    subgoal by (auto simp: state-wl-l-def can-delete-def)
    subgoal by auto
    subgoal by (force simp: state-wl-l-def)
    subgoal
      by (auto simp: state-wl-l-def correct-watching-fmdrop mark-garbage-wl-def
          mark-garbage-l-def ac-simps
        split: prod.splits)
    subgoal by (auto simp: state-wl-l-def)
    subgoal by (auto simp: state-wl-l-def)
    done
qed
```

This is only a specification and must be implemented. There are two ways to do so:

1. clean the watch lists and then iterate over all clauses to rebuild them.
2. iterate over the watch list and check whether the clause index is in the domain or not.

   It is not clear which is faster (but option 1 requires only 1 memory access per clause instead of two). The first option is implemented in SPASS-SAT. The latter version (partly) in cadical.

definition *rewatch-clauses* :: ‹*'v twl-st-wl ⇒ 'v twl-st-wl nres*› **where**
‹*rewatch-clauses* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
SPEC(λ(*M'*, *N'*, *D'*, *NE'*, *UE'*, *NEk'*, *UEk'*, *NS'*, *US'*, *N0'*, *U0'*, *Q'*, *W'*).
(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*) = (*M'*, *N'*, *D'*, *NE'*, *UE'*, *NEk'*, *UEk'*, *NS'*, *US'*, *N0'*, *U0'*, *Q'*) ∧
correct-watching (*M*, *N'*, *D*, *NE*, *UE*, *NEk'*, *UEk'*, *NS'*, *US'*, *N0'*, *U0'*, *Q*, *W'*)))›

definition *mark-to-delete-clauses-wl-post* **where**
‹*mark-to-delete-clauses-wl-post S T* ⟷
(∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧ *blits-in-*$\mathcal{L}_{in}$ *S* ∧
mark-to-delete-clauses-l-post S' T' ∧ *correct-watching S* ∧ *blits-in-*$\mathcal{L}_{in}$ *T* ∧
correct-watching T ∧ *get-unkept-learned-clss-wl T* = {#} ∧
get-subsumed-learned-clauses-wl T = {#} ∧ *get-learned-clauses0-wl T* = {#})›

definition *cdcl-twl-full-restart-wl-prog* :: ‹*'v twl-st-wl ⇒ 'v twl-st-wl nres*› **where**
‹*cdcl-twl-full-restart-wl-prog S* = *do* {
ASSERT(*mark-to-delete-clauses-wl-pre S*);
T ← *mark-to-delete-clauses-wl S*;
ASSERT(*mark-to-delete-clauses-wl-post S T*);
RETURN T
}›

lemma *correct-watching-correct-watching*: ‹*correct-watching S* ⟹ *correct-watching' S*›
by (*cases S*, *auto simp only*: *correct-watching.simps correct-watching'.simps all-lits-st-init-learned image-mset-union*)

lemma (**in** −) [*twl-st-l*, *simp*]:
‹(*Sa*, *x*) ∈ *twl-st-l None* ⟹ *get-all-learned-clss x* = *mset* `# (*get-learned-clss-l Sa*) + *get-unit-learned-clss-l Sa* + *get-subsumed-learned-clauses-l Sa* + *get-learned-clauses0-l Sa*›
by (*cases Sa*; *cases x*) (*auto simp*: *twl-st-l-def get-learned-clss-l-def mset-take-mset-drop-mset'*)

lemma *cdcl-twl-full-restart-wl-prog-final-rel*:
assumes
S-Sa: ‹(*S*, *Sa*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching' S* ∧ *blits-in-*$\mathcal{L}_{in}$ *S*}› **and**
pre-Sa: ‹*mark-to-delete-clauses-l-pre Sa*› **and**
pre-S: ‹*mark-to-delete-clauses-wl-pre S*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching' S* ∧ *literals-are-*$\mathcal{L}_{in}$*' S*}›
and
pre-l: ‹*mark-to-delete-clauses-l-post Sa Ta*›
shows ‹*mark-to-delete-clauses-wl-post S T*›
proof −
obtain *x* **where**
Sa-x: ‹(*Sa*, *x*) ∈ *twl-st-l None*› **and**
st: ‹*remove-one-annot-true-clause*** *Sa Ta*› **and**
list-invs: ‹*twl-list-invs Sa*› **and**
struct: ‹*twl-struct-invs x*› **and**
confl: ‹*get-conflict-l Sa* = *None*› **and**
upd: ‹*clauses-to-update-l Sa* = {#}› **and**
empty:
‹*get-unkept-learned-clss-l Ta* = {#}›
‹*get-subsumed-learned-clauses-l Ta* = {#}›
‹*get-learned-clauses0-l Ta* = {#}›
using *pre-l*
unfolding *mark-to-delete-clauses-l-post-def* **by** *blast*

have *corr-S*: ‹*correct-watching′ S*› **and** *corr-T*: ‹*correct-watching′ T*› **and**
blits-S: ‹*blits-in-$\mathcal{L}_{in}$ S*› **and**
blits-T: ‹*literals-are-$\mathcal{L}_{in}$′ T*› **and**
S-Sa: ‹(*S*, *Sa*) ∈ *state-wl-l None*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ *state-wl-l None*›
using *S-Sa T-Ta* **by** *auto*

have ‹*set-mset* (*all-init-lits-of-wl S*) = *set-mset* (*all-lits-st S*)›
using *literals-are-$\mathcal{L}_{in}$′-literals-are-$\mathcal{L}_{in}$-iff*(*4*)[*OF S-Sa Sa-x struct*] **.**

then have *corr-S′*: ‹*correct-watching S*›
using *corr-S*
by (*cases S*; *simp only*: *correct-watching′.simps correct-watching.simps*)
obtain *y* **where**
‹*cdcl-twl-restart-l*** *Sa Ta*› **and**
Ta-y: ‹(*Ta*, *y*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *x y*› **and**
struct: ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF st list-invs confl upd Sa-x struct*]
by *blast*

have *eq*: ‹*set-mset* (*all-init-lits-of-wl T*) = *set-mset* (*all-lits-st T*)›
using *literals-are-$\mathcal{L}_{in}$′-literals-are-$\mathcal{L}_{in}$-iff*(*4*)[*OF T-Ta Ta-y struct*] **.**

then have *corr-T′*: ‹*correct-watching T*›
using *corr-T*
by (*cases T*; *simp only*: *correct-watching′.simps correct-watching.simps*)

have ‹*blits-in-$\mathcal{L}_{in}$ T*›
using *blits-T eq*
unfolding *blits-in-$\mathcal{L}_{in}$-def blits-in-$\mathcal{L}_{in}$′-def all-lits-def literals-are-$\mathcal{L}_{in}$′-def all-init-lits-def*
by (*auto dest*: *multi-member-split simp*: *ac-simps*)
moreover have ‹*get-unkept-learned-clss-wl T* = {#} ∧
get-subsumed-learned-clauses-wl T = {#} ∧
get-learned-clauses0-wl T = {#}›
using *empty T-Ta* **by** *simp*
ultimately show *?thesis*
using *S-Sa T-Ta corr-T′ corr-S′ pre-l blits-S*
unfolding *mark-to-delete-clauses-wl-post-def*
by *blast*
qed

lemma *cdcl-twl-full-restart-wl-prog-final-rel′*:
assumes
S-Sa: ‹(*S*, *Sa*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S* ∧ *blits-in-$\mathcal{L}_{in}$ S*}› **and**
pre-Sa: ‹*mark-to-delete-clauses-l-pre Sa*› **and**
pre-S: ‹*mark-to-delete-clauses-wl-pre S*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching′ S* ∧ *literals-are-$\mathcal{L}_{in}$′ S*}›
and
pre-l: ‹*mark-to-delete-clauses-l-post Sa Ta*›
shows ‹*mark-to-delete-clauses-wl-post S T*›
proof −
obtain *x* **where**
Sa-x: ‹(*Sa*, *x*) ∈ *twl-st-l None*› **and**

st: ‹*remove-one-annot-true-clause*** *Sa Ta*› **and**
list-invs: ‹*twl-list-invs Sa*› **and**
struct: ‹*twl-struct-invs x*› **and**
confl: ‹*get-conflict-l Sa* = *None*› **and**
upd: ‹*clauses-to-update-l Sa* = {#}› **and**
empty:
‹*get-unkept-learned-clss-l Ta* = {#}›
‹*get-subsumed-learned-clauses-l Ta* = {#}›
‹*get-learned-clauses0-l Ta* = {#}›
using *pre-l*
unfolding *mark-to-delete-clauses-l-post-def* **by** *blast*

have *corr-S*: ‹*correct-watching S*› **and** *corr-T*: ‹*correct-watching′ T*› **and**
blits-T: ‹*literals-are-*$\mathcal{L}_{in}$*′ T*› **and**
blits-S: ‹*blits-in-*$\mathcal{L}_{in}$ *S*› **and**
S-Sa: ‹(*S*, *Sa*) ∈ *state-wl-l None*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ *state-wl-l None*›
using *S-Sa T-Ta* **by** *auto*
have *corr-S*: ‹*correct-watching′ S*›
using *correct-watching-correct-watching*[*OF corr-S*] **.**
have ‹*set-mset* (*all-init-lits-of-wl S*) = *set-mset* (*all-lits-st S*)›
using *literals-are-*$\mathcal{L}_{in}$*′-literals-are-*$\mathcal{L}_{in}$*-iff*(*4*)[*OF S-Sa Sa-x struct*] **.**
then have *corr-S′*: ‹*correct-watching S*›
using *corr-S*
by (*cases S*; *simp only*: *correct-watching′.simps correct-watching.simps*)
obtain *y* **where**
‹*cdcl-twl-restart-l*** *Sa Ta*› **and**
Ta-y: ‹(*Ta*, *y*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *x y*› **and**
struct: ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF st list-invs confl upd Sa-x struct*]
by *blast*

have *eq*: ‹*set-mset* (*all-init-lits-of-wl T*) = *set-mset* (*all-lits-st T*)›
using *literals-are-*$\mathcal{L}_{in}$*′-literals-are-*$\mathcal{L}_{in}$*-iff*(*4*)[*OF T-Ta Ta-y struct*] **.**

then have *corr-T′*: ‹*correct-watching T*›
using *corr-T*
by (*cases T*; *simp only*: *correct-watching′.simps correct-watching.simps*)

have ‹*blits-in-*$\mathcal{L}_{in}$ *T*›
using *blits-T eq*
unfolding *blits-in-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*′-def all-lits-def literals-are-*$\mathcal{L}_{in}$*′-def*
all-init-lits-def
by (*auto dest*: *multi-member-split simp*: *ac-simps*)
moreover have ‹*get-unkept-learned-clss-wl T* = {#} ∧
get-subsumed-learned-clauses-wl T = {#} ∧
get-learned-clauses0-wl T = {#}›
using *empty T-Ta* **by** *simp*
ultimately show *?thesis*
using *S-Sa T-Ta corr-T′ corr-S′ pre-l blits-S*
unfolding *mark-to-delete-clauses-wl-post-def* **apply** −
apply (*rule exI*[*of* - *Sa*])
apply (*rule exI*[*of* - *Ta*])

by *blast*
qed

lemma *mark-to-delete-clauses-l-pre-blits-in-$\mathcal{L}_{in}'$*:
assumes
S-Sa: ‹$(S,\ Sa) \in \{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S \wedge$ *blits-in-$\mathcal{L}_{in}$* $S\}$› **and**
pre-Sa: ‹*mark-to-delete-clauses-l-pre* Sa›
shows ‹*literals-are-$\mathcal{L}_{in}'$* S›
proof −
obtain x **where**
Sa-x: ‹$(Sa,\ x) \in$ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs* Sa› **and**
struct: ‹*twl-struct-invs* x›
using *pre-Sa*
unfolding *mark-to-delete-clauses-l-pre-def* **by** *blast*

have *corr-S*: ‹*correct-watching* S› **and**
blits-S: ‹*blits-in-$\mathcal{L}_{in}$* S› **and**
S-Sa: ‹$(S,\ Sa) \in$ *state-wl-l None*›
using *S-Sa* **by** *auto*
have *corr-S*: ‹*correct-watching'* S›
using *correct-watching-correct-watching*[*OF corr-S*] **.**
have *eq*: ‹*set-mset* (*all-init-lits-of-wl* S) = *set-mset* (*all-lits-st* S)›
using *literals-are-$\mathcal{L}_{in}'$-literals-are-$\mathcal{L}_{in}$-iff*(*4*)[*OF S-Sa Sa-x struct*] **.**

then have *corr-S'*: ‹*correct-watching* S›
using *corr-S*
by (*cases* S; *simp only*: *correct-watching'.simps correct-watching.simps*)

have ‹*literals-are-$\mathcal{L}_{in}'$* S›
using *blits-S eq*
unfolding *blits-in-$\mathcal{L}_{in}$-def blits-in-$\mathcal{L}_{in}'$-def all-lits-def literals-are-$\mathcal{L}_{in}'$-def*
all-init-lits-def all-lits-st-init-learned[*of* S] **by** *auto*
then show *?thesis*
using *S-Sa corr-S' blits-S*
unfolding *mark-to-delete-clauses-wl-post-def*
by *blast*
qed

lemma *cdcl-twl-full-restart-wl-prog-cdcl-full-twl-restart-l-prog*:
‹(*cdcl-twl-full-restart-wl-prog*, *cdcl-twl-full-restart-l-prog*)
$\in \{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S \wedge$ *blits-in-$\mathcal{L}_{in}$* $S\} \rightarrow_f$
$\langle\{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S \wedge$ *blits-in-$\mathcal{L}_{in}$* $S\}\rangle$*nres-rel*›
unfolding *cdcl-twl-full-restart-wl-prog-def cdcl-twl-full-restart-l-prog-def*
rewatch-clauses-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
mark-to-delete-clauses-wl-mark-to-delete-clauses-l[*THEN fref-to-Down*]
remove-one-annot-true-clause-imp-wl-remove-one-annot-true-clause-imp[*THEN fref-to-Down*])
subgoal unfolding *mark-to-delete-clauses-wl-pre-def*
by (*blast intro*: *correct-watching-correct-watching mark-to-delete-clauses-l-pre-blits-in-$\mathcal{L}_{in}'$*)
subgoal unfolding *mark-to-delete-clauses-wl-pre-def* **by** (*blast intro*: *correct-watching-correct-watching*)
subgoal
by (*rule cdcl-twl-full-restart-wl-prog-final-rel'*)
subgoal by (*auto simp*: *state-wl-l-def mark-to-delete-clauses-wl-post-def*)

done

datatype *restart-type* =
NO-RESTART |
GC |
RESTART |
INPROCESS

context *twl-restart-ops*
begin

definition (**in** *twl-restart-ops*) *restart-required-wl* :: ‹'*v twl-st-wl* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *restart-type nres*› **where**
‹*restart-required-wl S last-GC last-Restart n* = *do* {
ASSERT(*size* (*get-all-learned-clss-wl S*) ≥ *last-GC*);
ASSERT(*size* (*get-all-learned-clss-wl S*) ≥ *last-Restart*);
SPEC (λ*b*.
(*b* = *GC* ⟶ *f n* < *size* (*get-all-learned-clss-wl S*) − *last-GC*) ∧
(*b* = *INPROCESS* ⟶ *f n* < *size* (*get-all-learned-clss-wl S*) − *last-GC*) ∧
(*b* = *RESTART* ⟶ *last-Restart* < *size* (*get-all-learned-clss-wl S*)))}›

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-abs-wl-inv*
:: ‹'*v twl-st-wl* ⇒ *bool* × '*v twl-st-wl* × *nat* × *nat* × *nat* ⇒ *bool*› **where**
‹*cdcl-twl-stgy-restart-abs-wl-inv S*$_0$ = (λ(*brk*, *T*, *last-GC*, *last-Restart*, *n*).
(∃ *S*$_0$' *T*'.
(*S*$_0$, *S*$_0$') ∈ *state-wl-l None* ∧
(*T*, *T*') ∈ *state-wl-l None* ∧
cdcl-twl-stgy-restart-abs-l-inv S$_0$' (*brk*, *T*', *last-GC*, *last-Restart*, *n*) ∧
(¬*brk* ⟶ *correct-watching T*)))›
end

definition (**in** −) *cdcl-GC-clauses-pre-wl* :: ‹'*v twl-st-wl* ⇒ *bool*› **where**
‹*cdcl-GC-clauses-pre-wl S* ⟷ (
∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL S ∧
literals-are-$\mathcal{L}_{in}$' *S* ∧
cdcl-GC-clauses-pre T
)›

definition *cdcl-GC-clauses-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*cdcl-GC-clauses-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*cdcl-GC-clauses-pre-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
let b = *True*;
if b then do {
(*N*', -) ← *SPEC* (λ(*N*'', *m*). *GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N*'') ∧
0 ∉# *dom-m N*'');
Q ← *SPEC*(λ*Q*. *correct-watching*' (*M*, *N*', *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)∧
literals-are-$\mathcal{L}_{in}$' (*M*, *N*', *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
RETURN (*M*, *N*', *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *WS*, *Q*)
}
else RETURN (*M*, *N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *WS*, *Q*)})›

lemma *cdcl-GC-clauses-wl-cdcl-GC-clauses*:
‹(*cdcl-GC-clauses-wl*, *cdcl-GC-clauses*) ∈ {(*S*::'*v twl-st-wl*, *S*').

$(S,\ S') \in$ *state-wl-l None* $\wedge$ *no-lost-clause-in-WL* $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S\}$ $\rightarrow_f$ $\langle\{(S::'v$ *twl-st-wl*, $S')$.
$(S,\ S') \in$ *state-wl-l None* $\wedge$ *correct-watching'* $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S \wedge$
get-unkept-learned-clss-wl $S = \{\#\} \wedge$
get-subsumed-learned-clauses-wl $S = \{\#\} \wedge$
get-learned-clauses0-wl $S = \{\#\}\}\rangle$*nres-rel*›
unfolding *cdcl-GC-clauses-wl-def cdcl-GC-clauses-def*
apply (*intro frefI nres-relI*)
apply *refine-vcg*
subgoal unfolding *cdcl-GC-clauses-pre-wl-def* **by** *blast*
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by *auto*
subgoal for *x y x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h*
x2h x1i x2i x1j x2j x1k x2k b xa x' x1l x2l x1m x2m Q
by (*auto simp*: *state-wl-l-def blits-in-*$\mathcal{L}_{in}'$*-def literals-are-*$\mathcal{L}_{in}'$*-empty*
dest: *literals-are-*$\mathcal{L}_{in}'$*-empty*$(6-)$)
subgoal by *auto*
done

definition *mark-to-delete-clauses-GC-wl-inv*
:: ‹*'v twl-st-wl* $\Rightarrow$ *nat list* $\Rightarrow$ *nat* $\times$ *'v twl-st-wl*$\times$ *nat list* $\Rightarrow$ *bool*›
where
‹*mark-to-delete-clauses-GC-wl-inv* $= (\lambda S$ *xs0* $(i,\ T,$ *xs*).
$\exists\, S'\ T'.\ (S,\ S') \in$ *state-wl-l None* $\wedge\ (T,\ T') \in$ *state-wl-l None* $\wedge$
mark-to-delete-clauses-l-inv S' *xs0* $(i,\ T',$ *xs*$) \wedge$
no-lost-clause-in-WL $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ T)›

lemma *mark-to-delete-clauses-GC-wl-inv-alt-def*:
‹*mark-to-delete-clauses-GC-wl-inv* $= (\lambda S$ *xs0* $(i,\ T,$ *xs*).
$\exists\, S'\ T'.\ (S,\ S') \in$ *state-wl-l None* $\wedge\ (T,\ T') \in$ *state-wl-l None* $\wedge$
mark-to-delete-clauses-l-inv S' *xs0* $(i,\ T',$ *xs*$) \wedge$
set (*get-all-mark-of-propagated* (*get-trail-wl* T)) $\subseteq$ *set* (*get-all-mark-of-propagated* (*get-trail-wl* S)) $\cup\ \{0\} \wedge$
no-lost-clause-in-WL $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ T)›
unfolding *mark-to-delete-clauses-GC-wl-inv-def case-prod-beta*
apply (*intro impI iffI allI ext*; *normalize-goal+*)
subgoal for *S xs0 p x xa*
by (*rule-tac x=x* **in** *exI*, *rule-tac x=xa* **in** *exI*)
(*auto simp*: *mark-to-delete-clauses-l-inv-def*
dest!: *rtranclp-remove-one-annot-true-clause-get-all-mark-of-propagated*)
subgoal for *S xs0 p x xa*
by (*rule-tac x=x* **in** *exI*, *rule-tac x=xa* **in** *exI*) *auto*
done

definition *mark-to-delete-clauses-GC-wl-pre* :: ‹*'v twl-st-wl* $\Rightarrow$ *bool*›
where
‹*mark-to-delete-clauses-GC-wl-pre* $S \longleftrightarrow$
$(\exists\, T.\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *mark-to-delete-clauses-l-GC-pre* $T \wedge$
literals-are-$\mathcal{L}_{in}'$ S)›

lemma *mark-to-delete-clause-GC-wl-pre-alt-def*:
‹*mark-to-delete-clauses-GC-wl-pre* $S \longleftrightarrow$
$(\exists\, T.\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *mark-to-delete-clauses-l-GC-pre* $T \wedge$
literals-are-$\mathcal{L}_{in}'$ $S \wedge$ *set* (*get-all-mark-of-propagated* (*get-trail-wl* S)) $\subseteq \{0\})$›
unfolding *mark-to-delete-clauses-GC-wl-pre-def*

apply (*rule iffI impI conjI*; *normalize-goal*+)
subgoal for *T*
by (*rule exI*[*of* - *T*]) (*auto simp*: *mark-to-delete-clauses-l-GC-pre-def*)
subgoal for *T*
by (*rule exI*[*of* - *T*]) (*auto simp*: *mark-to-delete-clauses-l-GC-pre-def*)
done

Unlike the *mark-to-delete-clauses-wl-def*, this version is only used for garbage collection. Hence, there are a few differences.

definition *mark-to-delete-clauses-GC-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*mark-to-delete-clauses-GC-wl* = (λ*S*. *do* {
ASSERT(*mark-to-delete-clauses-GC-wl-pre S*);
xs ← *collect-valid-indices-wl S*;
l ← *SPEC*(λ-:: *nat*. *True*);
(-, *S*, -) ← *WHILE*$_T$*mark-to-delete-clauses-GC-wl-inv S xs*
(λ(*i*, *S*, *xs*). *i* < *length xs*)
(λ(*i*, *T*, *xs*). *do* {
if(*xs*!*i* ∉# *dom-m* (*get-clauses-wl T*)) *then RETURN* (*i*, *T*, *delete-index-and-swap xs i*)
else do {
ASSERT(*0* < *length* (*get-clauses-wl T*∝(*xs*!*i*)));
ASSERT (*get-clauses-wl T* ∝ (*xs* ! *i*) ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T*));
can-del ← *SPEC*(λ*b*. *b* ⟶
¬*irred* (*get-clauses-wl T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl T* ∝ (*xs*!*i*)) ≠ *2*);
ASSERT(*i* < *length xs*);
if can-del
then
RETURN (*i*, *mark-garbage-wl* (*xs*!*i*) *T*, *delete-index-and-swap xs i*)
else
RETURN (*i*+*1*, *T*, *xs*)
}
})
(*l*, *S*, *xs*);
remove-all-learned-subsumed-clauses-wl S
})›

lemma *mark-to-delete-clauses-GC-wl-invD1*:
assumes ‹*mark-to-delete-clauses-GC-wl-inv S xs* (*i*, *T*, *ys*)› **and**
‹*C* ∈# *dom-m* (*get-clauses-wl T*)› **and**
‹*0* < *length* (*get-clauses-wl T* ∝ *C*)›
shows
‹*get-clauses-wl T* ∝ *C* ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T*)›
proof −
have ‹*get-clauses-wl T* ∝ *C* ! *0* ∈# *all-lits-st T*›
using *assms*(*2*,*3*) **by** *auto*
moreover have ‹*literals-are-*$\mathcal{L}_{in}$' *T*›
using *assms* **unfolding** *mark-to-delete-clauses-GC-wl-inv-def* **by** *blast*
ultimately show ‹*?thesis*›
unfolding *all-lits-st-init-learned* $\mathcal{L}_{all}$*-all-init-atms literals-are-*$\mathcal{L}_{in}$*'-def*
by *auto*
qed

lemma *no-lost-clause-in-WL-drop-irrel*[*simp*]:
‹¬*irred a C* ⟹
no-lost-clause-in-WL (*x1d*, *a*, *aa*, *ab*, *ac*, *NEk*, *UEk*, *ad*, *US*, *af*, *ag*, *ah*, *b*) ⟹
no-lost-clause-in-WL (*x1d*, *fmdrop C a*, *aa*, *ab*, {#}, *NEk*, *UEk*, *ad*, {#}, *af*, {#}, *ah*, *b*)›

by (*auto simp*: *no-lost-clause-in-WL-def all-init-lits-of-wl-def*
dest: *in-diffD*)

lemma *literals-are-$\mathcal{L}_{in}$'-drop-irrel*:
assumes
irred: ‹¬ *irred N C*› **and**
‹*literals-are-$\mathcal{L}_{in}$'* (*M'*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
shows
‹*literals-are-$\mathcal{L}_{in}$'* (*M*, *fmdrop C N*, *D*, *NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *Q*, *W*)›
using *assms*
by (*cases* ‹*C* ∈# *dom-m N*›)
(*auto simp*: *literals-are-$\mathcal{L}_{in}$'-def blits-in-$\mathcal{L}_{in}$'-def
all-learned-lits-of-wl-def image-mset-remove1-mset-if all-lits-of-mm-union
all-init-lits-of-wl-def learned-clss-l-l-fmdrop ran-m-fmdrop-notin
split: *if-splits*
dest!: *all-lits-of-mm-diffD*)

lemma *remove-all-learned-subsumed-clauses-wl-remove-all-learned-subsumed-clauses3*:
‹(*remove-all-learned-subsumed-clauses-wl*, *remove-all-learned-subsumed-clauses*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧ *literals-are-$\mathcal{L}_{in}$' S*} →$_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧ *literals-are-$\mathcal{L}_{in}$' S*}⟩*nres-rel*›
by (*auto intro*!: *frefI nres-relI*
simp: *remove-all-learned-subsumed-clauses-wl-def remove-all-learned-subsumed-clauses-def*
literals-are-$\mathcal{L}_{in}$'-def correct-watching'.simps state-wl-l-def
all-init-lits-of-wl-def all-learned-lits-of-wl-def
no-lost-clause-in-WL-def
clause-to-update-def all-lits-of-mm-union blits-in-$\mathcal{L}_{in}$'-def)

lemma *mark-to-delete-clauses-wl-mark-to-delete-clauses-l2*:
‹(*mark-to-delete-clauses-GC-wl*, *mark-to-delete-clauses-l*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧ *literals-are-$\mathcal{L}_{in}$' S* ∧
set (*get-all-mark-of-propagated* (*get-trail-wl S*)) ⊆ {*0*}} →$_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧ *literals-are-$\mathcal{L}_{in}$' S*}⟩*nres-rel*›
proof −
have [*refine0*]: ‹*collect-valid-indices-wl S* ≤ ⇓ *Id* (*collect-valid-indices S'*)›
if ‹(*S*, *S'*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧
mark-to-delete-clauses-GC-wl-pre S}›
for *S S'*
using *that* **by** (*auto simp*: *collect-valid-indices-wl-def collect-valid-indices-def*)
have *if-inv*: ‹(*if A then RETURN P else RETURN Q*) = *RETURN* (*if A then P else Q*)› **for** *A P Q*
by *auto*
have *Ball-range*[*simp*]: ‹(∀ *x*∈*range f* ∪ *range g*. *P x*)⟷(∀ *x*. *P* (*f x*) ∧ *P* (*g x*))› **for** *P f g*
by *auto*
show *?thesis*
unfolding *mark-to-delete-clauses-GC-wl-def mark-to-delete-clauses-l-def*
uncurry-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
WHILEIT-refine[**where**
R = ‹{((*i*, *S*, *xs*), (*j*, *T*, *ys*)). *i* = *j* ∧ (*S*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧
xs = *ys* ∧ *literals-are-$\mathcal{L}_{in}$' S*}›]
remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp[*THEN fref-to-Down-curry*]
remove-all-learned-subsumed-clauses-wl-remove-all-learned-subsumed-clauses3[*THEN fref-to-Down*])
subgoal for *x y*
unfolding *mark-to-delete-clauses-GC-wl-pre-def mark-to-delete-clauses-l-GC-pre-def*
mark-to-delete-clauses-l-pre-def

```
    apply normalize-goal+
    apply (rule-tac x=y in exI)
    by fastforce
  subgoal by auto
  subgoal by (auto simp: state-wl-l-def)
  subgoal unfolding mark-to-delete-clauses-GC-wl-inv-def by fast
  subgoal by auto
  subgoal by (force simp: state-wl-l-def)
  subgoal by auto
  subgoal by (force simp: state-wl-l-def)
  subgoal for x y xs xsa l to-keep xa x' x1 x2 x1a x2a x1b x2b x1c x2c
    by (auto simp: mark-to-delete-clauses-GC-wl-invD1)
  subgoal by (auto simp: state-wl-l-def can-delete-def
    mark-to-delete-clauses-GC-wl-inv-alt-def
    dest!: split-list)
  subgoal by auto
  subgoal by (force simp: state-wl-l-def)
  subgoal
    by (auto simp: state-wl-l-def correct-watching-fmdrop mark-garbage-wl-def
        mark-garbage-l-def
      literals-are-\mathcal{L}_{in}'-drop-irrel
      split: prod.splits)
  subgoal by (auto simp: state-wl-l-def)
  subgoal by (auto simp: ac-simps)
  done
qed

lemma correct-watching'-nobin-clauses-pointed-to:
  assumes
    xa-xb: ‹(xa, xb) ∈ state-wl-l None› and
    corr: ‹correct-watching'-nobin xa› and
    pre: ‹cdcl-GC-clauses-pre xb› and
    L: ‹literals-are-\mathcal{L}_{in}' xa›
  shows ‹set-mset (dom-m (get-clauses-wl xa))
        ⊆ clauses-pointed-to
          (Neg ‘ set-mset (all-init-atms-st xa) ∪
           Pos ‘ set-mset (all-init-atms-st xa))
          (get-watched-wl xa)›
       (is ‹?G1 is ‹- ⊆ ?A›) and
    ‹no-lost-clause-in-WL xa› (is ‹?G2)
proof −
  let ?A = ‹all-init-atms (get-clauses-wl xa) (get-unit-init-clss-wl xa)›
  show ?G1
  proof
    fix C
    assume C: ‹C ∈# dom-m (get-clauses-wl xa)›
    obtain M N D NE UE NEk UEk NS US N0 U0 Q W where
      xa: ‹xa = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)›
      by (cases xa)
    obtain x where
      xb-x: ‹(xb, x) ∈ twl-st-l None› and
      ‹twl-list-invs xb› and
      struct-invs: ‹twl-struct-invs x› and
      ‹get-conflict-l xb = None› and
      ‹clauses-to-update-l xb = {#}› and
      ‹count-decided (get-trail-l xb) = 0› and
```

```
    ‹∀ L∈set (get-trail-l xb). mark-of L = 0›
     using pre unfolding cdcl-GC-clauses-pre-def by fast
   have ‹twl-st-inv x›
     using xb-x C struct-invs
     by (auto simp: twl-struct-invs-def
       cdcl_W-restart-mset.cdcl_W-all-struct-inv-def)
   then have le0: ‹get-clauses-wl xa ∝ C ≠ []›
     using xb-x C xa-xb
     by (cases x; cases ‹irred N C›)
       (auto simp: twl-struct-invs-def twl-st-inv.simps
         twl-st-l-def state-wl-l-def xa ran-m-def conj-disj-distribR
         Collect-disj-eq Collect-conv-if
       dest!: multi-member-split)
   then have le: ‹N ∝ C ! 0 ∈ set (watched-l (N ∝ C))›
     by (cases ‹N ∝ C›) (auto simp: xa)
   have eq: ‹set-mset (\mathcal{L}_{all} (all-init-atms N NE)) =
       set-mset (all-lits-of-mm (mset `# init-clss-lf N + NE))›
     by (auto simp del: all-init-atms-def[symmetric]
       simp: all-init-atms-def xa \mathcal{L}_{all}-atm-of-all-lits-of-mm[symmetric]
         all-init-lits-def)

   have H: ‹get-clauses-wl xa ∝ C ! 0 ∈# all-init-lits-of-wl xa›
     using L C le0 apply −
     unfolding all-init-atms-def[symmetric] all-init-lits-def[symmetric]
     apply (subst literals-are-\mathcal{L}_{in}'-literals-are-\mathcal{L}_{in}-iff(4)[OF xa-xb xb-x struct-invs])
     apply (cases ‹N ∝ C›; auto simp: literals-are-\mathcal{L}_{in}-def all-lits-def ran-m-def eq
         all-lits-of-mm-add-mset is-\mathcal{L}_{all}-def xa all-lits-of-m-add-mset
         \mathcal{L}_{all}-all-atms-all-lits all-lits-st-def
       dest!: multi-member-split)
     done
   moreover {
     have ‹{#i ∈# fst `# mset (W (N ∝ C ! 0)). i ∈# dom-m N#} =
           add-mset C {#Ca ∈# remove1-mset C (dom-m N). N ∝ C ! 0 ∈ set (watched-l (N ∝
Ca))#}›
       using corr H C le unfolding xa
       by (auto simp: clauses-pointed-to-def correct-watching'-nobin.simps xa
         simp flip: all-init-atms-def all-init-lits-def all-init-atms-alt-def
           all-init-lits-alt-def
         simp: clause-to-update-def
         simp del: all-init-atms-def[symmetric]
         dest!: multi-member-split)
     from arg-cong[OF this, of set-mset] have ‹C ∈ fst ` set (W (N ∝ C ! 0))›
       using corr H C le unfolding xa
       by (auto simp: clauses-pointed-to-def correct-watching'.simps xa
         simp: all-init-atms-def all-init-lits-def clause-to-update-def
         simp del: all-init-atms-def[symmetric]
         dest!: multi-member-split) }
   ultimately show ‹C ∈ ?A›
     by (cases ‹N ∝ C ! 0›)
       (auto simp: clauses-pointed-to-def correct-watching'.simps xa
         simp flip: all-init-lits-def all-init-atms-alt-def
           all-init-lits-alt-def
         simp: clause-to-update-def all-init-atms-st-def all-init-lits-of-wl-def all-init-atms-def
         simp del: all-init-atms-def[symmetric]
       dest!: multi-member-split)
 qed
```

moreover have ‹*set-mset* (*all-init-lits-of-wl xa*) =
Neg ‘ *set-mset* (*all-init-atms-st xa*) ∪ *Pos* ‘ *set-mset* (*all-init-atms-st xa*)›
unfolding *all-init-lits-of-wl-def*
all-lits-of-mm-def all-init-atms-st-def all-init-atms-def
by (*auto simp*: *all-init-atms-def all-init-lits-def all-lits-of-mm-def image-image*
image-Un
simp del: *all-init-atms-def*[*symmetric*])
moreover have ‹*distinct-watched* (*watched-by xa* (*Pos L*))›
‹*distinct-watched* (*watched-by xa* (*Neg L*))›
if ‹*L* ∈# *all-init-atms-st xa*› **for** *L*
using *that corr*
by (*cases xa*;
auto simp: *correct-watching'-nobin.simps all-init-lits-of-wl-def all-init-atms-def*
all-lits-of-mm-union all-init-lits-def all-init-atms-st-def literal.atm-of-def
in-all-lits-of-mm-uminus-iff[*symmetric*, *of* ‹*Pos* -›]
simp del: *all-init-atms-def*[*symmetric*]
split: *literal.splits*; *fail*)+
ultimately show *?G2*
unfolding *no-lost-clause-in-WL-def*
by (*auto simp del*: *all-init-atms-def*[*symmetric*])
qed

definition *cdcl-GC-clauses-prog-copy-wl-entry*
:: ‹'*v clauses-l* ⇒ '*v watched* ⇒ '*v literal* ⇒
'*v clauses-l* ⇒ ('*v clauses-l* × '*v clauses-l*) *nres*›
where
‹*cdcl-GC-clauses-prog-copy-wl-entry* = (λ*N W A N'*. *do* {
let le = *length W*;
(*i*, *N*, *N'*) ← *WHILE*$_T$
(λ(*i*, *N*, *N'*). *i* < *le*)
(λ(*i*, *N*, *N'*). *do* {
ASSERT(*i* < *length W*);
let C = *fst* (*W* ! *i*);
if C ∈# *dom-m N then do* {
D ← *SPEC*(λ*D*. *D* ∉# *dom-m N'* ∧ *D* ≠ *0*);
RETURN (*i*+*1*, *fmdrop C N*, *fmupd D* (*N* ∝ *C*, *irred N C*) *N'*)
} *else RETURN* (*i*+*1*, *N*, *N'*)
}) (*0*, *N*, *N'*);
RETURN (*N*, *N'*)
})›

lemma *cdcl-GC-clauses-prog-copy-wl-entry*:
fixes *A* :: ‹'*v literal*› **and** *WS* :: ‹'*v literal* ⇒ '*v watched*›
defines [*simp*]: ‹*W* ≡ *WS A*›
assumes ‹*ran m0* ⊆ *set-mset* (*dom-m N0'*) ∧
(∀ *L*∈*dom m0*. *L* ∉# (*dom-m N0*)) ∧
set-mset (*dom-m N0*) ⊆ *clauses-pointed-to* (*set-mset* $\mathcal{A}$) *WS* ∧
0 ∉# *dom-m N0'*›
shows
‹*cdcl-GC-clauses-prog-copy-wl-entry N0 W A N0'* ≤
SPEC(λ(*N*, *N'*). (∃ *m*. *GC-remap*** (*N0*, *m0*, *N0'*) (*N*, *m*, *N'*) ∧
ran m ⊆ *set-mset* (*dom-m N'*) ∧
(∀ *L*∈*dom m*. *L* ∉# (*dom-m N*)) ∧
set-mset (*dom-m N*) ⊆ *clauses-pointed-to* (*set-mset* (*remove1-mset A* $\mathcal{A}$)) *WS*) ∧
(∀ *L* ∈ *set W*. *fst L* ∉# *dom-m N*) ∧
0 ∉# *dom-m N'*)›

proof −
have [*simp*]:
‹*x* ∈# *remove1-mset a* (*dom-m aaa*) ⟷ *x* ≠ *a* ∧ *x* ∈# *dom-m aaa*› **for** *x a aaa*
using *distinct-mset-dom*[*of aaa*]
by (*cases* ‹*a* ∈# *dom-m aaa*›)
(*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset*)

show *?thesis*
unfolding *cdcl-GC-clauses-prog-copy-wl-entry-def*
apply (*refine-vcg*
WHILET-rule[**where** *I* = ‹λ(*i*, *N*, *N'*). ∃ *m*. *GC-remap*** (*N0*, *m0*, *N0'*) (*N*, *m*, *N'*) ∧
ran m ⊆ *set-mset* (*dom-m N'*) ∧
(∀ *L*∈*dom m*. *L* ∉# (*dom-m N*)) ∧
set-mset (*dom-m N*) ⊆ *clauses-pointed-to* (*set-mset* (*remove1-mset A* $\mathcal{A}$)) *WS* ∪
(*fst*) ‘ *set* (*drop i W*) ∧
(∀ *L* ∈ *set* (*take i W*). *fst L* ∉# *dom-m N*) ∧
0 ∉# *dom-m N'*› **and**
R = ‹*measure* (λ(*i*, *N*, *N'*). *length W* − *i*)›])
subgoal by *auto*
subgoal
using *assms*
by (*cases* ‹*A* ∈# $\mathcal{A}$›) (*auto dest*!: *multi-member-split*)
subgoal by *auto*
subgoal for *s aa ba aaa baa x x1 x2 x1a x2a*
apply *clarify*
apply (*subgoal-tac* ‹(∃ *m'*. *GC-remap* (*aaa*, *m*, *baa*) (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*, *m'*,
fmupd x (*the* (*fmlookup aaa* (*fst* (*W* ! *aa*)))) *baa*) ∧
ran m' ⊆ *set-mset* (*dom-m* (*fmupd x* (*the* (*fmlookup aaa* (*fst* (*W* ! *aa*)))) *baa*)) ∧
(∀ *L*∈*dom m'*. *L* ∉# (*dom-m* (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*)))) ∧
set-mset (*dom-m* (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*)) ⊆
clauses-pointed-to (*set-mset* (*remove1-mset A* $\mathcal{A}$)) *WS* ∪
fst ‘ *set* (*drop* (*Suc aa*) *W*) ∧
(∀ *L* ∈ *set* (*take* (*Suc aa*) *W*). *fst L* ∉# *dom-m* (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*))›)
apply (*auto intro*: *rtranclp.rtrancl-into-rtrancl*)[]
apply (*auto simp*: *GC-remap.simps Cons-nth-drop-Suc*[*symmetric*]
take-Suc-conv-app-nth
dest: *multi-member-split*)
apply (*rule-tac x*= ‹*m*(*fst* (*W* ! *aa*) ↦ *x*)› **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*= ‹*fst* (*W* ! *aa*)› **in** *exI*)
apply (*force dest*: *rtranclp-GC-remap-ran-m-no-lost*)
apply *auto*
by (*smt basic-trans-rules*(*31*) *fun-upd-apply mem-Collect-eq option.simps*(*1*) *ran-def*)
subgoal by *auto*
subgoal by (*auto 5 5 simp*: *GC-remap.simps Cons-nth-drop-Suc*[*symmetric*]
take-Suc-conv-app-nth
dest: *multi-member-split*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *cdcl-GC-clauses-prog-single-wl*

:: ‹'v clauses-l ⇒ ('v literal ⇒ 'v watched) ⇒ 'v ⇒
'v clauses-l ⇒ ('v clauses-l × 'v clauses-l × ('v literal ⇒ 'v watched)) nres›
where
‹cdcl-GC-clauses-prog-single-wl = (λN WS A N'. do {
L ← RES {Pos A, Neg A};
(N, N') ← cdcl-GC-clauses-prog-copy-wl-entry N (WS L) L N';
let WS = WS(L := []);
(N, N') ← cdcl-GC-clauses-prog-copy-wl-entry N (WS (−L)) (−L) N';
let WS = WS(−L := []);
RETURN (N, N', WS)
})›

lemma *cdcl-GC-clauses-prog-single-wl-removed*:
‹∀ L∈set (W (Pos A)). fst L ∉# dom-m aaa ⟹
∀ L∈set (W (Neg A)). fst L ∉# dom-m a ⟹
GC-remap** (aaa, ma, baa) (a, mb, b) ⟹
set-mset (dom-m a) ⊆ clauses-pointed-to (set-mset (negs A + poss A − {#Neg A, Pos A#})) W
⟹
xa ∈# dom-m a ⟹
xa ∈ clauses-pointed-to (Neg ‘ set-mset (remove1-mset A A) ∪ Pos ‘ set-mset (remove1-mset A
A))
(W(Pos A := [], Neg A := []))›
‹∀ L∈set (W (Neg A)). fst L ∉# dom-m aaa ⟹
∀ L∈set (W (Pos A)). fst L ∉# dom-m a ⟹
GC-remap** (aaa, ma, baa) (a, mb, b) ⟹
set-mset (dom-m a) ⊆ clauses-pointed-to (set-mset (negs A + poss A − {#Pos A, Neg A#})) W
⟹
xa ∈# dom-m a ⟹
xa ∈ clauses-pointed-to
(Neg ‘ set-mset (remove1-mset A A) ∪ Pos ‘ set-mset (remove1-mset A A))
(W(Neg A := [], Pos A := []))›
supply *poss-remove-Pos*[*simp*] *negs-remove-Neg*[*simp*]
by (*case-tac* [!] ‹A ∈# A›)
(*fastforce simp*: *clauses-pointed-to-def*
dest!: *multi-member-split*
dest: *rtranclp-GC-remap-ran-m-no-lost*)+

lemma *cdcl-GC-clauses-prog-single-wl*:
fixes A :: ‹'v› **and** WS :: ‹'v literal ⇒ 'v watched› **and**
N0 :: ‹'v clauses-l›
assumes ‹ran m ⊆ set-mset (dom-m N0') ∧
(∀ L∈dom m. L ∉# (dom-m N0)) ∧
set-mset (dom-m N0) ⊆
clauses-pointed-to (set-mset (negs A + poss A)) W ∧
0 ∉# dom-m N0'›
shows
‹cdcl-GC-clauses-prog-single-wl N0 W A N0' ≤
SPEC(λ(N, N', WS'). ∃ m'. GC-remap** (N0, m, N0') (N, m', N') ∧
ran m' ⊆ set-mset (dom-m N') ∧
(∀ L∈dom m'. L ∉# dom-m N) ∧
WS' (Pos A) = [] ∧ WS' (Neg A) = [] ∧
(∀ L. L ≠ Pos A ⟶ L ≠ Neg A ⟶ W L = WS' L) ∧
set-mset (dom-m N) ⊆
clauses-pointed-to
(set-mset (negs (remove1-mset A A) + poss (remove1-mset A A))) WS' ∧
0 ∉# dom-m N'

```
  )›
proof −
  have [simp]: ‹A ∉# 𝒜 ⟹ negs 𝒜 + poss 𝒜 − {#Neg A, Pos A#} =
   negs 𝒜 + poss 𝒜›
    by (induction 𝒜) auto
  have [simp]: ‹A ∉# 𝒜 ⟹ negs 𝒜 + poss 𝒜 − {#Pos A, Neg A#} =
   negs 𝒜 + poss 𝒜›
    by (induction 𝒜)  auto
  show ?thesis
    unfolding cdcl-GC-clauses-prog-single-wl-def
    apply (refine-vcg)
    subgoal for x
      apply (rule order-trans, rule cdcl-GC-clauses-prog-copy-wl-entry[of - - -
          ‹negs 𝒜 + poss 𝒜›])
       apply(solves ‹use assms in auto›)
      apply (rule RES-rule)
      apply (refine-vcg)
      apply clarify
      subgoal for aa ba aaa baa ma
        apply (rule order-trans,
            rule cdcl-GC-clauses-prog-copy-wl-entry[of ma - -
              ‹remove1-mset x (negs 𝒜 + poss 𝒜)›])
         apply (solves ‹auto simp: clauses-pointed-to-remove1-if›)[]
        unfolding Let-def
        apply (rule RES-rule)
        apply clarsimp
        apply (simp add: eq-commute[of ‹Neg -›]
            uminus-lit-swap clauses-pointed-to-remove1-if)
        apply auto
         apply (rule-tac x=mb in exI)
         apply (auto dest!:
            simp: clauses-pointed-to-remove1-if
            clauses-pointed-to-remove1-if2
            clauses-pointed-to-remove1-if2-eq)
         apply (subst (asm) clauses-pointed-to-remove1-if2-eq)
          apply (force dest: rtranclp-GC-remap-ran-m-no-lost)
         apply (auto intro!: cdcl-GC-clauses-prog-single-wl-removed)[]
         apply (rule-tac x=mb in exI)
         apply (auto dest: multi-member-split[of A]
            simp: clauses-pointed-to-remove1-if
            clauses-pointed-to-remove1-if2
            clauses-pointed-to-remove1-if2-eq)
         apply (subst (asm) clauses-pointed-to-remove1-if2-eq)
          apply (force dest: rtranclp-GC-remap-ran-m-no-lost)
        apply (auto intro!: cdcl-GC-clauses-prog-single-wl-removed)[]
        done
    done
    done
qed


definition (in −)cdcl-GC-clauses-prog-wl-inv
  :: ‹'v multiset ⇒ 'v clauses-l ⇒
    'v multiset × ('v clauses-l × 'v clauses-l × ('v literal ⇒ 'v watched)) ⇒ bool›
where
‹cdcl-GC-clauses-prog-wl-inv 𝒜 N0 = (λ(ℬ, (N, N', WS)). ℬ ⊆# 𝒜 ∧
```

$(\forall A \in \textit{set-mset } \mathcal{A} - \textit{set-mset } \mathcal{B}.\ (WS\ (Pos\ A) = []) \wedge WS\ (Neg\ A) = []) \wedge$
$0 \notin\# \textit{ dom-m } N' \wedge$
$(\exists m.\ \textit{GC-remap}^{**}\ (N0, (\lambda\text{-}.\ None), \textit{fmempty})\ (N, m, N') \wedge$
$\textit{ran } m \subseteq \textit{set-mset } (\textit{dom-m } N') \wedge$
$(\forall L \in \textit{dom } m.\ L \notin\# \textit{ dom-m } N) \wedge$
$\textit{set-mset } (\textit{dom-m } N) \subseteq \textit{clauses-pointed-to } (Neg\ `\ \textit{set-mset } \mathcal{B} \cup Pos\ `\ \textit{set-mset } \mathcal{B})\ WS))\rangle$

definition *cdcl-GC-clauses-prog-wl* :: $\langle 'v\ \textit{twl-st-wl} \Rightarrow 'v\ \textit{twl-st-wl nres}\rangle$ **where**
$\langle \textit{cdcl-GC-clauses-prog-wl} = (\lambda(M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS).\ \textit{do}\ \{$
$\textit{ASSERT}(\textit{cdcl-GC-clauses-pre-wl } (M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS));$
$\mathcal{A} \leftarrow \textit{SPEC}(\lambda\mathcal{A}.\ \textit{set-mset } \mathcal{A} = \textit{set-mset } (\textit{all-init-atms-st } (M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS)));$
$(\text{-}, (N, N', WS)) \leftarrow \textit{WHILE}_T{}^{\textit{cdcl-GC-clauses-prog-wl-inv } \mathcal{A}\ N_0}$
$(\lambda(\mathcal{B}, \text{-}).\ \mathcal{B} \neq \{\#\})$
$(\lambda(\mathcal{B}, (N, N', WS)).\ \textit{do}\ \{$
$\textit{ASSERT}(\mathcal{B} \neq \{\#\});$
$A \leftarrow \textit{SPEC } (\lambda A.\ A \in\# \mathcal{B});$
$(N, N', WS) \leftarrow \textit{cdcl-GC-clauses-prog-single-wl } N\ WS\ A\ N';$
$\textit{RETURN } (\textit{remove1-mset } A\ \mathcal{B}, (N, N', WS))$
$\})$
$(\mathcal{A}, (N_0, \textit{fmempty}, WS));$
$\textit{RETURN } (M, N', D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS)$
$\})\rangle$

lemma *no-lost-clause-in-WL-no-lost-clause-in-WL0D*:
$\langle \textit{no-lost-clause-in-WL } S \Longrightarrow \textit{no-lost-clause-in-WL0 } S\rangle$
by (*auto simp*: *no-lost-clause-in-WL-def no-lost-clause-in-WL0-def*)

lemma *no-lost-clause-in-WL-alt-def*:
$\langle \textit{no-lost-clause-in-WL0 } (M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS) \longleftrightarrow$
$\textit{set-mset } (\textit{dom-m } N_0) \subseteq \textit{clauses-pointed-to}$
$(Neg\ `\ \textit{set-mset } (\textit{all-init-atms } N_0\ (NE{+}NEk{+}NS{+}N0)) \cup Pos\ `\ \textit{set-mset } (\textit{all-init-atms } N_0\ (NE{+}NEk{+}NS{+}N0)))$
$WS\rangle$
proof −
have [*simp*]: $\langle \textit{set-mset } (\textit{all-init-lits-of-wl } ([], N_0, D, NE, \{\#\}, NEk, \{\#\}, NS, \{\#\}, N0, \{\#\}, Q, WS)) =$
$(Neg\ `\ \textit{set-mset } (\textit{all-init-atms } N_0\ (NE + NEk + NS + N0)) \cup Pos\ `\ \textit{set-mset } (\textit{all-init-atms } N_0\ (NE + NEk + NS + N0)))\rangle$
unfolding *all-init-lits-of-wl-def all-init-atms-def image-image*
all-lits-of-mm-def all-init-lits-def set-image-mset image-mset-union
sum-mset.union image-Un
by (*auto simp add*: *image-image image-Un*)
show *?thesis*
unfolding *no-lost-clause-in-WL0-def*
by *auto*
qed

lemma *cdcl-GC-clauses-prog-wl*:
assumes $\langle((M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS), S) \in \textit{state-wl-l None} \wedge$
no-lost-clause-in-WL $(M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS) \wedge$
cdcl-GC-clauses-pre $S \wedge$
literals-are-$\mathcal{L}_{in}{}'$ $(M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS)\rangle$
shows
$\langle \textit{cdcl-GC-clauses-prog-wl } (M, N_0, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, WS) \leq$
$(\textit{SPEC}(\lambda(M', N', D', NE', UE', NEk', UEk', NS', US', N0', U0', Q', WS').$
$(M', D', NE', UE', NEk', UEk', NS', US', N0', U0', Q') = (M, D, NE, UE, NEk, UEk, NS,$

```
US, N0, U0, Q) ∧
        (∃ m. GC-remap** (N0, (λ-. None), fmempty) (fmempty, m, N')) ∧
        0 ∉# dom-m N' ∧ (∀ L ∈# all-init-lits N0 (NE+NEk+NS+N0). WS' L = [])))›
proof −
  show ?thesis
    supply[[goals-limit=1]]
    unfolding cdcl-GC-clauses-prog-wl-def
    apply (refine-vcg
      WHILEIT-rule[where R = ‹measure (λ(A::'v multiset, (-::'v clauses-l, - ::'v clauses-l,
          -:: 'v literal ⇒ 'v watched)). size A)›])
    subgoal
      using assms
      unfolding cdcl-GC-clauses-pre-wl-def
      by blast
    subgoal by auto
    subgoal using assms
      no-lost-clause-in-WL-no-lost-clause-in-WL0D[of ‹(M, N0, D, NE, UE, NEk, UEk, NS, US, N0,
U0, Q, WS)›, unfolded no-lost-clause-in-WL-alt-def]
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      by (auto simp: all-init-atms-st-def)
    subgoal by auto
    subgoal for a b aa ba ab bb ac bc ad bd ae be af bf ag bg ah bh ai bi aj bj ak bk x s al bl am
        bm an bn xa
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      apply clarify
      apply (rule order-trans,
        rule-tac m=m and A=al in cdcl-GC-clauses-prog-single-wl)
      subgoal by (auto simp: all-init-atms-st-def)
      subgoal
        apply (rule RES-rule)
        apply clarify
        apply (rule RETURN-rule)
        apply clarify
        apply (intro conjI)
            apply (solves auto)
           apply (solves ‹auto dest!: multi-member-split›)
          apply (solves auto)
         apply (rule-tac x=m' in exI)
         apply (solves ‹auto›)[]
        apply (simp-all add: size-Diff1-less)[]
        done
      done
    subgoal
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      by auto
    subgoal
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      by auto
    subgoal
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      by auto
    subgoal
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      by (intro ballI, rename-tac L, case-tac L)
       (auto simp: in-all-lits-of-mm-ain-atms-of-iff all-init-atms-def all-init-atms-st-def
          simp del: all-init-atms-def[symmetric]
```

dest!: *multi-member-split*)
done
qed

lemma *cdcl-GC-clauses-prog-wl2*:
assumes ‹((*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*), *S*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*) ∧
cdcl-GC-clauses-pre S ∧
literals-are-$\mathcal{L}_{in}$′ (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*)›
‹$N_0 = N_0'$›
shows
‹*cdcl-GC-clauses-prog-wl* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*) ≤ ⇓ *Id*
(*SPEC*(λ(*M*′, *N*′, *D*′, *NE*′, *UE*′, *NEk*′, *UEk*′, *NS*′, *US*′, *N0*′, *U0*′, *Q*′, *WS*′).
(*M*′, *D*′, *NE*′, *UE*′, *NEk*′, *UEk*′, *NS*′, *US*′, *N0*′, *U0*′, *Q*′) = (*M*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*) ∧
(∃ *m*. *GC-remap*** (N_0', (λ-. *None*), *fmempty*) (*fmempty*, *m*, *N*′)) ∧
0 ∉# *dom-m N*′ ∧ (∀ *L* ∈# *all-init-lits* N_0 (*NE*+*NEk*+*NS*+*N0*). *WS*′ *L* = [])))›
proof −
show *?thesis*
unfolding ‹$N_0 = N_0'$›[*symmetric*]
apply (*rule order-trans*[*OF cdcl-GC-clauses-prog-wl*[*OF assms*(*1*)]])
apply (*rule RES-refine*)
apply (*fastforce dest*: *rtranclp-GC-remap-all-init-lits*)
done
qed

end
theory *Watched-Literals-Watch-List-Inprocessing*
imports *Watched-Literals-Watch-List Watched-Literals-List-Inprocessing*
Watched-Literals-Watch-List-Restart
begin

definition *simplify-clause-with-unit-st-wl-pre* **where**
‹*simplify-clause-with-unit-st-wl-pre C S* ⟷ (∃ *T*.
(*S*, *T*) ∈ *state-wl-l None* ∧
simplify-clause-with-unit-st-pre C T)›

definition *simplify-clause-with-unit-st-wl* :: ‹*nat* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*simplify-clause-with-unit-st-wl* = (λ*C* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(*simplify-clause-with-unit-st-wl-pre C* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
ASSERT (*C* ∈# *dom-m* N_0 ∧ *count-decided M* = *0* ∧ *D* = *None* ∧ *no-dup M* ∧ *C* ≠ *0*);
let S = (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*);
if False
then RETURN S
else do {
let E = *mset* (N_0 ∝ *C*);
let irr = *irred* N_0 *C*;
(*unc*, *b*, *N*) ← *simplify-clause-with-unit C M* N_0;
if unc then do {
ASSERT (*N* = N_0);
RETURN S
}
else if b then do {

let $T = (M,$ *fmdrop* $C\ N,\ D,$ (*if* irr *then add-mset* E *else* id) $NE,$ (*if* $\neg irr$ *then add-mset* E *else* id) $UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$;
ASSERT(*set-mset* (*all-learned-lits-of-wl* T) = *set-mset* (*all-learned-lits-of-wl* S));
ASSERT(*set-mset* (*all-init-lits-of-wl* T) = *set-mset* (*all-init-lits-of-wl* S));
RETURN T
}
else if size $(N \propto C) = 1$
then do {
let $L = ((N \propto C)\ !\ 0)$;
let $T = ($*Propagated* $L\ 0\ \#\ M,$ *fmdrop* $C\ N,\ D,\ NE,\ UE,$ (*if* irr *then add-mset* $\{\#L\#\}$ *else* id) $NEk,$ (*if* $\neg irr$ *then add-mset* $\{\#L\#\}$ *else* id)$UEk,$ (*if* irr *then add-mset* E *else* id) $NS,$ (*if* $\neg irr$ *then add-mset* E *else* id)$US,\ N0,\ U0,$ *add-mset* $(-L)\ Q,\ W)$;
ASSERT(*set-mset* (*all-learned-lits-of-wl* T) = *set-mset* (*all-learned-lits-of-wl* S));
ASSERT(*set-mset* (*all-init-lits-of-wl* T) = *set-mset* (*all-init-lits-of-wl* S));
ASSERT (*undefined-lit* $M\ L \wedge L \in\#$ *all-lits-st* S);
RETURN T
}
else if size $(N \propto C) = 0$
then do {
let $T =$ $(M,$ *fmdrop* $C\ N,$ *Some* $\{\#\},\ NE,\ UE,\ NEk,\ UEk,$ (*if* irr *then add-mset* E *else* id) $NS,$ (*if* $\neg irr$ *then add-mset* E *else* id) $US,$ (*if* irr *then add-mset* $\{\#\}$ *else* id) $N0,$ (*if* $\neg irr$ *then add-mset* $\{\#\}$ *else* id)$U0,\ \{\#\},\ W)$;
ASSERT(*set-mset* (*all-learned-lits-of-wl* T) = *set-mset* (*all-learned-lits-of-wl* S));
ASSERT(*set-mset* (*all-init-lits-of-wl* T) = *set-mset* (*all-init-lits-of-wl* S));
RETURN T
}
else do {
let $T =$ $(M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,$ (*if* irr *then add-mset* E *else* id) $NS,$ (*if* $\neg irr$ *then add-mset* E *else* id) $US,\ N0,\ U0,\ Q,\ W)$;
ASSERT(*set-mset* (*all-learned-lits-of-wl* T) = *set-mset* (*all-learned-lits-of-wl* S));
ASSERT(*set-mset* (*all-init-lits-of-wl* T) = *set-mset* (*all-init-lits-of-wl* S));
RETURN T
}
}
})›

definition *simplify-clauses-with-unit-st-wl-pre* **where**
‹*simplify-clauses-with-unit-st-wl-pre* $S \longleftrightarrow (\exists\ T.$
$(S,\ T) \in$ *state-wl-l None* $\wedge$ *no-lost-clause-in-WL* $S \wedge$
simplify-clauses-with-unit-st-pre $T)$›

definition *simplify-clauses-with-unit-st-wl-inv* **where**
‹*simplify-clauses-with-unit-st-wl-inv* S_0 *it* $S \longleftrightarrow (\exists\ T_0\ T.$
$(S_0,\ T_0) \in$ *state-wl-l None* $\wedge$
$(S,\ T) \in$ *state-wl-l None* $\wedge$
simplify-clauses-with-unit-st-inv T_0 *it* $T \wedge$
no-lost-clause-in-WL $S)$›

definition *simplify-clauses-with-unit-st-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*simplify-clauses-with-unit-st-wl* $S =$
do {
ASSERT(*simplify-clauses-with-unit-st-wl-pre* S);
$xs \leftarrow$ *SPEC*($\lambda xs.$ *finite* xs);
$T \leftarrow$ *FOREACHci*($\lambda it\ T.$ *simplify-clauses-with-unit-st-wl-inv* S *it* T)
(xs)
($\lambda S.$ *get-conflict-wl* S = *None*)

```
    (λi S. if i ∈# dom-m (get-clauses-wl S) then simplify-clause-with-unit-st-wl i S else RETURN S)
     S;
   ASSERT(set-mset (all-learned-lits-of-wl T) ⊆ set-mset (all-learned-lits-of-wl S));
   ASSERT(set-mset (all-init-lits-of-wl T) = set-mset (all-init-lits-of-wl S));
   RETURN T
  }›
lemma clauses-pointed-to-union:
  ‹clauses-pointed-to (A ∪ B) W = clauses-pointed-to A W ∪ clauses-pointed-to B W›
  by (auto simp: clauses-pointed-to-def)

lemma clauses-pointed-to-mono: ‹A ⊆ B ⟹ clauses-pointed-to A W ⊆ clauses-pointed-to B W›
  by (auto simp: clauses-pointed-to-def)

lemma simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st:
  assumes ST: ‹(S, T) ∈ state-wl-l None› and ‹(i,j) ∈ nat-rel› and
    point: ‹no-lost-clause-in-WL S›
  shows
  ‹simplify-clause-with-unit-st-wl i S ≤ ⇓ {(S',T). (S',T) ∈ state-wl-l None ∧
    no-lost-clause-in-WL S' ∧
    get-watched-wl S' = get-watched-wl S}
  (simplify-clause-with-unit-st j T)›
proof −
  have Id: ‹A = B ⟹ A ≤ ⇓Id B› for A B
    by auto
  have ij: ‹i = j›
    using assms by auto
  have [simp]:
    ‹irred b j ⟹ j ∈# dom-m b ⟹ add-mset (mset (b ∝ j))
    ({#mset (fst x). x ∈# remove1-mset (the (fmlookup b j)) (init-clss-l b)#} + d + f + h) =
    ({#mset (fst x). x ∈# (init-clss-l b)#} + d + f + h)› for C D d b f h j
    by (auto simp: image-mset-remove1-mset-if ran-m-def
      dest!: multi-member-split)
  have KK[simp]: ‹irred b j ⟹ j ∈# dom-m b ⟹  C ⊆# mset (b ∝ j) ⟹
    set-mset (all-lits-of-mm (add-mset (mset (b ∝ j))
    (add-mset C
    (mset `# remove1-mset (b ∝ j) (init-clss-lf b) + d + f + h)))) =
    set-mset (all-lits-of-mm (mset `# (init-clss-lf b) + d + f + h))›
    for b j C d f h
    using all-lits-of-m-mono[of C ‹mset (b ∝ j)›]
    by (auto simp: image-mset-remove1-mset-if ran-m-def conj-disj-distribR Collect-disj-eq
      image-Un Collect-conv-if all-lits-of-mm-add-mset
      simp flip: insert-compr
      dest!: multi-member-split[of j])

  have H: ‹fmdrop j x2 = fmdrop j b ⟹
    mset (x2 ∝ j) ⊆# mset (b ∝ j) ⟹
    irred x2 j ⟹
    irred b j ⟹
    j ∈# dom-m b ⟹
    j ∈# dom-m x2 ⟹
    set-mset (all-lits-of-mm (add-mset (mset (b ∝ j)) ({#mset (fst x). x ∈# init-clss-l x2#} +
    d + f + h))) =
    set-mset (all-lits-of-mm ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h))›
    for j x2 b d f h
    using distinct-mset-dom[of x2] distinct-mset-dom[of b]
    apply (subgoal-tac ‹{#mset (fst x). x ∈# filter-mset snd {#the (fmlookup b x). x ∈# remove1-mset
```

```
j (dom-m b)#}#} =
    {#mset (fst x). x ∈# filter-mset snd {#the (fmlookup x2 x). x ∈# remove1-mset j (dom-m x2)#}
#}›)

  apply (auto simp: ran-m-def all-lits-of-mm-add-mset
    dest!: multi-member-split[of - ‹dom-m -›]
    dest: all-lits-of-m-mono
    intro!: image-mset-cong2 filter-mset-cong2)
  apply (auto 5 3 simp: ran-m-def all-lits-of-mm-add-mset
    dest!: multi-member-split[of - ‹dom-m -›]
    dest: all-lits-of-m-mono
    intro!: image-mset-cong2 filter-mset-cong2)
  apply (metis fmdrop-eq-update-eq fmupd-lookup union-single-eq-member)
  by (metis add-mset-remove-trivial dom-m-fmdrop)
 have [simp]: ‹mset a ⊆# mset b ⟹ length a= 1 ⟹ a ! 0 ∈ set b› for a b
  by (cases a, auto)
 have K1: ‹∀ L∈#all-lits-of-mm ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h).
  distinct-watched (k L) ⟹
  irred b j ⟹
  j ∈# dom-m b ⟹
  L ∈# all-lits-of-m (mset (b ∝ j)) ⟹ distinct-watched (k L)› for b d f h k j L
  by (auto simp: ran-m-def all-lits-of-mm-add-mset dest!: multi-member-split)
 have K2: ‹∀ L∈#all-lits-of-mm ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h).
  distinct-watched (k L) ⟹
  irred b j ⟹
  j ∈# dom-m b ⟹
  mset C ⊆# mset (b ∝ j) ⟹
  length C = Suc 0 ⟹
  L ∈# all-lits-of-m ({#C!0#}) ⟹ distinct-watched (k L)› for b d f h k j L C
  using all-lits-of-m-mono[of ‹mset C› ‹mset (b ∝ j)›]
   all-lits-of-m-mono[of ‹{#C!0#}› ‹mset C›]
  by (auto simp: ran-m-def all-lits-of-mm-add-mset dest!: multi-member-split[of - ‹dom-m -›])
 have K3: ‹∀ L∈#all-lits-of-mm ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h).
  distinct-watched (k L) ⟹
   L ∈# all-lits-of-mm ({#mset (fst x). x ∈# remove1-mset (the (fmlookup b j)) (init-clss-l b)#} +
d + f + h) ⟹
  distinct-watched (k L)› for b d f h k j L C
  by (cases ‹j ∈# dom-m b›; cases ‹irred b j›)
   (auto  dest!: multi-member-split[of - ‹dom-m -›] simp: ran-m-def
      all-lits-of-mm-union all-lits-of-mm-add-mset image-mset-remove1-mset-if
     split: if-splits)
 have K4: ‹
  irred b j ⟹ j ∈# dom-m b ⟹
  all-lits-of-mm
  (add-mset (mset (b ∝ j))
  ({#mset (fst x). x ∈# init-clss-l (fmdrop j b)#} + d + f + h)) =
  all-lits-of-mm
  ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h)›
  ‹¬irred b j ⟹ j ∈# dom-m b ⟹
  all-lits-of-mm
  (add-mset (mset (b ∝ j))
  ({#mset (fst x). x ∈# learned-clss-l (fmdrop j b)#} + d + f + h)) =
  all-lits-of-mm
  ({#mset (fst x). x ∈# learned-clss-l b#} + d + f + h)›
  for d f h j b
  using distinct-mset-dom[of b]
```

apply (*auto simp add*: *init-clss-l-fmdrop learned-clss-l-fmdrop-if*)
by (*smt* (*z3*) *fmupd-same image-mset-add-mset learned-clss-l-mapsto-upd prod.collapse union-mset-add-mset-left*)

show *?thesis*
supply [[*goals-limit=1*]]
using *ST point*
apply (*cases S*; *hypsubst*)
apply (*cases T*; *hypsubst*)
unfolding *simplify-clause-with-unit-st-wl-def simplify-clause-with-unit-st-def ij state-wl-l-def prod.simps Let-def*[*of* ‹(-,-)›]
apply *refine-rcg*
subgoal for *a b c d e f g h i ja k l m aa ba ca da ea fa ga ha ia jaa ka la ma*
using *ST*
unfolding *simplify-clause-with-unit-st-wl-pre-def*
by (*rule-tac x* = ‹(*aa*, *ba*, *ca*, *da*, *ea*, *fa*, *ga*, *ha*, *ia*, *jaa*, *ka*, *la*, *ma*)› **in** *exI*)
simp
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
apply (*rule Id*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal
by (*auto simp add*: *all-learned-lits-of-wl-def all-init-lits-of-l-def all-learned-lits-of-l-def get-init-clss-l-def*)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def all-learned-lits-of-l-def get-init-clss-l-def*)
subgoal by (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop init-clss-l-fmdrop-irrelev literals-are-*$\mathcal{L}_{in}$*'-def blits-in-*$\mathcal{L}_{in}$*'-def no-lost-clause-in-WL-def dest*: *in-diffD*)
subgoal by *auto*
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def all-learned-lits-of-l-def get-init-clss-l-def*)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def all-learned-lits-of-l-def get-init-clss-l-def all-init-lits-of-wl-def*)
subgoal by *auto*
subgoal by (*auto simp*: *all-lits-st-alt-def all-learned-lits-of-wl-def all-init-lits-of-l-def all-init-lits-of-wl-def get-init-clss-l-def*)
subgoal apply (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop init-clss-l-fmdrop-irrelev add-mset-commute no-lost-clause-in-WL-def dest*: *in-diffD intro*:)
done
subgoal by *auto*
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def all-learned-lits-of-l-def get-init-clss-l-def*)

subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def all-init-lits-of-wl-def)
subgoal by (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop*
init-clss-l-fmdrop-irrelev all-lits-of-mm-add-mset
no-lost-clause-in-WL-def
dest: *in-diffD*)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def all-init-lits-of-wl-def)
subgoal for *a b c d e f g h i ja k aa ba ca da ea fa ga ha ia jaa ka x x' x1 x2 x1a x2a*
apply (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop*
init-clss-l-fmdrop-irrelev H
no-lost-clause-in-WL-def
dest: *in-diffD*
intro:)
apply (*metis* (*no-types*, *lifting*) *basic-trans-rules*(*31*) *dom-m-fmdrop insert-DiffM*)
apply (*metis* (*no-types*, *lifting*) *basic-trans-rules*(*31*) *dom-m-fmdrop init-clss-l-fmdrop-irrelev insert-DiffM*)
done
done
qed

lemma [*twl-st-wl*, *simp*]:
assumes ‹(*σ*, *σ'*) ∈ *state-wl-l None*›
shows
all-learned-lits-of-l-all-learned-lits-of-wl:
‹*all-learned-lits-of-l σ' = all-learned-lits-of-wl σ*› **and**
all-init-lits-of-l-all-init-lits-of-wl:
‹*all-init-lits-of-l σ' = all-init-lits-of-wl σ*›
using *assms* **by** (*auto simp*: *state-wl-l-def all-learned-lits-of-l-def*
all-learned-lits-of-wl-def all-init-lits-of-l-def
all-init-lits-of-wl-def)

lemma *simplify-clauses-with-unit-st-wl-simplify-clause-with-unit-st*:
assumes *ST*: ‹(*S*, *T*) ∈ *state-wl-l None*› **and**
point: ‹*correct-watching'-nobin S*› **and**
lits: ‹*literals-are-$\mathcal{L}_{in}$' S*›
shows
‹*simplify-clauses-with-unit-st-wl S* ≤ ⇓ {(*S'*,*T*). (*S'*,*T*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL S' ∧
literals-are-$\mathcal{L}_{in}$' S' ∧ *get-watched-wl S'* = *get-watched-wl S*}
(*simplify-clauses-with-unit-st T*)›
proof −
have [*refine0*]: ‹*inj-on id xs*› **for** *xs*
by *auto*
have *1*: ‹*simplify-clauses-with-unit-st-wl S* = *do* {
T ← *simplify-clauses-with-unit-st-wl S*;
RETURN T}›
by *auto*
have *2*: ‹*simplify-clauses-with-unit-st T* = *do* {
T ← *simplify-clauses-with-unit-st T*;
RETURN T}›
by *auto*
have *ST2*: ‹(*S*,*T*) ∈ {(*S'*,*U*). (*S'*,*U*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL S' ∧

get-watched-wl S = get-watched-wl S'}›
if ‹*simplify-clauses-with-unit-st-pre T*›
using *assms that correct-watching'-nobin-clauses-pointed-to0*[*OF ST*]
unfolding *simplify-clauses-with-unit-st-inv-def*
simplify-clauses-with-unit-st-pre-def
by *auto*
show *?thesis*
apply (*subst 1*)
unfolding *simplify-clauses-with-unit-st-wl-def simplify-clauses-with-unit-st-def*
nres-monad3
apply (*refine-rcg simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st*)
subgoal
using *assms ST2* **unfolding** *simplify-clauses-with-unit-st-wl-pre-def*
by *blast*
subgoal
using *ST* **by** *auto*
apply (*rule ST2*, *assumption*)
subgoal by *auto*
subgoal for *xs xsa it σ it' σ'*
using *assms* **apply** −
unfolding *simplify-clauses-with-unit-st-wl-inv-def*
apply (*rule exI*[*of - T*])
apply (*rule exI*[*of - σ'*])
by *auto*
subgoal by *auto*
apply (*rule-tac T1=σ'* **and** *j1 = x'* **in**
simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st[*THEN order-trans*])
subgoal
by *auto*
subgoal
by *auto*
subgoal
by *auto*
subgoal
by (*rule conc-fun-R-mono*)
(*use assms*(*3*) **in** ‹*auto simp*: *literals-are-$\mathcal{L}_{in}$'-def*
blits-in-$\mathcal{L}_{in}$'-def›)
subgoal
using *ST* **by** *auto*
subgoal
using *ST lits*
by (*auto 4 3 simp*: *literals-are-$\mathcal{L}_{in}$'-def watched-by-alt-def*
blits-in-$\mathcal{L}_{in}$'-def)
subgoal
using *ST lits*
by (*auto 4 3 simp*: *literals-are-$\mathcal{L}_{in}$'-def watched-by-alt-def*
blits-in-$\mathcal{L}_{in}$'-def)
done
qed

lemma *simplify-clauses-with-unit-st-wl-simplify-clause-with-unit-st2*:
assumes *ST*: ‹(*S*, *T*) ∈ *state-wl-l None*› **and**
point: ‹*no-lost-clause-in-WL S*› **and**
lits: ‹*literals-are-$\mathcal{L}_{in}$' S*›

```
  shows
    ‹simplify-clauses-with-unit-st-wl S ≤ ⇓ {(S',T). (S',T) ∈ state-wl-l None ∧
    no-lost-clause-in-WL S' ∧
    literals-are-ℒ_in' S' ∧ get-watched-wl S' = get-watched-wl S}
    (simplify-clauses-with-unit-st T)›
proof −
  have [refine0]: ‹inj-on id xs› for xs
    by auto
  have 1: ‹simplify-clauses-with-unit-st-wl S = do {
    T ← simplify-clauses-with-unit-st-wl S;
    RETURN T}›
    by auto
  have 2: ‹simplify-clauses-with-unit-st T = do {
    T ← simplify-clauses-with-unit-st T;
    RETURN T}›
    by auto
  have ST2: ‹(S,T) ∈  {(S',U). (S',U) ∈ state-wl-l None ∧
    no-lost-clause-in-WL S' ∧
    get-watched-wl S = get-watched-wl S'}›
    if ‹simplify-clauses-with-unit-st-pre T›
    using assms that correct-watching'-nobin-clauses-pointed-to0[OF ST]
    unfolding simplify-clauses-with-unit-st-inv-def
      simplify-clauses-with-unit-st-pre-def
    by auto
  show ?thesis
    apply (subst 1)
    unfolding simplify-clauses-with-unit-st-wl-def simplify-clauses-with-unit-st-def
      nres-monad3
    apply (refine-rcg simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st)
    subgoal
      using assms ST2 unfolding simplify-clauses-with-unit-st-wl-pre-def
      by blast
    subgoal
      using ST by auto
        apply (rule ST2, assumption)
    subgoal by auto
    subgoal for xs xsa it σ it' σ'
      using assms apply −
      unfolding simplify-clauses-with-unit-st-wl-inv-def
      apply (rule exI[of - T])
      apply (rule exI[of - σ'])
      by auto
    subgoal by auto
    apply (rule-tac T1=σ' and j1 = x' in
        simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st[THEN order-trans])
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by (rule conc-fun-R-mono)
        (use assms(3) in ‹auto simp: literals-are-ℒ_in'-def
        blits-in-ℒ_in'-def›)
    subgoal
```

using *ST* **by** *auto*
subgoal
using *ST lits*
by (*auto 4 3 simp*: *literals-are-$\mathcal{L}_{in}$'-def watched-by-alt-def*
blits-in-$\mathcal{L}_{in}$'-def)
subgoal
using *ST lits*
by (*auto 4 3 simp*: *literals-are-$\mathcal{L}_{in}$'-def watched-by-alt-def*
blits-in-$\mathcal{L}_{in}$'-def)
done
qed

definition *simplify-clauses-with-units-st-wl-pre* **where**
‹*simplify-clauses-with-units-st-wl-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *literals-are-$\mathcal{L}_{in}$' S*)›

definition *simplify-clauses-with-units-st-wl* **where**
‹*simplify-clauses-with-units-st-wl S* = *do* {
ASSERT(*simplify-clauses-with-units-st-wl-pre S*);
new-units ← *SPEC* (λ*b*. *b* ⟶ *get-conflict-wl S* = *None*);
if new-units
then simplify-clauses-with-unit-st-wl S
else RETURN S}›

lemma *simplify-clauses-with-units-st-wl-simplify-clause-with-units-st*:
assumes *ST*: ‹(*S*, *T*) ∈ *state-wl-l None*› **and**
point: ‹*correct-watching'-nobin S*› **and**
lits: ‹*literals-are-$\mathcal{L}_{in}$' S*›
shows
‹*simplify-clauses-with-units-st-wl S* ≤ ⇓ {(*S'*,*T*). (*S'*,*T*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL S' ∧
literals-are-$\mathcal{L}_{in}$' S' ∧ *get-watched-wl S'* = *get-watched-wl S*}
(*simplify-clauses-with-units-st T*)›
unfolding *simplify-clauses-with-units-st-wl-def simplify-clauses-with-units-st-def*
apply (*refine-vcg simplify-clauses-with-unit-st-wl-simplify-clause-with-unit-st*)
subgoal using *assms* **unfolding** *simplify-clauses-with-units-st-wl-pre-def* **by** *fast*
subgoal using *ST* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **unfolding** *simplify-clauses-with-units-st-pre-def*
by (*fast intro*!: *correct-watching'-nobin-clauses-pointed-to0*(*2*))
done

lemma *simplify-clauses-with-units-st-wl-simplify-clause-with-units-st2*:
assumes *ST*: ‹(*S*, *T*) ∈ *state-wl-l None*› **and**
point: ‹*no-lost-clause-in-WL S*› **and**
lits: ‹*literals-are-$\mathcal{L}_{in}$' S*›
shows
‹*simplify-clauses-with-units-st-wl S* ≤ ⇓ {(*S'*,*T*). (*S'*,*T*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL S' ∧
literals-are-$\mathcal{L}_{in}$' S' ∧ *get-watched-wl S'* = *get-watched-wl S*}
(*simplify-clauses-with-units-st T*)›
unfolding *simplify-clauses-with-units-st-wl-def simplify-clauses-with-units-st-def*

apply (*refine-vcg simplify-clauses-with-unit-st-wl-simplify-clause-with-unit-st2*)
subgoal using *assms* **unfolding** *simplify-clauses-with-units-st-wl-pre-def* **by** *fast*
subgoal using *ST* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **unfolding** *simplify-clauses-with-units-st-pre-def*
by (*fast intro*!: *correct-watching'-nobin-clauses-pointed-to0*(*2*))
done

6.5.9 Forward subsumption

We follow the implementation of forward that is in Splatz and not the more advanced one in CaDiCaL that relies on occurrence lists. Both version are similar (so changing it is not a problem), but IsaSAT needs a way to say that the state is not watching. This in turns means that GC needs to go over the clause domain instead of the watch lists, but makes it possible to reuse the watch lists for other things, like forward subsumption (that again only differs by the lists we use to check subsumption).

Compared to Splatz the literal-move-to-front trick is not included (at least not currently).

There is critical but major subtility: The algorithm does not work on binary clauses: Binary clauses can yield new units, which in turn, can shorten clauses later, forcing a rehash of the clauses. There are two solutions to this problem:

- avoid it completely by making sure that there are no binary clauses, requiring to duplicate the code (even if only few invariants change)
- give up on the invariant
- implement forward subsumption directly.

Long story short: we gave up and implemented the forward approach directly.

We do the simplifications in two rounds:

- once with binary clauses only (this was part of the sc2020 version)
- once with all other clauses (this is onging work)

Binary clauses

This version does not enforce that binary clauses have not been deleted.

fun *correct-watching'-leaking-bin* :: ‹*'v twl-st-wl* ⇒ *bool*› **where**
‹*correct-watching'-leaking-bin* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-init-lits-of-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
distinct-watched (*W L*) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*).
i ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* '# *mset* (*W L*)) =
clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}))›

declare *correct-watching'-leaking-bin.simps*[*simp del*]

definition *clause-remove-duplicate-clause-wl-pre* :: ‹-› **where**
‹*clause-remove-duplicate-clause-wl-pre C S* ⟷ (∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧
clause-remove-duplicate-clause-pre C S' ∧ *correct-watching'-leaking-bin S*)›

definition *clause-remove-duplicate-clause-wl* :: ‹*nat* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*clause-remove-duplicate-clause-wl C* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*).
do {
ASSERT (*clause-remove-duplicate-clause-wl-pre C* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*,
WS, *Q*));
RETURN (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *NEk*, *UEk*, (*if irred N C then add-mset* (*mset* (*N* ∝ *C*))
else id) *NS*, (*if irred N C then id else add-mset* (*mset* (*N* ∝ *C*))) *US*, *N0*, *U0*, *WS*, *Q*)
})›

lemma *filter-image-mset-removeAll*:
‹{#*C* ∈# *A*. *P C*#} − {#*C* ∈# *replicate-mset* (*count A C'*) *C'*. *P C*#} =
{#*C* ∈# *A*. *P C* ∧ *C* ≠ *C'*#}›
by (*metis count-filter-mset filter-filter-mset filter-mset-neq image-filter-replicate-mset replicate-mset-0*)

lemma *filter-image-mset-swap*: ‹*distinct-mset A* ⟹ *distinct-mset B* ⟹
{#*i* ∈# *A*. *i* ∈# *B* ∧ *P i*#} = {#*i* ∈# *B*. *i* ∈# *A* ∧ *P i*#}›
by (*smt* (*z3*) *Collect-cong distinct-mset-filter distinct-set-mset-eq set-mset-filter*)

lemma *correctly-marked-as-binary-fmdrop*:
‹*i* ∈# *dom-m x1m* ⟹ *i* ≠ *C'* ⟹ *correctly-marked-as-binary x1m* (*i*, *K*, *b*)⟹ *correctly-marked-as-binary*
(*fmdrop C' x1m*) (*i*, *K*, *b*) ›
by (*auto simp*: *correctly-marked-as-binary.simps*)

lemma *correct-watching'-leaking-bin-remove-subsumedI*:
‹*correct-watching'-leaking-bin* (*x1l*, *x1m*, *x1n*, *x1o*, *x1p*, *x1q*, *x1r*, *x1s*, *x1t*, *x1u*, *x1v*, *x1w*, *x2w*) ⟹
C' ∈# *dom-m x1m* ⟹
irred x1m C' ⟹
correct-watching'-leaking-bin
(*x1l*, *fmdrop C' x1m*, *x1n*, *x1o*, *x1p*, *x1q*, *x1r*, *add-mset* (*mset* (*x1m* ∝ *C'*)) *x1s*, *x1t*, *x1u*,
x1v, *x1w*, *x2w*)›
‹*correct-watching'-leaking-bin* (*x1l*, *x1m*, *x1n*, *x1o*, *x1p*, *x1q*, *x1r*, *x1s*, *x1t*, *x1u*, *x1v*, *x1w*, *x2w*) ⟹
C' ∈# *dom-m x1m* ⟹
¬ *irred x1m C'* ⟹
correct-watching'-leaking-bin
(*x1l*, *fmdrop C' x1m*, *x1n*, *x1o*, *x1p*, *x1q*, *x1r*, *x1s*, *add-mset* (*mset* (*x1m* ∝ *C'*)) *x1t*, *x1u*,
x1v, *x1w*, *x2w*)›

‹*correct-watching'-leaking-bin* (*x1l*, *x1m*, *x1n*, *x1o*, *x1p*, *x1q*, *x1r*, *x1s*, *x1t*, *x1u*, *x1v*, *x1w*, *x2w*) ⟹
C' ∈# *dom-m x1m* ⟹
irred x1m C' ⟹
correct-watching'-leaking-bin
(*x1l*, *fmdrop C' x1m*, *x1n*, *add-mset* (*mset* (*x1m* ∝ *C'*)) *x1o*, *x1p*, *x1q*, *x1r*, *x1s*, *x1t*, *x1u*,
x1v, *x1w*, *x2w*)›
‹*correct-watching'-leaking-bin* (*x1l*, *x1m*, *x1n*, *x1o*, *x1p*, *x1q*, *x1r*, *x1s*, *x1t*, *x1u*, *x1v*, *x1w*, *x2w*) ⟹
C' ∈# *dom-m x1m* ⟹
¬ *irred x1m C'* ⟹
correct-watching'-leaking-bin
(*x1l*, *fmdrop C' x1m*, *x1n*, *x1o*, *add-mset* (*mset* (*x1m* ∝ *C'*)) *x1p*, *x1q*, *x1r*, *x1s*, *x1t*, *x1u*,
x1v, *x1w*, *x2w*)›

apply (*auto simp*: *correct-watching'-leaking-bin.simps all-init-lits-of-wl-def*
image-mset-remove1-mset-if distinct-mset-remove1-All distinct-mset-dom correctly-marked-as-binary.simps
clause-to-update-def filter-image-mset-removeAll)
apply (*drule bspec*)
apply *assumption*
apply *normalize-goal+*
apply (*drule arg-cong*[**where** *f*=‹*filter-mset* (λ*C*. *C* ≠ *C'*)›])
apply (*auto simp add*: *filter-filter-mset intro*: *filter-mset-cong*)
apply (*drule bspec*)
apply *assumption*
apply *normalize-goal+*
apply (*drule arg-cong*[**where** *f*=‹*filter-mset* (λ*C*. *C* ≠ *C'*)›])
apply (*auto simp add*: *filter-filter-mset intro*: *filter-mset-cong*)
apply (*drule bspec*)
apply *assumption*
apply *normalize-goal+*
apply (*drule arg-cong*[**where** *f*=‹*filter-mset* (λ*C*. *C* ≠ *C'*)›])
apply (*auto simp add*: *filter-filter-mset intro*: *filter-mset-cong*)
apply (*drule bspec*)
apply *assumption*
apply *normalize-goal+*
apply (*drule arg-cong*[**where** *f*=‹*filter-mset* (λ*C*. *C* ≠ *C'*)›])
apply (*auto simp add*: *filter-filter-mset intro*: *filter-mset-cong*)
done

lemma *clause-remove-duplicate-clause-wl-clause-remove-duplicate-clause*:
assumes ‹(*C*,*C'*)∈*nat-rel*› ‹(*S*,*T*)∈ *state-wl-l None*› ‹*correct-watching'-leaking-bin S*›
shows ‹*clause-remove-duplicate-clause-wl C S* ≤
⇓{(*U*, *V*). (*U*,*V*)∈ *state-wl-l None* ∧ *correct-watching'-leaking-bin U* ∧ *get-watched-wl U* = *get-watched-wl S*} (*clause-remove-duplicate-clause C' T*)›
using *assms* **unfolding** *clause-remove-duplicate-clause-wl-def*
clause-remove-duplicate-clause-def
apply (*refine-vcg*)
subgoal unfolding *clause-remove-duplicate-clause-wl-pre-def*
by *fast*
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h x1i x2i x1j x2j*
x1k x2k x1l x2l x1m x2m x1n x2n x1o x2o x1p x2p x1q x2q x1r x2r x1s x2s x1t x2t x1u
x2u x1v x2v x1w x2w
by (*auto simp*: *state-wl-l-def clause-remove-duplicate-clause-pre-def*
intro!: *correct-watching'-leaking-bin-remove-subsumedI*)
done

definition *binary-clause-subres-lits-wl-pre* :: ‹-› **where**
‹*binary-clause-subres-lits-wl-pre C L L' S* ⟷ (∃*S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧
binary-clause-subres-lits-pre C L L' S')›

definition *binary-clause-subres-wl* :: ‹*nat* ⇒ *'v literal* ⇒ *'v literal* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*›
where
‹*binary-clause-subres-wl C L L'* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT (*binary-clause-subres-lits-wl-pre C L L'* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
RETURN (*Propagated L 0* # *M*, *fmdrop C N*, *D*, *NE*, *UE*,
(*if irred N C then add-mset* {#*L*#} *else id*) *NEk*, (*if irred N C then id else add-mset* {#*L*#}) *UEk*,
(*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *NS*, (*if irred N C then id else add-mset* (*mset* (*N* ∝ *C*))) *US*,

```
      N0, U0, add-mset (−L) Q, W)
 })›

lemma all-init-lits-of-wl-add-drop-irred:
  assumes ‹C' ∈# dom-m (x1m)› ‹irred x1m C'›
  shows ‹all-init-lits-of-wl
            ([], fmdrop C' x1m, x1n, x1o, {#}, x1q, {#},
             add-mset (mset (x1m ∝ C')) x1s, {#}, x1u, {#}, x1w, x2w) =
  all-init-lits-of-wl
            ([], x1m, x1n, x1o, {#}, x1q, {#},
    x1s, {#}, x1u, {#}, x1w, x2w)› (is ?A) and
    ‹K' ∈ set (x1m ∝ C') ⟹ set-mset (all-init-lits-of-wl
     (x1l, x1m, None, x1o, x1p, add-mset {#K'#} x1q, x1r, x1s, x1t, x1u, x1v,
      add-mset (− K') x1w, x2w)) = set-mset (all-init-lits-of-wl
     (x1l, x1m, None, x1o, x1p, x1q, x1r, x1s, x1t, x1u, x1v, x1w, x2w))› (is ‹- ⟹ ?B›)
proof −
  have A: ‹init-clss-l x1m = add-mset (x1m ∝ C', irred x1m C') (init-clss-l (fmdrop C' x1m))›
    by (smt (z3) assms(1) assms(2) eq-fst-iff fmdrop-eq-update-eq2 init-clss-l-fmdrop-if
      init-clss-l-mapsto-upd le-boolD le-boolI' sndE)
  show ?A
    using assms
    by (auto simp: all-init-lits-of-wl-def all-lits-of-mm-add-mset init-clss-l-fmdrop-if
        image-mset-remove1-mset-if
      dest!: multi-member-split)[]
  show ?B if ‹K' ∈ set (x1m ∝ C')›
    using assms that apply −
    apply (auto simp: all-init-lits-of-wl-def all-lits-of-mm-add-mset all-lits-of-m-add-mset
      in-all-lits-of-mm-uminus-iff)
   apply (subst A)
   apply (auto simp add: all-lits-of-mm-add-mset all-lits-of-m-add-mset add-mset-eq-add-mset
     in-clause-in-all-lits-of-m in-set-mset-eq-in)
   apply (subst A)
   apply (auto simp add: all-lits-of-mm-add-mset all-lits-of-m-add-mset add-mset-eq-add-mset
     in-clause-in-all-lits-of-m in-set-mset-eq-in)
   done
qed

lemma correct-watching'-leaking-bin-fmdropI:
  assumes ‹C' ∈# dom-m (x1m)› ‹irred x1m C'›
  shows
     ‹correct-watching'-leaking-bin
    (x1l, x1m, x1n, x1o, x1p,
    x1q, x1r, x1s, x1t, x1u,
    x1v, x1w, x2w) ⟹ correct-watching'-leaking-bin
     (Propagated K' 0 # x1l, fmdrop C' x1m, x1n, x1o, x1p,
    x1q, x1r, add-mset (mset (x1m ∝ C')) x1s, x1t, x1u,
    x1v, x1w, x2w)›
  using assms distinct-mset-dom[of x1m]
  unfolding correct-watching'-leaking-bin.simps
  apply (auto simp: all-init-lits-of-wl-add-drop-irred distinct-mset-remove1-All clause-to-update-def
      filter-image-mset-removeAll correctly-marked-as-binary.simps
    dest: multi-member-split[of C'])
  apply (drule bspec)
  apply assumption
  apply normalize-goal+
  apply (drule arg-cong[where f=‹filter-mset (λC. C ≠ C')›])
```

apply (*auto simp add*: *filter-filter-mset intro*: *filter-mset-cong*)
done

lemma *correct-watching'-leaking-bin-fmdropI-red*:
assumes ‹$C' \in\# \mathit{dom\text{-}m}\ (x1m)$› ‹$\neg \mathit{irred}\ x1m\ C'$›
shows
‹*correct-watching'-leaking-bin*
($x1l$, $x1m$, $x1n$, $x1o$, $x1p$,
$x1q$, $x1r$, $x1s$, $x1t$, $x1u$,
$x1v$, $x1w$, $x2w$) $\Longrightarrow$ *correct-watching'-leaking-bin*
(*Propagated* $K'\ 0\ \#\ x1l$, *fmdrop* $C'\ x1m$, $x1n$, $x1o$, $x1p$,
$x1q$, $x1r$, $x1s$, *add-mset* (*mset* ($x1m \propto C'$)) $x1t$, $x1u$,
$x1v$, $x1w$, $x2w$)›
‹*correct-watching'-leaking-bin*
($x1l$, $x1m$, $x1n$, $x1o$, $x1p$,
$x1q$, $x1r$, $x1s$, $x1t$, $x1u$,
$x1v$, $x1w$, $x2w$) $\Longrightarrow$ *correct-watching'-leaking-bin*
($x1l$, $x1m$, *None*, $x1o$, $x1p$, $x1q$, *add-mset* $\{\#K'\#\}$ $x1r$, $x1s$, $x1t$, $x1u$, $x1v$,
add-mset ($-\ K'$) $x1w$, $x2w$)›
subgoal
using *assms distinct-mset-dom*[*of* $x1m$]
unfolding *correct-watching'-leaking-bin.simps*
apply (*auto simp*: *all-init-lits-of-wl-add-drop-irred distinct-mset-remove1-All clause-to-update-def*
filter-image-mset-removeAll correctly-marked-as-binary.simps
dest: *multi-member-split*[*of* C'])[]
apply (*drule bspec*)
apply *assumption*
apply *normalize-goal+*
apply (*drule arg-cong*[**where** f=‹*filter-mset* ($\lambda C.\ C \neq C'$)›])
apply (*auto simp add*: *filter-filter-mset intro*: *filter-mset-cong*)
done
subgoal
using *assms distinct-mset-dom*[*of* $x1m$]
unfolding *correct-watching'-leaking-bin.simps*
by (*auto simp*: *all-init-lits-of-wl-add-drop-irred distinct-mset-remove1-All clause-to-update-def*
filter-image-mset-removeAll all-init-lits-of-wl-def correctly-marked-as-binary.simps
dest: *multi-member-split*[*of* C'])[]
done

lemma *correct-watching'-leaking-bin-add-unitI*:
assumes ‹$K' \in\# \mathit{mset}\ (x1m \propto C')$› ‹$C' \in\# \mathit{dom\text{-}m}\ x1m$› ‹$\mathit{irred}\ x1m\ C'$›
shows ‹*correct-watching'-leaking-bin*
($x1l$, $x1m$, *None*, $x1o$, $x1p$, $x1q$, $x1r$, $x1s$, $x1t$, $x1u$, $x1v$, $x1w$, $x2w$) $\Longrightarrow$
correct-watching'-leaking-bin
($x1l$, $x1m$, *None*, $x1o$, $x1p$, *add-mset* $\{\#K'\#\}$ $x1q$, $x1r$, $x1s$, $x1t$, $x1u$, $x1v$,
add-mset ($-\ K'$) $x1w$, $x2w$)›
using *assms*
by (*auto simp*: *correct-watching'-leaking-bin.simps clause-to-update-def*
all-init-lits-of-wl-add-drop-irred)

lemma *binary-clause-subres-wl-binary-clause-subres*:
assumes ‹$(C,C') \in \mathit{nat\text{-}rel}$› ‹$(K,K') \in Id$› ‹$(L,L') \in Id$› ‹$(S,S') \in$ *state-wl-l None*›
‹*correct-watching'-leaking-bin* S›
shows ‹*binary-clause-subres-wl* $C\ K\ L\ S \leq$
$\Downarrow\{(U,\ V).\ (U,V) \in \mathit{state\text{-}wl\text{-}l}\ None \wedge \mathit{correct\text{-}watching'\text{-}leaking\text{-}bin}\ U \wedge \mathit{get\text{-}watched\text{-}wl}\ U = \mathit{get\text{-}watched\text{-}wl}\ S\}$ (*binary-clause-subres* $C'\ K'\ L'\ S'$)›

using *assms* **unfolding** *binary-clause-subres-wl-def binary-clause-subres-def*
apply (*refine-vcg*)
subgoal unfolding *binary-clause-subres-lits-wl-pre-def*
by *fast*
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h*
x1i x2i x1j x2j x1k x2k x1l x2l x1m x2m x1n x2n x1o x2o x1p x2p x1q
x2q x1r x2r x1s x2s x1t x2t x1u x2u x1v x2v x1w x2w
apply (*auto simp*: *state-wl-l-def*
intro!: *correct-watching'-leaking-bin-fmdropI correct-watching'-leaking-bin-add-unitI*
correct-watching'-leaking-bin-fmdropI-red)
apply (*auto simp*: *binary-clause-subres-lits-pre-def*)[*2*]
apply (*auto simp*: *state-wl-l-def*
intro!: *correct-watching'-leaking-bin-fmdropI correct-watching'-leaking-bin-add-unitI*
correct-watching'-leaking-bin-fmdropI-red)
apply (*auto simp*: *binary-clause-subres-lits-pre-def*)[*2*]
apply (*auto simp*: *state-wl-l-def*
intro!: *correct-watching'-leaking-bin-fmdropI correct-watching'-leaking-bin-add-unitI*
correct-watching'-leaking-bin-fmdropI-red)
done
done

definition *deduplicate-binary-clauses-pre-wl* :: ‹*'v literal* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*deduplicate-binary-clauses-pre-wl L S* ⟷ (∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧
deduplicate-binary-clauses-pre L T ∧ *correct-watching'-leaking-bin S* ∧
literals-are-$\mathcal{L}_{in}$*' S*)›

definition *deduplicate-binary-clauses-inv-wl* :: ‹*'v twl-st-wl* ⇒ *'v literal* ⇒ *bool* × *nat* × -× *'v twl-st-wl*
⇒ *bool*› **where**
‹*deduplicate-binary-clauses-inv-wl S L* = (λ(*abort*, *i*, *mark*, *T*).
(∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧
deduplicate-binary-clauses-inv L (*fst* ‘# *mset* (*watched-by T L*)) *S'*
(*abort*, *fst* ‘# *mset* (*drop i* (*watched-by T L*)), *mark*, *T'*) ∧ *correct-watching'-leaking-bin T* ∧
literals-are-$\mathcal{L}_{in}$*' S* ∧ *get-watched-wl T* = *get-watched-wl S*))›

lemma *deduplicate-binary-clauses-inv-wl-literals-are-in*:
‹*deduplicate-binary-clauses-inv-wl S L* (*abort*, *i*, *mark*, *T*) ⟹
literals-are-$\mathcal{L}_{in}$*' T*›
supply [[*goals-limit=1*]]
unfolding *deduplicate-binary-clauses-inv-wl-def prod.simps*
deduplicate-binary-clauses-inv-def literals-are-$\mathcal{L}_{in}$*'-def blits-in-*$\mathcal{L}_{in}$*'-def*
apply *normalize-goal+*
apply (*frule rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l*)
apply (*frule rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*)
by (*auto simp*: *watched-by-alt-def*)

definition *deduplicate-binary-clauses-wl* :: ‹*'v literal* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*deduplicate-binary-clauses-wl L S* = *do* {
ASSERT (*deduplicate-binary-clauses-pre-wl L S*);
let CS = (λ-. *None*);
let l = *length* (*watched-by S L*);
(-, -, -, *S*) ← *WHILE*$_T$*deduplicate-binary-clauses-inv-wl S L* (λ(*abort*, *i*, *CS*, *S*). ¬*abort* ∧ *i* < *l* ∧
get-conflict-wl S = *None*)
(λ(*abort*, *i*, *CS*, *S*).
do {
let (*C*,*L'*, *b*) = (*watched-by S L* ! *i*);
if C ∉# *dom-m* (*get-clauses-wl S*) ∨ ¬*b then*

```
        RETURN (abort, i+1, CS, S)
      else do {
        let L' = L';
        if defined-lit (get-trail-wl S) L' then do {
          U ← simplify-clause-with-unit-st-wl C S;
          ASSERT (set-mset (all-init-atms-st U) = set-mset (all-init-atms-st S));
          RETURN (defined-lit (get-trail-wl U) L, i+1, CS, U)
        }
        else if CS L' ≠ None then do {
         let C' = (if ¬snd (the (CS L')) ⟶ ¬irred (get-clauses-wl S) C then C else fst (the (CS L')));
           let CS = (if ¬snd (the (CS L')) ⟶ ¬irred (get-clauses-wl S) C then CS else CS (L' :=
Some (C, irred (get-clauses-wl S) C)));
          S ← clause-remove-duplicate-clause-wl C' S;
          RETURN (abort, i+1, CS, S)
        } else if CS (−L') ≠ None then do {
          S ← binary-clause-subres-wl C L (−L') S;
          RETURN (True, i+1, CS, S)
        } else do {
          RETURN (abort, i+1, CS (L' := Some (C, irred (get-clauses-wl S) C)), S)
        }
      }
    })
    (defined-lit (get-trail-wl S) L, 0, CS, S);
  RETURN S
}›
```

lemma *correct-watching'-leaking-bin-pure-lit*:
  assumes
    ‹*correct-watching'-leaking-bin* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *ja*, *k*, *l*, *m*)›
    ‹*L* ∈# *all-init-lits-of-wl* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *ja*, *k*, *l*, *m*)›
  shows ‹*correct-watching'-leaking-bin*
    (*Propagated L 0* # *a*, *b*, *c*, *d*, *e*,
     add-mset {#*L*#} *f*, *g*, *h*, *i*, *ja*, *k*,
     add-mset (−*L*) *l*, *m*)›
proof −
  have [*simp*]: ‹*set-mset* (*all-init-lits-of-wl*
        ([], *b*, *c*, *d*, {#}, *add-mset* {#*L*#} *f*, {#}, *h*,
    {#}, *ja*, {#}, *add-mset* (−*L*) *l*, *m*)) =
    set-mset (*all-init-lits-of-wl*
    ([], *b*, *c*, *d*, {#}, *f*, {#}, *h*, {#}, *ja*, {#}, *l*, *m*))›
    using *assms* **unfolding** *all-init-lits-of-wl-def*
    by (*auto simp*: *all-init-lits-of-wl-def all-lits-of-mm-add-mset*
      all-lits-of-m-add-mset all-lits-of-mm-union in-all-lits-of-mm-uminus-iff)
  show *?thesis*
    using *assms*
    by (*auto simp*: *correct-watching'-leaking-bin.simps clause-to-update-def*)
qed

lemma *correct-watching'-leaking-bin-propagate-unit-irred*:
  assumes
    ‹*irred b j*› **and**
    ‹*j* ∈# *dom-m b*› **and**
    ‹*correct-watching'-leaking-bin* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *ja*, *k*, *l*, *m*)›
    ‹*L* ∈ *set* (*b* ∝ *j*)›
  shows ‹*correct-watching'-leaking-bin*

```
    (Propagated L 0 # a, b, c, d, e,
     add-mset {#L#} f, g, h, i, ja, k,
     add-mset (−L) l, m)›
proof −
  have ‹L ∈# all-init-lits-of-wl (a, b, c, d, e, f, g, h, i, ja, k, l, m)›
    using assms by (auto simp: all-lits-of-mm-union all-lits-of-mm-add-mset all-init-lits-of-wl-def
      ran-m-def in-clause-in-all-lits-of-m
      dest!: multi-member-split)
  from correct-watching'-leaking-bin-pure-lit[OF - this] show ?thesis
    using assms by blast
qed


lemma correct-watching'-leaking-bin-propagate-unit-red:
  assumes
    ‹¬irred b j› and
    ‹j ∈# dom-m b› and
    ‹correct-watching'-leaking-bin (a, b, None, d, e, f, g, h, i, ja, k, l, m)›
    ‹L ∈ set (b ∝ j)›
  shows ‹correct-watching'-leaking-bin
    (Propagated L 0 # a, b, None, d, e,
     f, add-mset {#L#} g, h, i, ja, k,
     add-mset (−L) l, m)›
proof −
  have [simp]: ‹set-mset (all-init-lits-of-wl
        ([], b, None, d, {#}, f, {#}, h, {#}, ja, {#},
        add-mset (− L) l, m)) =
    set-mset (all-init-lits-of-wl
     ([], b, None, d, {#}, f, {#}, h, {#}, ja, {#},
       l, m))›
    using assms unfolding all-init-lits-of-wl-def
    by (auto simp: all-init-lits-of-wl-def all-lits-of-mm-add-mset
      all-lits-of-m-add-mset all-lits-of-mm-union)
  show ?thesis
    using assms
    by (auto simp: correct-watching'-leaking-bin.simps clause-to-update-def)
qed

lemma correct-watching'-leaking-bin-empty-conflict:
  ‹correct-watching'-leaking-bin (a, b, c, d, e, f, g, h, i, ja, k, l, m) ⟹
  correct-watching'-leaking-bin (a, b, Some {#}, d, e, f, g, h, i, add-mset {#} ja, k, {#}, m)›
  ‹correct-watching'-leaking-bin (a, b, c, d, e, f, g, h, i, ja, k, l, m) ⟹
  correct-watching'-leaking-bin (a, b, Some {#}, d, e, f, g, h, i, ja, add-mset {#} k, {#}, m)›
  by (auto simp: correct-watching'-leaking-bin.simps all-init-lits-of-wl-def
    all-lits-of-mm-add-mset all-lits-of-m-add-mset clause-to-update-def)
```

For binary clauses, we can prove a stronger version of ⟦(*?S*, *?T*) ∈ *state-wl-l None*; (*?i*, *?j*) ∈ *nat-rel*; *no-lost-clause-in-WL ?S*⟧ ⟹ *simplify-clause-with-unit-st-wl ?i ?S* ≤ ⇓ {(*S′*, *T*). (*S′*, *T*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S′* ∧ *get-watched-wl S′* = *get-watched-wl ?S*} (*simplify-clause-with-unit-st ?j ?T*), because watched literals do not have to be changed.

```
lemma simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st-correct-watching:
  assumes ST: ‹(S, T) ∈ state-wl-l None› and ij: ‹(i,j) ∈ nat-rel› and
    point: ‹correct-watching'-leaking-bin S› and
    ‹i ∈# dom-m (get-clauses-wl S)› and
    Si: ‹length (get-clauses-wl S ∝ i) = 2›
```

shows
‹*simplify-clause-with-unit-st-wl i S* ≤ ⇓ {(*S'*,*T*). (*S'*,*T*) ∈ *state-wl-l None* ∧
correct-watching'-leaking-bin S' ∧
get-watched-wl S' = *get-watched-wl S*}
(*simplify-clause-with-unit-st j T*)›
proof −
have *Id*: ‹*A* = *B* ⟹ *A* ≤ ⇓*Id B*› **for** *A B*
by *auto*
have *ij*: ‹*i* = *j*›
using *assms* **by** *auto*
have [*simp*]:
‹*irred b j* ⟹ *j* ∈# *dom-m b* ⟹ *add-mset* (*mset* (*b* ∝ *j*))
({#*mset* (*fst x*). *x* ∈# *remove1-mset* (*the* (*fmlookup b j*)) (*init-clss-l b*)#} + *d* + *f* + *h*) =
({#*mset* (*fst x*). *x* ∈# (*init-clss-l b*)#} + *d* + *f* + *h*)› **for** *C D d b f h j*
by (*auto simp*: *image-mset-remove1-mset-if ran-m-def*
dest!: *multi-member-split*)
have *KK*[*simp*]: ‹*irred b j* ⟹ *j* ∈# *dom-m b* ⟹ *C* ⊆# *mset* (*b* ∝ *j*) ⟹
set-mset (*all-lits-of-mm* (*add-mset* (*mset* (*b* ∝ *j*))
(*add-mset C*
(*mset* '# *remove1-mset* (*b* ∝ *j*) (*init-clss-lf b*) + *d* + *f* + *h*)))) =
set-mset (*all-lits-of-mm* (*mset* '# (*init-clss-lf b*) + *d* + *f* + *h*))›
for *b j C d f h*
using *all-lits-of-m-mono*[*of C* ‹*mset* (*b* ∝ *j*)›]
by (*auto simp*: *image-mset-remove1-mset-if ran-m-def conj-disj-distribR Collect-disj-eq*
image-Un Collect-conv-if all-lits-of-mm-add-mset
simp flip: *insert-compr*
dest!: *multi-member-split*[*of j*])

have *H*: ‹*fmdrop j x2* = *fmdrop j b* ⟹
mset (*x2* ∝ *j*) ⊆# *mset* (*b* ∝ *j*) ⟹
irred x2 j ⟹
irred b j ⟹
j ∈# *dom-m b* ⟹
j ∈# *dom-m x2* ⟹
set-mset (*all-lits-of-mm* (*add-mset* (*mset* (*b* ∝ *j*)) ({#*mset* (*fst x*). *x* ∈# *init-clss-l x2*#} +
d + *f* + *h*))) =
set-mset (*all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *init-clss-l b*#} + *d* + *f* + *h*))›
for *j x2 b d f h*
using *distinct-mset-dom*[*of x2*] *distinct-mset-dom*[*of b*]
apply (*subgoal-tac* ‹{#*mset* (*fst x*). *x* ∈# *filter-mset snd* {#*the* (*fmlookup b x*). *x* ∈# *remove1-mset*
j (*dom-m b*)#}#} =
{#*mset* (*fst x*). *x* ∈# *filter-mset snd* {#*the* (*fmlookup x2 x*). *x* ∈# *remove1-mset j* (*dom-m x2*)#}
#}›)
apply (*auto 5 2 simp*: *ran-m-def all-lits-of-mm-add-mset*
dest!: *multi-member-split*[*of* - ‹*dom-m* -›]
dest: *all-lits-of-m-mono*
intro!: *image-mset-cong2 filter-mset-cong2*)
apply (*metis all-lits-of-m-union mset-subset-eq-exists-conv union-iff*)
apply (*metis fmdrop-eq-update-eq fmupd-lookup union-single-eq-member*)
apply (*metis add-mset-remove-trivial dom-m-fmdrop*)
done
have [*simp*]: ‹*mset a* ⊆# *mset b* ⟹ *length a*= *1* ⟹ *a* ! *0* ∈ *set b*› **for** *a b*
by (*cases a*, *auto*)
have *K1*: ‹∀ *L*∈#*all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *init-clss-l b*#} + *d* + *f* + *h*).
distinct-watched (*k L*) ⟹
irred b j ⟹

```
    j ∈# dom-m b ⟹
    L ∈# all-lits-of-m (mset (b ∝ j)) ⟹ distinct-watched (k L)› for b d f h k j L
    by (auto simp: ran-m-def all-lits-of-mm-add-mset dest!: multi-member-split)
  have K2: ‹∀ L∈#all-lits-of-mm ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h).
    distinct-watched (k L) ⟹
    irred b j ⟹
    j ∈# dom-m b ⟹
    mset C ⊆# mset (b ∝ j) ⟹
    length C = Suc 0 ⟹
    L ∈# all-lits-of-m ({#C!0#}) ⟹ distinct-watched (k L)› for b d f h k j L C
    using all-lits-of-m-mono[of ‹mset C› ‹mset (b ∝ j)›]
     all-lits-of-m-mono[of ‹{#C!0#}› ‹mset C›]
    by (auto simp: ran-m-def all-lits-of-mm-add-mset dest!: multi-member-split[of - ‹dom-m -›])
  have K3: ‹∀ L∈#all-lits-of-mm ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h).
    distinct-watched (k L) ⟹
    L ∈# all-lits-of-mm ({#mset (fst x). x ∈# remove1-mset (the (fmlookup b j)) (init-clss-l b)#} +
d + f + h) ⟹
    distinct-watched (k L)› for b d f h k j L C
    by (cases ‹j ∈# dom-m b›; cases ‹irred b j›)
     (auto  dest!: multi-member-split[of - ‹dom-m -›] simp: ran-m-def
        all-lits-of-mm-union all-lits-of-mm-add-mset image-mset-remove1-mset-if
      split: if-splits)
  have K4: ‹
    irred b j ⟹ j ∈# dom-m b ⟹
    all-lits-of-mm
    (add-mset (mset (b ∝ j))
    ({#mset (fst x). x ∈# init-clss-l (fmdrop j b)#} + d + f + h)) =
    all-lits-of-mm
    ({#mset (fst x). x ∈# init-clss-l b#} + d + f + h)›
    ‹¬irred b j ⟹ j ∈# dom-m b ⟹
    all-lits-of-mm
    (add-mset (mset (b ∝ j))
    ({#mset (fst x). x ∈# learned-clss-l (fmdrop j b)#} + d + f + h)) =
    all-lits-of-mm
    ({#mset (fst x). x ∈# learned-clss-l b#} + d + f + h)›
    for d f h j b
    using distinct-mset-dom[of b]
    apply (auto simp add: init-clss-l-fmdrop learned-clss-l-fmdrop-if)
    by (smt (z3) fmupd-same image-mset-add-mset learned-clss-l-mapsto-upd prod.collapse
       union-mset-add-mset-left)
```

This case is most likely impossible. It comes from the fact that we do not attempt to prove that the unchanged exactly captures when things are unchanged (missing backimplication).

```
  have correct-watching-after-same-length: ‹¬ irred b j ⟶
    correct-watching'-leaking-bin
    (a, x2a, None, d, e, f, g, h, add-mset (mset (b ∝ j)) i, ja, k,
    l, m)› (is ?A)
     ‹irred b j ⟶
    correct-watching'-leaking-bin
    (a, x2a, None, d, e, f, g, add-mset (mset (b ∝ j)) h, i, ja, k,
    l, m)› (is ?B)
    if
      st: ‹aa = a ∧
      ba = b ∧
      da = d ∧
      ea = e ∧
```

$fa = f \wedge ga = g \wedge ha = h \wedge ia = i \wedge jaa = ja \wedge ka = k \wedge ma = l\rangle$ **and**
corr: $\langle$*correct-watching'-leaking-bin* $(a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ l,\ m)\rangle$ **and**
S: $\langle S = (a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ l,\ m)\rangle$ **and**
$\langle T = (a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ \{\#\},\ l)\rangle$ **and**
$\langle$*simplify-clause-with-unit-st-pre j*
$(a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ \{\#\},\ l)\rangle$ **and**
$\langle ca = None \wedge la = \{\#\}\rangle$ **and**
$\langle$*simplify-clause-with-unit-st-wl-pre j*
$(a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ l,\ m)\rangle$ **and**
$\langle j \in\# dom\text{-}m\ b \wedge$
count-decided $a = 0 \wedge c = None \wedge$ *no-dup* $a \wedge 0 < j\rangle$ **and**
$\langle x2c = x2a\rangle$ **and**
$\langle x2 = (False,\ x2a)\rangle$ **and**
$\langle x' = (False,\ False,\ x2a)\rangle$ **and**
$\langle x2b = (False,\ x2a)\rangle$ **and**
$\langle x = (False,\ False,\ x2a)\rangle$ **and**
b: $\langle fmdrop\ j\ x2a = fmdrop\ j\ b \wedge$
irred $x2a\ j =$ *irred* $b\ j \wedge$
mset $(x2a \propto j) \subseteq\#$ *mset* $(b \propto j) \wedge j \in\#$ *dom-m* $x2a\rangle$ **and**
$\langle \neg\ x1b\rangle$ **and**
$\langle \neg\ x1\rangle$ **and**
$\langle \neg\ x1c\rangle$ **and**
$\langle \neg\ x1a\rangle$ **and**
le: $\langle$*length* $(x2a \propto j) \neq Suc\ 0\rangle$ $\langle x2a \propto j \neq []\rangle$ **and**
eq: $\langle$*set-mset*
(*all-init-lits-of-l*
$(a,\ x2a,\ None,\ d,\ e,\ f,\ g,$
(*if irred b j then add-mset* (*mset* $(ba \propto j)$) *else id*) $h,$
(*if irred b j then id else add-mset* (*mset* $(ba \propto j)$)) $i,\ ja,\ k,$
$\{\#\},\ l)) =$
set-mset
(*all-init-lits-of-l*
$(a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ \{\#\},\ l))\rangle$
$\langle$*set-mset*
(*all-learned-lits-of-l*
$(a,\ x2a,\ None,\ d,\ e,\ f,\ g,$
(*if irred b j then add-mset* (*mset* $(ba \propto j)$) *else id*) $h,$
(*if irred b j then id else add-mset* (*mset* $(ba \propto j)$)) $i,\ ja,\ k,$
$\{\#\},\ l)) =$
set-mset
(*all-learned-lits-of-l*
$(a,\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ \{\#\},\ l))\rangle$
$\langle$*set-mset*
(*all-learned-lits-of-wl*
$([],\ x2a,\ None,\ d,\ e,\ f,\ g,$
(*if irred b j then add-mset* (*mset* $(b \propto j)$) *else id*) $h,$
(*if irred b j then id else add-mset* (*mset* $(b \propto j)$)) $i,\ ja,\ k,$
$l,\ m)) =$
set-mset
(*all-learned-lits-of-wl*
$([],\ b,\ None,\ d,\ e,\ f,\ g,\ h,\ i,\ ja,\ k,\ l,\ m))\rangle$
$\langle$*set-mset*
(*all-init-lits-of-wl*
$([],\ x2a,\ None,\ d,\ \{\#\},\ f,\ \{\#\},$
(*if irred b j then add-mset* (*mset* $(b \propto j)$) *else id*) $h,\ \{\#\},\ ja,$
$\{\#\},\ l,\ m)) =$

```
    set-mset
    (all-init-lits-of-wl
    ([], b, None, d, {#}, f, {#}, h, {#}, ja, {#}, l, m))›
  for a b c d e f g h i ja k l m aa ba ca da ea fa ga ha ia jaa ka la ma
    x x' x1 x2 x1a x2a x1b x2b x1c x2c
proof −
  note [[goals-limit=1]]
  have [simp]: ‹aa ∈# dom-m x2a ⟹ length (x2a ∝ aa) = length (b ∝ aa)› for aa
    apply (cases ‹aa = j›)
    subgoal
      using le b Si ij size-mset-mono[of ‹mset (x2a ∝ aa)› ‹mset (b ∝ aa)›]
      by (cases ‹x2a ∝ aa›; cases ‹tl (x2a ∝ aa)›)
       (clarsimp simp add: length-list-2 S subseteq-mset-size-eql-iff add-mset-eq-add-mset)+
    subgoal
      using b apply −
      apply normalize-goal+
      by (drule arg-cong [of - - ‹λa. a ∝ aa›]) auto
    done
  have [simp]: ‹aa ∈# dom-m x2a ⟹ set (x2a ∝ aa) = set (b ∝ aa)› for aa
    apply (cases ‹aa = j›)
    subgoal
      using le b Si ij size-mset-mono[of ‹mset (x2a ∝ aa)› ‹mset (b ∝ aa)›]
      apply (simp add: S)
      apply (clarsimp-all simp add: length-list-2 S)
      by (cases ‹x2a ∝ aa›; cases ‹tl (x2a ∝ aa)›)
       (auto simp add: length-list-2 S subseteq-mset-size-eql-iff add-mset-eq-add-mset)
    subgoal
      using b apply −
      apply normalize-goal+
      by (drule arg-cong [of - - ‹λa. a ∝ aa›]) auto
    done
  have [simp]: ‹aa ∈# dom-m x2a ⟹ set (watched-l (x2a ∝ aa)) = set (watched-l (b ∝ aa))› for aa
    apply (cases ‹aa = j›)
    subgoal
      using le b Si ij size-mset-mono[of ‹mset (x2a ∝ aa)› ‹mset (b ∝ aa)›]
      by (simp add: S)
    subgoal
      using b apply −
      apply normalize-goal+
      by (drule arg-cong [of - - ‹λa. a ∝ aa›]) auto
    done
  have [simp]: ‹dom-m x2a = dom-m b›
    using b by (metis dom-m-fmdrop insert-DiffM that(8))
  show ?A ?B
    using corr eq
  by (auto simp: st correct-watching'-leaking-bin.simps clause-to-update-def correctly-marked-as-binary.simps
      intro!: filter-mset-cong)
qed
show ?thesis
  supply [[goals-limit=1]]
  using ST point
  apply (cases S; hypsubst)
  apply (cases T; hypsubst)
  unfolding simplify-clause-with-unit-st-wl-def simplify-clause-with-unit-st-def ij
    state-wl-l-def prod.simps Let-def[of ‹(-,-)›]
  apply refine-rcg
```

subgoal for *a b c d e f g h i ja k l m aa ba ca da ea fa ga ha ia jaa ka la ma*
using *ST*
unfolding *simplify-clause-with-unit-st-wl-pre-def*
by (*rule-tac x* = ‹(*aa*, *ba*, *ca*, *da*, *ea*, *fa*, *ga*, *ha*, *ia*, *jaa*, *ka*, *la*, *ma*)› **in** *exI*)
simp
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
apply (*rule Id*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal
by (*auto simp add*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def)
subgoal by (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop*
init-clss-l-fmdrop-irrelev literals-are-$\mathcal{L}_{in}$*'-def blits-in-*$\mathcal{L}_{in}$*'-def*
no-lost-clause-in-WL-def
intro: *correct-watching'-leaking-bin-remove-subsumedI*
dest: *in-diffD*)
subgoal by *auto*
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def all-init-lits-of-wl-def)
subgoal by *auto*
subgoal by (*auto simp*: *all-lits-st-alt-def all-learned-lits-of-wl-def*
all-init-lits-of-l-def all-init-lits-of-wl-def get-init-clss-l-def)
subgoal for *a b c d e f g h i ja k l m aa ba ca da ea fa ga ha ia jaa ka la ma*
x x' x1 x2 x1a x2a x1b x2b x1c x2c
apply (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop*
init-clss-l-fmdrop-irrelev add-mset-commute
no-lost-clause-in-WL-def
intro: *correct-watching'-leaking-bin-remove-subsumedI correct-watching'-leaking-bin-propagate-unit-irred*[**where** *j*=*j*]
correct-watching'-leaking-bin-propagate-unit-red[**where** *j*=*j*]
dest: *in-diffD*
intro:)
apply (*rule correct-watching'-leaking-bin-remove-subsumedI*)
apply (*rule correct-watching'-leaking-bin-propagate-unit-irred*[**where** *j*=*j*])
apply *auto*
apply (*rule correct-watching'-leaking-bin-remove-subsumedI*)
apply (*rule correct-watching'-leaking-bin-propagate-unit-red*[**where** *j*=*j*])
apply *auto*
done
subgoal by *auto*
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def)

subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def all-init-lits-of-wl-def)
subgoal by (*auto simp*: *all-init-lits-of-wl-def init-clss-l-fmdrop*
init-clss-l-fmdrop-irrelev all-lits-of-mm-add-mset
intro: *correct-watching'-leaking-bin-remove-subsumedI correct-watching'-leaking-bin-empty-conflict*
dest: *in-diffD*)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def)
subgoal by (*auto simp*: *all-learned-lits-of-wl-def all-init-lits-of-l-def*
all-learned-lits-of-l-def get-init-clss-l-def all-init-lits-of-wl-def)
subgoal for *a b c d e f g h i ja k l m aa ba ca da ea fa ga ha ia jaa ka la ma*
x x' x1 x2 x1a x2a x1b x2b x1c x2c
by (*simp*)
(*intro conjI*; *rule correct-watching-after-same-length*; *assumption*)+
done
qed

lemma *WHILEIT-refine-with-inv*:
assumes *R0*: $I'\ x' \Longrightarrow (x,x')\in R$
assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
shows *WHILEIT I b f x* $\le\Downarrow\{(a,b).\ (a,b) \in R \wedge I\ a \wedge I'\ b\}$ (*WHILEIT I' b' f' x'*)
apply (*rule WHILEIT-refine-with-post*)
subgoal using *R0 IREF* **by** *blast*
subgoal using *IREF* **by** *blast*
subgoal using *COND-REF* **by** *blast*
subgoal using *STEP-REF IREF* **by** (*smt* (*verit*, *best*) *in-pair-collect-simp inres-SPEC pw-ref-iff*)
done

lemma *deduplicate-binary-clauses-wl-deduplicate-binary-clauses*:
assumes ‹$(L,L')\in Id$› ‹$(S,S')\in$ *state-wl-l None*›
‹*correct-watching'-leaking-bin S*› ‹*literals-are-*$\mathcal{L}_{in}$*' S*›
shows ‹*deduplicate-binary-clauses-wl L S* $\le$
$\Downarrow\{(S,\ T).\ (S,T)\in$ *state-wl-l None* $\wedge$ *correct-watching'-leaking-bin S* $\wedge$ *literals-are-*$\mathcal{L}_{in}$*' S*$\}$ (*deduplicate-binary-clauses L' S'*)›
proof −
let *?watched* = ‹$\{(i,\ CS).\ i = (length\ (watched\text{-}by\ S\ L)) \wedge CS = fst\ `\#\ mset\ (watched\text{-}by\ S\ L) \wedge$
$(\forall C.\ (C \in\#\ CS \longrightarrow C \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}l\ S') \longrightarrow L' \in set\ (get\text{-}clauses\text{-}l\ S' \propto C)))\}$›
have [*refine0*]: ‹*deduplicate-binary-clauses-pre-wl L S* $\Longrightarrow$ *RETURN* (*length* (*watched-by S L*))
$\le \Downarrow$ *?watched* (*SPEC* ($\lambda CS.\ \forall C.\ (C \in\#\ CS \longrightarrow C \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}l\ S') \longrightarrow L' \in set$
$(get\text{-}clauses\text{-}l\ S' \propto C)) \wedge distinct\text{-}mset\ CS$))›
using *assms*
apply (*cases S*)
apply (*auto simp*: *RETURN-RES-refine-iff correct-watching'-leaking-bin.simps*
deduplicate-binary-clauses-pre-wl-def deduplicate-binary-clauses-pre-def)
apply (*drule bspec*)
apply *assumption*
apply *auto*
apply (*drule bspec*)
apply *assumption*
apply (*auto simp*: *clause-to-update-def distinct-mset-image-mset distinct-watched-alt-def dest*: *in-set-takeD*
dest!: *multi-member-split split*: *if-splits*)
apply (*meson in-set-takeD*)

apply (*smt* (*z3*) *filter-mset-add-mset filter-mset-eq-add-msetD fst-conv image-mset-add-mset in-multiset-in-set multi-member-split*)
done
let *?S* = ‹{((*abort*, *i*, *CS*, *U*), (*abort'*, *xs*, *CS'*, *U'*)). *abort*=*abort'* ∧ *fst* '# *mset* (*drop i* (*watched-by S L*)) = *xs* ∧ *CS*=*CS'* ∧
(*U*,*U'*)∈ *state-wl-l None* ∧ *get-watched-wl U* = *get-watched-wl S* ∧ *correct-watching'-leaking-bin U*}›
have [*refine0*]: ‹(*length* (*watched-by S L*), *xs*) ∈ *?watched* ⟹
((*defined-lit* (*get-trail-wl S*) *L*, *0*, *Map.empty*, *S*), *defined-lit* (*get-trail-l S'*) *L'*, *xs*,
Map.empty, *S'*) ∈ *?S*› **for** *xs*
using *assms* **by** *auto*
have *watched-by*: ‹*RETURN* (*watched-by x2e L* ! *x1d*)
≤ ⇓ {((*C*, *K*, *b*), *C'*). *C*=*C'* ∧
(*C* ∈# *dom-m* (*get-clauses-wl x2e*) ⟶ *b* = (*length* (*get-clauses-wl x2e* ∝ *C*) = *2*) ∧ *K* ≠ *L* ∧ *K* ∈ *set* (*get-clauses-wl x2e* ∝ *C*) ∧ *L* ∈ *set* (*get-clauses-wl x2e* ∝ *C*))}
(*SPEC* (λ*C*. *C* ∈# *x1a*))› (**is** ‹- ≤ ⇓*?watch* -›)
if
LL': ‹(*L*, *L'*) ∈ *Id*› **and**
SS': ‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
‹*correct-watching'-leaking-bin S*› **and**
‹*literals-are-*$\mathcal{L}_{in}$' *S*› **and**
pre: ‹*deduplicate-binary-clauses-pre L' S'*› **and**
‹*deduplicate-binary-clauses-pre-wl L S*› **and**
watched: ‹(*length* (*watched-by S L*), *xs*) ∈ *?watched*› **and**
xx': ‹(*x*, *x'*) ∈ *?S*› **and**
abort: ‹*case x of*
(*abort*, *i*, *CS*, *Sa*) ⇒
¬ *abort* ∧ *i* < *length* (*watched-by S L*) ∧ *get-conflict-wl Sa* = *None*› **and**
‹*case x' of*
(*abort*, *xs*, *CS*, *S*) ⇒ ¬ *abort* ∧ *xs* ≠ {#} ∧ *get-conflict-l S* = *None*› **and**
inv: ‹*deduplicate-binary-clauses-inv-wl S L x*› **and**
inv2: ‹*deduplicate-binary-clauses-inv L' xs S' x'*› **and**
st: ‹*x2a* = (*x1b*, *x2b*)›
‹*x2* = (*x1a*, *x2a*)›
‹*x'* = (*x1*, *x2*)›
‹*x2d* = (*x1e*, *x2e*)›
‹*x2c* = (*x1d*, *x2d*)›
‹*x* = (*x1c*, *x2c*)›
for *xs x x' x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e*
proof −
have *L-S*: ‹*L* ∈# *all-init-lits-of-wl S*› ‹*cdcl-twl-inprocessing-l*** *S' x2b*›
‹*set-mset* (*all-init-lits-of-l x2b*) = *set-mset* (*all-init-lits-of-l S'*)›
‹*set-mset* (*all-init-lits-of-wl x2e*) = *set-mset* (*all-init-lits-of-wl S*)›
using *pre SS' LL' inv2 xx'* **by** (*auto simp*: *deduplicate-binary-clauses-pre-def*
deduplicate-binary-clauses-inv-def st
dest: *rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l*)
have ‹*x1d* < *length* (*get-watched-wl S L*)›
using *xx' abort* **by** (*auto simp*: *watched-by-alt-def st*)
then have ‹*watched-by x2e L* ! *x1d* ∈ *set* (*get-watched-wl S L*)›
‹*get-watched-wl x2e* = *get-watched-wl S*›
‹*correct-watching'-leaking-bin x2e*›
‹*x1a* = *fst* '# *mset* (*drop x1d* (*watched-by S L*))›
‹*get-watched-wl S L* ! *x1d* ∈ *set* (*drop x1d* (*get-watched-wl S L*))›
using *xx' abort* **by** (*auto simp*: *watched-by-alt-def st intro*!: *in-set-dropI*)
moreover have ‹*fst* (*get-watched-wl S L* ! *x1d*) ∈# *dom-m* (*get-clauses-wl x2e*) ⟹
L ∈ *set* (*get-clauses-wl x2e* ∝ *fst* (*get-watched-wl S L* ! *x1d*))›

using *L-S(1) xx′ ‹watched-by x2e L ! x1d ∈ set (get-watched-wl S L)› xx′*
multi-member-split[of ‹watched-by x2e L ! x1d› ‹mset (get-watched-wl S L)›]
unfolding *L-S(4)[symmetric]*
by (*cases x2e*)
(*auto simp*: *watched-by-alt-def correct-watching′-leaking-bin.simps st*
clause-to-update-def dest!: *multi-member-split[of L]*
dest!: *filter-mset-eq-add-msetD′ in-set-takeD*)
ultimately show *?thesis*
using *L-S* **unfolding** *st*
by (*cases x2e*)
(*auto simp*: *RETURN-RES-refine-iff watched-by-alt-def eq-commute[of ‹(-, -, -)›]*
correct-watching′-leaking-bin.simps correctly-marked-as-binary.simps
intro!: *bexI[of - ‹watched-by x2e L ! x1d›]*
dest!: *multi-member-split[of L]*)
qed
have *correct-blit*: ‹*mset (get-clauses-l x2b ∝ C) = {#L′, x1g#}*›
if
LL′: ‹*(L, L′) ∈ Id*› **and**
SS′: ‹*(S, S′) ∈ state-wl-l None*› **and**
‹*correct-watching′-leaking-bin S*› **and**
‹*literals-are-$\mathcal{L}_{in}$′ S*› **and**
pre: ‹*deduplicate-binary-clauses-pre L′ S′*› **and**
‹*deduplicate-binary-clauses-pre-wl L S*› **and**
xs: ‹*(length (watched-by S L), xs) ∈ ?watched*› **and**
xx′: ‹*(x, x′) ∈ ?S*› **and**
abort: ‹*case x of*
(abort, i, CS, Sa) ⇒
¬ abort ∧
i < length (watched-by S L) ∧ get-conflict-wl Sa = None› **and**
‹*case x′ of*
(abort, xs, CS, S) ⇒
¬ abort ∧ xs ≠ {#} ∧ get-conflict-l S = None› **and**
inv: ‹*deduplicate-binary-clauses-inv-wl S L x*› **and**
inv2: ‹*deduplicate-binary-clauses-inv L′ xs S′ x′*› **and**
st: ‹*x2a = (x1b, x2b)*›
‹*x2 = (x1a, x2a)*›
‹*x′ = (x1, x2)*›
‹*x2d = (x1e, x2e)*›
‹*x2c = (x1d, x2d)*›
‹*x = (x1c, x2c)*› **and**
watch: ‹*(watched-by x2e L ! x1d, C) ∈ ?watch x2e*› **and**
watcher: ‹*watched-by x2e L ! x1d = (x1f, x2f)*›
‹*x2f = (x1g, x2g)*› **and**
bin: ‹*¬ (x1f ∉# dom-m (get-clauses-wl x2e) ∨ ¬ x2g)*›
‹*¬ (C ∉# dom-m (get-clauses-l x2b) ∨ length (get-clauses-l x2b ∝ C) ≠ 2)*›
for *xs x x′ x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e C x1f x2f x1g*
x2g
proof −
show *?thesis*
using *watch xs bin SS′ xx′ LL′ watcher*
by (*auto simp*: *st length-list-2 watched-by-alt-def*)
qed

have *post-inv*: ‹*(x2e, x2b)*
∈ {(S, T).
(S, T) ∈ state-wl-l None ∧

```
      correct-watching'-leaking-bin S ∧ literals-are-ℒ_in' S}›
  if
    ‹(L, L') ∈ Id› and
    SS': ‹(S, S') ∈ state-wl-l None› and
    ‹correct-watching'-leaking-bin S› and
    ‹literals-are-ℒ_in' S› and
    ‹deduplicate-binary-clauses-pre L' S'› and
    ‹deduplicate-binary-clauses-pre-wl L S› and
    ‹(length (watched-by S L), xs) ∈ ?watched› and
    xx': ‹(x, x') ∈{(a,b). (a,b) ∈ ?S ∧
      deduplicate-binary-clauses-inv-wl S L a ∧
     deduplicate-binary-clauses-inv L' xs S' b}› and
    st: ‹x2a = (x1b, x2b)›
      ‹x2 = (x1a, x2a)›
      ‹x' = (x1, x2)›
      ‹x2d = (x1e, x2e)›
      ‹x2c = (x1d, x2d)›
      ‹x = (x1c, x2c)›
    for xs x x' x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e
proof −
  show ?thesis
  using xx' assms(4) SS' apply − unfolding mem-Collect-eq prod.simps st deduplicate-binary-clauses-inv-def
    apply normalize-goal+
    using rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l[of S' x2b]
      rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l[of S' x2b]
    by (auto simp add: literals-are-ℒ_in'-def st blits-in-ℒ_in'-def watched-by-alt-def)
qed
show ?thesis
  supply [[goals-limit=1]]
  using assms unfolding deduplicate-binary-clauses-wl-def deduplicate-binary-clauses-def
  apply (refine-vcg simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st-correct-watching
    clause-remove-duplicate-clause-wl-clause-remove-duplicate-clause
    binary-clause-subres-wl-binary-clause-subres WHILEIT-refine-with-inv)
  subgoal unfolding deduplicate-binary-clauses-pre-wl-def
    by fast
  subgoal for xs x x'
    unfolding deduplicate-binary-clauses-inv-wl-def prod.simps case-prod-beta
    apply (rule exI[of - S'])
    apply (rule exI[of - ‹snd (snd (snd x'))›])
    by (auto simp: watched-by-alt-def)
  subgoal for xs x x'
    by auto
  apply (rule watched-by; assumption)
  subgoal for xs x x' x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e
    by (auto intro!: imageI in-set-dropI simp: watched-by-alt-def)
  subgoal by (auto simp flip: Cons-nth-drop-Suc simp: watched-by-alt-def)
  subgoal by (rule correct-blit)
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal
    by (clarsimp dest!: rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l
      simp: all-init-atms-st-alt-def)
```

subgoal by (*auto simp flip*: *Cons-nth-drop-Suc simp*: *watched-by-alt-def*
dest: *deduplicate-binary-clauses-inv-wl-literals-are-in*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp flip*: *Cons-nth-drop-Suc simp*: *watched-by-alt-def*
dest: *deduplicate-binary-clauses-inv-wl-literals-are-in*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp flip*: *Cons-nth-drop-Suc simp*: *watched-by-alt-def*
dest: *deduplicate-binary-clauses-inv-wl-literals-are-in*)
subgoal by (*auto simp flip*: *Cons-nth-drop-Suc simp*: *watched-by-alt-def*
dest: *deduplicate-binary-clauses-inv-wl-literals-are-in*)
subgoal by (*auto simp flip*: *Cons-nth-drop-Suc simp*: *watched-by-alt-def*
dest: *deduplicate-binary-clauses-inv-wl-literals-are-in*)
subgoal by (*rule post-inv*)
done
qed

definition *mark-duplicated-binary-clauses-as-garbage-pre-wl* :: ‹*'v twl-st-wl* ⇒ *bool*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-pre-wl* = (λ*S*. (∃ *T*. (*S*,*T*) ∈ *state-wl-l None* ∧
correct-watching'-leaking-bin S ∧ *literals-are-*$\mathcal{L}_{in}$*' S*))›

definition *mark-duplicated-binary-clauses-as-garbage-wl-inv* :: ‹*'v multiset* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl*
× *'v multiset* ⇒ *bool*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-wl-inv* = (λ*xs S* (*U*, *ys*).
∃ *S' U'*. (*U*, *U'*) ∈ *state-wl-l None* ∧ (*S*, *S'*) ∈ *state-wl-l None* ∧
mark-duplicated-binary-clauses-as-garbage-inv xs S' (*U'*, *ys*))›

definition *mark-duplicated-binary-clauses-as-garbage-wl* :: ‹- ⇒ *'v twl-st-wl nres*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-wl S* = *do* {
ASSERT (*mark-duplicated-binary-clauses-as-garbage-pre-wl S*);
Ls ← *SPEC* (λ*Ls*:: *'v multiset*. ∀ *L*∈#*Ls*. *L* ∈# *atm-of* '# *all-init-lits-of-wl S*);
(*S*, -) ← *WHILE*$_T$*mark-duplicated-binary-clauses-as-garbage-wl-inv Ls S*(λ(*S*, *Ls*). *Ls* ≠ {#} ∧ *get-conflict-wl*
S = *None*)
(λ(*S*, *Ls*). *do* {
L ← *SPEC* (λ*L*. *L* ∈# *Ls*);
ASSERT (*L* ∈# *atm-of* '# *all-init-lits-of-wl S*);
skip ← *RES* (*UNIV* :: *bool set*);
if skip then RETURN (*S*, *remove1-mset L Ls*)
else do {
S ← *deduplicate-binary-clauses-wl* (*Pos L*) *S*;
S ← *deduplicate-binary-clauses-wl* (*Neg L*) *S*;
RETURN (*S*, *remove1-mset L Ls*)
}
})
(*S*, *Ls*);
RETURN S
}›

lemma *mark-duplicated-binary-clauses-as-garbage-wl*:

assumes ‹$(S,\ S') \in \{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching'-leaking-bin* $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S\}$›
shows ‹*mark-duplicated-binary-clauses-as-garbage-wl* $S \leq$
$\Downarrow \{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching'-leaking-bin* $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S\}$
(*mark-duplicated-binary-clauses-as-garbage* S')›
proof −
let *?R* = ‹$\{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching'-leaking-bin* $S \wedge$ *literals-are-*$\mathcal{L}_{in}'$ $S\}$ $\times_f$ *Id*›
have *loop*: ‹$(Ls,Ls') \in Id \Longrightarrow ((S,\ Ls),\ (S',\ Ls')) \in$ *?R*› **for** *Ls Ls'*
using *assms* **by** *auto*
show *?thesis*
unfolding *mark-duplicated-binary-clauses-as-garbage-wl-def mark-duplicated-binary-clauses-as-garbage-def*
apply (*refine-vcg loop deduplicate-binary-clauses-wl-deduplicate-binary-clauses*)
subgoal using *assms* **unfolding** *mark-duplicated-binary-clauses-as-garbage-pre-wl-def* **apply** −
by (*rule exI*[*of* - *S'*]) *auto*
subgoal using *assms* **by** *auto*
subgoal for *Ls Lsa x x'*
unfolding *mark-duplicated-binary-clauses-as-garbage-wl-inv-def*
apply (*cases x*, *cases x'*, *hypsubst*, *unfold prod.simps*)
apply (*rule exI*[*of* - *S'*], *rule exI*[*of* - ‹*fst x'*›])
apply (*use assms* **in** *simp*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

Large clauses

definition *subsume-clauses-match-pre* :: ‹- › **where**
‹*subsume-clauses-match-pre C C' N* $\longleftrightarrow$
length $(N \propto C) \leq$ *length* $(N \propto C') \wedge C \in\#$ *dom-m* $N \wedge C' \in\#$ *dom-m* $N \wedge$ *distinct* $(N \propto C) \wedge$ *distinct* $(N \propto C') \wedge$
$\neg$*tautology* (*mset* $(N \propto C)$)›

definition *subsume-clauses-match* :: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ (*nat*, *'v literal list* $\times$ *bool*) *fmap* $\Rightarrow$ *'v subsumption nres*› **where**
‹*subsume-clauses-match C C' N* = *do* {
ASSERT (*subsume-clauses-match-pre C C' N*);
$(i,\ st) \leftarrow$ *WHILE*$_T$ $\lambda(i,s).$ *try-to-subsume C' C* (*N* (*C* $\hookrightarrow$ *take i* $(N \propto C)$)) *s* $(\lambda(i,\ st).\ i <$ *length* $(N \propto C) \wedge st \neq$ *NONE*)
$(\lambda(i,\ st).$ *do* {
let L = $N \propto C$! *i*;

```
    if L ∈ set (N ∝ C')
    then RETURN (i+1, st)
    else if −L ∈ set (N ∝ C')
    then if is-subsumed st
    then RETURN (i+1, STRENGTHENED-BY L C)
    else RETURN (i+1, NONE)
    else RETURN (i+1, NONE)
   })
    (0, SUBSUMED-BY C);
  RETURN st
  }›

lemma subset-remove1-mset-notin:
  ‹b ∉# A ⟹ A ⊆# remove1-mset b B ⟷ A⊆#B›
  by (metis diff-subset-eq-self mset-le-subtract remove1-mset-eqE subset-mset.order-trans)

lemma subsume-clauses-match:
  assumes ‹subsume-clauses-match-pre C C' N›
  shows ‹subsume-clauses-match C C' N ≤ ⇓ Id (SPEC(try-to-subsume C' C N))›
proof −
  let ?R = ‹measure (λ(i, -). Suc (length(N ∝ C)) − i)›
  have [refine]: ‹wf ?R›
    by auto
  have H: ‹distinct-mset(mset (N ∝ C))› ‹distinct (N ∝ C')›
    using assms by (auto simp: subsume-clauses-match-pre-def)
  then have [simp]: ‹a < length (N ∝ C) ⟹ distinct-mset (add-mset (N ∝ C ! a) (mset (take a (N
∝ C))))›
    ‹a < length (N ∝ C) ⟹ distinct-mset ((mset (take a (N ∝ C))))›for a
    by (simp-all add: distinct-in-set-take-iff)
  then have [simp]: ‹a < length (N ∝ C) ⟹ distinct-mset (add-mset (N ∝ C ! a) (remove1-mset L
(mset (take a (N ∝ C)))))› for a L
    using diff-subset-eq-self distinct-mset-add-mset in-diffD distinct-mset-mono by metis
  have neg-notin: ‹a < length (N ∝ C) ⟹− N ∝ C ! a ∉ set (N ∝ C)› for a
    using assms
    by (smt (z3) mset-le-trans mset-lt-single-iff nth-mem set-mset-mset subsume-clauses-match-pre-def
tautology-minus)
  have neg-notin2: ‹a < length (N ∝ C) ⟹− N ∝ C ! a ∉ set (take a (N ∝ C))› for a
    using assms by (meson in-set-takeD neg-notin)
  have [simp]: ‹fmupd C (the (fmlookup N C)) N = N›
    by (meson assms fmupd-same subsume-clauses-match-pre-def)
  have [simp]: ‹try-to-subsume C' C N NONE›
    by (auto simp: try-to-subsume-def)
  have [simp]: ‹a < length (N ∝ C) ⟹
    x21 ∈ set (take a (N ∝ C)) ⟹
    N ∝ C ! a ∈# remove1-mset (− x21) (mset (N ∝ C')) ⟷ N ∝ C ! a ∈# mset (N ∝ C')› for a
x21
    apply (cases ‹(− x21) ∈# mset (N ∝ C')›)
    apply (drule multi-member-split)
    by (auto simp del: set-mset-mset in-multiset-in-set simp: uminus-lit-swap neg-notin2
      eq-commute[of ‹N ∝ C ! -›])
  show ?thesis
    unfolding subsume-clauses-match-def
    apply (refine-vcg)
    subgoal using assms by auto
    subgoal by (auto simp add: try-to-subsume-def)
    subgoal for s a b x1 x2
```

by (*auto 9 7 simp*: *try-to-subsume-def take-Suc-conv-app-nth subset-remove1-mset-notin neg-notin2*
split: *subsumption.splits*
simp del: *distinct-mset-add-mset*
simp flip: *distinct-subseteq-iff*)
subgoal
by *auto*
subgoal for *s a b x1 x2*
by (*auto 7 4 simp*: *try-to-subsume-def take-Suc-conv-app-nth subset-remove1-mset-notin neg-notin2*
split: *subsumption.splits*
simp del: *distinct-mset-add-mset*
simp flip: *distinct-subseteq-iff*)
subgoal by *auto*
subgoal for *s a b x1 x2*
by (*auto 7 4 simp*: *try-to-subsume-def take-Suc-conv-app-nth*
split: *subsumption.splits*
simp del: *distinct-mset-add-mset*
simp flip: *distinct-subseteq-iff*)
subgoal by *auto*
subgoal by (*auto simp*: *try-to-subsume-def*)
subgoal by *auto*
subgoal by *auto*
done
qed

definition *subsume-or-strengthen-wl-pre* :: ‹*nat* ⇒ *'v subsumption* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*subsume-or-strengthen-wl-pre C s S* ⟷ (∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧
subsume-or-strengthen-pre C s T ∧ *length* (*get-clauses-wl S* ∝ *C*) > *2*)›

definition *strengthen-clause-wl* :: ‹*nat* ⇒ *nat* ⇒ *'v literal* ⇒
'v twl-st-wl ⇒ *'v twl-st-wl nres*› **where**
‹*strengthen-clause-wl* = (λ*C C' L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT (*subsume-or-strengthen-wl-pre C* (*STRENGTHENED-BY L C'*) (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
E ← *SPEC* (λ*E*. *mset E* = *mset* (*remove1* (−*L*) (*N* ∝ *C*)));
if length (*N* ∝ *C*) = *2*
then do {
ASSERT (*length* (*remove1* (−*L*) (*N* ∝ *C*)) = *1*);
let L = *hd E*;
RETURN (*Propagated L 0* # *M*, *fmdrop C'* (*fmdrop C N*), *D*,
(*if irred N C' then add-mset* (*mset* (*N* ∝ *C'*)) *else id*) *NE*,
(*if* ¬*irred N C' then add-mset* (*mset* (*N* ∝ *C'*)) *else id*) *UE*,
(*if irred N C then add-mset* {#*L*#} *else id*) *NEk*, (*if* ¬*irred N C then add-mset* {#*L*#} *else id*) *UEk*,
((*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*)) *NS*,
((*if* ¬*irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*)) *US*,
N0, *U0*, *add-mset* (−*L*) *Q*, *W*)
}
else if length (*N* ∝ *C*) = *length* (*N* ∝ *C'*)
then RETURN (*M*, *fmdrop C'* (*fmupd C* (*E*, *irred N C* ∨ *irred N C'*) *N*), *D*, *NE*, *UE*, *NEk*, *UEk*,
((*if irred N C' then add-mset* (*mset* (*N* ∝ *C'*)) *else id*) *o* (*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*)) *NS*,
((*if* ¬*irred N C' then add-mset* (*mset* (*N* ∝ *C'*)) *else id*) *o* (*if* ¬*irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*)) *US*,
N0, *U0*, *Q*, *W*)
else RETURN (*M*, *fmupd C* (*E*, *irred N C*) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*,
(*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *NS*,

(*if* ¬*irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *US*, *N0*, *U0*, *Q*, *W*)})›

definition *subsume-or-strengthen-wl* :: ‹*nat* ⇒ *'v subsumption* ⇒ *'v twl-st-wl* ⇒ - *nres*› **where**
‹*subsume-or-strengthen-wl* = (λ*C s* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*). *do* {
ASSERT(*subsume-or-strengthen-wl-pre C s* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*));
(*case s of*
NONE ⇒ *RETURN* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)
| *SUBSUMED-BY C'* ⇒ *do* {
let T = (*M*, *fmdrop C* (*if* ¬*irred N C'* ∧ *irred N C then fmupd C'* (*N* ∝ *C'*, *True*) *N else N*), *D*,
NE, *UE*, *NEk*, *UEk*, (*if irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *NS*,
(*if* ¬*irred N C then add-mset* (*mset* (*N* ∝ *C*)) *else id*) *US*, *N0*, *U0*, *Q*, *W*);
ASSERT (*set-mset* (*all-init-atms-st T*) = *set-mset* (*all-init-atms-st* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)));
RETURN T
}
| *STRENGTHENED-BY L C'* ⇒ *strengthen-clause-wl C C' L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))
})›

definition *strengthen-clause-pre* :: ‹-› **where**
‹*strengthen-clause-pre xs C s t S* ⟷
distinct xs ∧ *C* ∈# *dom-m* (*get-clauses-wl S*) ›

lemma *no-lost-clause-in-WLI*:
‹*no-lost-clause-in-WL S* ⟹ *set-mset* (*dom-m* (*get-clauses-wl T*)) ⊆ *set-mset* (*dom-m* (*get-clauses-wl S*)) ⟹
set-mset (*all-init-lits-of-wl T*) = *set-mset* (*all-init-lits-of-wl S*) ⟹
get-watched-wl S = *get-watched-wl T* ⟹
no-lost-clause-in-WL T›
by (*auto simp*: *no-lost-clause-in-WL-def clauses-pointed-to-def watched-by-alt-def dest*!: *multi-member-split*)

lemma *filter-mset-remove-add-mset*: ‹*a* ∈# *M* ⟹
{#*x* ∈# *M* − *add-mset a M'*. *P x*#} = (*if P a then remove1-mset a* {#*x* ∈# *M* − *M'*. *P x*#} *else* {#*x* ∈# *M* − *M'*. *P x*#})›
by (*auto dest*!: *multi-member-split*)

lemma *image-mset-remove-add-mset*: ‹*a* ∈# *M* ⟹ *a* ∉# *M'* ⟹
{#*f x*. *x* ∈# *M* − *add-mset a M'*#} = *remove1-mset* (*f a*) {#*f x*. *x* ∈# *M* − *M'*#}›
by (*auto dest*!: *multi-member-split*)

lemma *strengthen-clause-wl-strengthen-clause*:
assumes
‹(*C*, *C'*) ∈ *nat-rel*› **and**
‹(*s*, *s'*) ∈ *nat-rel*› **and**
‹(*t*, *t'*) ∈ *Id*› **and**
‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
b: ‹*strengthen-clause-pre xs C s t S*› **and**
pre2: ‹*subsume-or-strengthen-wl-pre C* (*STRENGTHENED-BY t' s'*) *S*›
shows ‹*strengthen-clause-wl C s t S*
≤ ⇓ {(*T*, *T'*).
(*T*, *T'*) ∈ *state-wl-l None* ∧ *get-watched-wl T* = *get-watched-wl S*}
(*strengthen-clause C' s' t' S'*)›
proof −
have *dist*: ‹*distinct xs*›

using *b* **unfolding** *strengthen-clause-pre-def* **by** *auto*
have [*simp*]: ‹*x* ∉# *dom-m x1m* − {#*a*, *x*#}›
‹*x* ∉# *dom-m x1m* − {#*x*, *a*#}›
‹*x* ∉# *dom-m x1m* − {#*x*#}› **for** *x1m a x*
by (*smt* (*z3*) *add-mset-commute add-mset-diff-bothsides add-mset-remove-trivial-eq*
distinct-mset-add-mset distinct-mset-dom in-diffD)+
have *H*: ‹*C* ∈# *dom-m* (*get-clauses-wl S*)›
using *assms* **by** (*auto simp*: *strengthen-clause-pre-def*)
have [*simp*]: ‹*s'* ∈# *remove1-mset C* (*dom-m a*) ⟷ *s'* ≠ *C* ∧ *s'* ∈#*dom-m a*› **for** *a s' C*
by (*auto dest*: *in-diffD*)
show *?thesis*
supply [[*goals-limit=1*]]
using *assms*
unfolding *strengthen-clause-wl-def strengthen-clause-def*
apply *refine-vcg*
subgoal using *pre2* **by** *fast*
subgoal by (*auto simp*: *state-wl-l-def split*: *subsumption.splits*)
subgoal by (*auto simp*: *state-wl-l-def split*: *subsumption.splits*)
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h x1i x2i x1j*
x2j x1k x2k x1l x2l x1m x2m x1n x2n x1o x2o x1p x2p x1q x2q x1r x2r x1s x2s x1t
x2t x1u x2u x1v x2v x1w x2w
using *distinct-mset-dom*[*of* ‹*get-clauses-wl S*›]
by (*auto 5 2 simp*: *state-wl-l-def all-init-lits-of-wl-def init-clss-l-fmdrop-if strengthen-clause-pre-def*
Watched-Literals-List-Inprocessing.strengthen-clause-pre-def image-mset-remove-add-mset
split: *subsumption.splits dest*: *in-diffD dest*: *multi-member-split*)
subgoal by (*auto simp*: *state-wl-l-def split*: *subsumption.splits*)
subgoal by (*auto simp*: *state-wl-l-def length-remove1 no-lost-clause-in-WL-def split*: *subsumption.splits*)
subgoal by (*auto simp*: *state-wl-l-def length-remove1 no-lost-clause-in-WL-def split*: *subsumption.splits*)
subgoal using *H*
apply (*cases* ‹*irred* (*get-clauses-wl S*) *C*›)
by (*auto simp*: *state-wl-l-def no-lost-clause-in-WL-def all-init-lits-of-wl-def*
all-lits-of-mm-add-mset init-clss-l-mapsto-upd all-lits-of-mm-union init-clss-l-mapsto-upd-irrel
dest!: *all-lits-of-m-diffD split*: *subsumption.splits*)
done
qed

lemma *case-subsumption-refine*:
‹(*a*,*b*)∈*Id* ⟹
(*is-subsumed a* ⟹ *f*(*subsumed-by a*)≤ ⇓*R* (*f'* (*subsumed-by b*))) ⟹
(*is-strengthened a* ⟹ *g* (*strengthened-on-lit a*) (*strengthened-by a*) ≤ ⇓*R* (*g'* (*strengthened-on-lit a*) (*strengthened-by a*))) ⟹
(*a* = *NONE* ⟹ *h* ≤⇓*R h'*) ⟹
case-subsumption f g h a ≤ ⇓ *R* (*case-subsumption f' g' h' b*)›
by (*cases a*) *auto*

lemma *subsume-or-strengthen-wl-subsume-or-strengthen*:
assumes
‹(*C*, *C'*) ∈ *nat-rel*› **and**
‹(*s*, *s'*) ∈ *Id*› **and**
‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
‹*C* ∈# *dom-m* (*get-clauses-wl S*)› ‹*length* (*get-clauses-wl S* ∝ *C*) > *2*›
shows ‹*subsume-or-strengthen-wl C s S* ≤ ⇓{(*T*, *T'*). (*T*, *T'*) ∈ *state-wl-l None* ∧ *get-watched-wl T* = *get-watched-wl S*}
(*subsume-or-strengthen C' s' S'*)›

using *assms*
unfolding *subsume-or-strengthen-wl-def subsume-or-strengthen-def Let-def*
apply (*refine-vcg strengthen-clause-wl-strengthen-clause case-subsumption-refine*)
subgoal unfolding *subsume-or-strengthen-wl-pre-def* **by** *fast*
subgoal premises *p*
using *assms p*(*32*−) **unfolding** *p*(*1*−*31*)
by (*auto simp*: *state-wl-l-def all-init-lits-of-wl-def all-init-lits-of-l-def*
all-init-atms-st-alt-def get-init-clss-l-def)
subgoal
unfolding *in-pair-collect-simp*
by (*auto simp*: *state-wl-l-def subsume-or-strengthen-pre-def strengthen-clause-pre-def*
intro!: *strengthen-clause-wl-strengthen-clause*[*THEN order-trans*] *ASSERT-leI*
split: *subsumption.splits*)
subgoal
by (*auto simp*: *state-wl-l-def subsume-or-strengthen-pre-def strengthen-clause-pre-def*
intro!: *strengthen-clause-wl-strengthen-clause*[*THEN order-trans*]
split: *subsumption.splits*)
subgoal
by (*auto simp*: *state-wl-l-def subsume-or-strengthen-pre-def strengthen-clause-pre-def*
no-lost-clause-in-WL-def
intro!: *strengthen-clause-wl-strengthen-clause*[*THEN order-trans*]
split: *subsumption.splits*)
done

definition *forward-subsumption-one-wl-pre* :: ‹*nat* ⇒ *nat multiset* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*forward-subsumption-one-wl-pre* = (λ*C cands S*.
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧ *forward-subsumption-one-pre C cands S'* ∧
literals-are-$\mathcal{L}_{in}$*' S*))›

definition *forward-subsumption-one-wl-inv* :: ‹*'v twl-st-wl* ⇒ *nat* ⇒ *nat multiset* ⇒ *nat multiset* × *'v subsumption* ⇒ *bool*› **where**
‹*forward-subsumption-one-wl-inv* = (λ*S C cands* (*i*, *x*).
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *forward-subsumption-one-inv C S' cands* (*i*, *x*)))›

definition *forward-subsumption-one-wl-select* **where**
‹*forward-subsumption-one-wl-select C cands S* = (λ*xs*. *C* ∉# *xs* ∧ *cands* ∩# *xs* = {#} ∧
(∀ *D*∈#*xs*. *D* ∈# *dom-m* (*get-clauses-wl S*) ⟶
(∀ *L*∈ *set* (*get-clauses-wl S* ∝ *D*). *undefined-lit* (*get-trail-wl S*) *L*) ∧
(*length* (*get-clauses-wl S* ∝ *D*) ≤ *length* (*get-clauses-wl S* ∝ *C*))))›

definition *forward-subsumption-one-wl* :: ‹*nat* ⇒ *nat multiset* ⇒ *'v twl-st-wl* ⇒ (*'v twl-st-wl* × *bool*) *nres*› **where**
‹*forward-subsumption-one-wl* = (λ*C cands* S_0 . *do* {
ASSERT (*forward-subsumption-one-wl-pre C cands* S_0);
ys ← *SPEC* (*forward-subsumption-one-wl-select C cands* S_0);
(*xs*, *s*) ←
$WHILE_T$ *forward-subsumption-one-wl-inv* S_0 *C ys* (λ(*xs*, *s*). *xs* ≠ {#} ∧ *s* = *NONE*)
(λ(*xs*, *s*). *do* {
C' ← *SPEC* (λ*C'*. *C'* ∈# *xs*);
if C' ∉# *dom-m* (*get-clauses-wl* S_0)
then RETURN (*remove1-mset C' xs*, *s*)
else do {
s ← *subsume-clauses-match C' C* (*get-clauses-wl* S_0);

```
    RETURN (remove1-mset C' xs, s)
   }
  })
  (ys, NONE);
 S ← subsume-or-strengthen-wl C s S₀;
 ASSERT (literals-are-ℒ_in' S ∧ set-mset (all-init-lits-of-wl S) = set-mset (all-init-lits-of-wl S₀));
 RETURN (S, s ≠ NONE)
 })⟩


lemma forward-subsumption-one-wl:
 assumes
  SS': ⟨(S, S') ∈ state-wl-l None⟩ and ⟨no-lost-clause-in-WL S⟩ and ⟨literals-are-ℒ_in' S⟩ and
  cands: ⟨(cands, cands') ∈ Id⟩
 shows
  ⟨forward-subsumption-one-wl C cands S ≤ ⇓({(T, T').
  (T, T') ∈ state-wl-l None ∧ get-watched-wl T = get-watched-wl S} ×_f bool-rel)
  (forward-subsumption-one C cands' S')⟩
proof −
 have [refine0]: ⟨(xs, ys) ∈ Id ⟹ ((xs, NONE), (ys, NONE)) ∈ {(a,b). a = b ∧ b ⊆# ys} ×_f Id⟩ (is
⟨- ⟹ - ∈ ?loop⟩) for xs ys :: ⟨nat multiset⟩
  by auto
 have subsume-clauses-match-pre: ⟨subsume-clauses-match-pre C' C (get-clauses-wl S)⟩
  if
   pre: ⟨forward-subsumption-one-pre C cands' S'⟩ and
   ⟨forward-subsumption-one-wl-pre C cands S⟩ and
   xs-ys: ⟨(ys, xs) ∈ Id⟩ and
   ys: ⟨ys ∈ Collect (forward-subsumption-one-wl-select C cands S)⟩ and
   xs: ⟨xs ∈ Collect (forward-subsumption-one-select C cands' S')⟩ and
   xx': ⟨(x, x') ∈ ?loop xs⟩ and
   x: ⟨case x of (xs, s) ⇒ xs ≠ {#} ∧ s = NONE⟩ and
   x': ⟨case x' of (xs, s) ⇒ xs ≠ {#} ∧ s = NONE⟩ and
   ⟨forward-subsumption-one-wl-inv S C xs0 x⟩ and
   inv: ⟨forward-subsumption-one-inv C S' ys0 x'⟩ and
   st: ⟨x' = (x1, x2)⟩ ⟨x = (x1a, x2a)⟩ and
   C': ⟨(C', C'a) ∈ nat-rel⟩
    ⟨C' ∈ {C'. C' ∈# x1a}⟩
    ⟨C'a ∈ {C'. C' ∈# x1}⟩
    ⟨¬ C' ∉# dom-m (get-clauses-wl S)⟩
    ⟨¬ C'a ∉# dom-m (get-clauses-l S')⟩
  for C C' C'a x x' x1 x2 x1a x2a xs ys ys0 xs0
 proof −
   obtain S'' where
    inv: ⟨forward-subsumption-one-inv C S' ys0 (x1a, x2a)⟩ and
    ⟨C ≠ 0⟩ and
    S'S'': ⟨(S', S'') ∈ twl-st-l None⟩ and
    struct: ⟨twl-struct-invs S''⟩ and
    list-invs: ⟨twl-list-invs S'⟩ and
    C-dom: ⟨C ∈# dom-m (get-clauses-wl S)⟩
    using inv C' pre assms xx' unfolding forward-subsumption-one-wl-inv-def prod.simps
      forward-subsumption-one-wl-pre-def forward-subsumption-one-pre-def st
    by auto
   have ⟨length (get-clauses-wl S ∝ C') ≤ length (get-clauses-wl S ∝ C)⟩ and
    k-dom: ⟨C' ∈# dom-m (get-clauses-wl S)⟩
    using xs-ys xx' xs C' SS' unfolding C' st forward-subsumption-one-select-def
    by (auto)
```

moreover have ‹*distinct* (*get-clauses-wl S* ∝ *C*)› ‹*distinct* (*get-clauses-wl S* ∝ *C′*)›
‹¬*tautology* (*mset* (*get-clauses-wl S* ∝ *C*))›
‹¬*tautology* (*mset* (*get-clauses-wl S* ∝ *C′*))›
using *C′ SS′ S′S″ struct k-dom list-invs C-dom* **unfolding** *C′ st twl-list-invs-def*
by (*auto simp*: *subsume-or-strengthen-pre-def state-wl-l-def twl-struct-invs-def*
pcdcl-all-struct-invs-def cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-alt-def ran-m-def add-mset-eq-add-mset*
split: *subsumption.splits*
dest!: *multi-member-split*)
ultimately show *?thesis*
using *C-dom C′ SS′* **unfolding** *subsume-clauses-match-pre-def*
by *auto*
qed
show *?thesis*
unfolding *forward-subsumption-one-wl-def forward-subsumption-one-def*
apply (*refine-vcg*
subsume-clauses-match[*unfolded Down-id-eq*])
subgoal using *assms* **unfolding** *forward-subsumption-one-wl-pre-def* **by** *fast*
subgoal using *assms* **unfolding** *forward-subsumption-one-wl-select-def forward-subsumption-one-select-def*
by *auto*
subgoal for *ys xs x x′*
using *assms* **unfolding** *forward-subsumption-one-wl-inv-def case-prod-beta* **by** (*cases x*; *cases x′*)
fastforce
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
apply (*rule subsume-clauses-match-pre*; *assumption?*)
subgoal using *assms* **by** *auto*
subgoal by *auto*
apply (*rule subsume-or-strengthen-wl-subsume-or-strengthen*)
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **unfolding** *forward-subsumption-one-inv-def subsume-or-strengthen-pre-def*
prod.simps
apply − **apply** *normalize-goal+*
by (*simp add*: *try-to-subsume-def*
forward-subsumption-one-inv-def subsume-or-strengthen-pre-def split: *subsumption.splits*)
subgoal unfolding *forward-subsumption-one-wl-pre-def forward-subsumption-one-pre-def*
by *normalize-goal+ auto*
subgoal using *assms*(*3*) *SS′* **by** (*force simp*: *literals-are-*$\mathcal{L}_{in}$*′-def blits-in-*$\mathcal{L}_{in}$*′-def*
watched-by-alt-def)
subgoal using *SS′* **by** *auto*
subgoal by *auto*
done
qed

definition *try-to-forward-subsume-wl-pre* :: ‹*nat* ⇒ *nat multiset* ⇒ *′v twl-st-wl* ⇒ *bool*› **where**
‹*try-to-forward-subsume-wl-pre C cands S* = (∃ *T*. (*S*,*T*)∈*state-wl-l None* ∧ *try-to-forward-subsume-pre C cands T* ∧ *literals-are-*$\mathcal{L}_{in}$*′ S* ∧ *no-lost-clause-in-WL S*)›

definition *try-to-forward-subsume-wl-inv* :: ‹-› **where**
‹*try-to-forward-subsume-wl-inv S cands C* = (λ(*i*,*break*, *T*).
(∃ *S′ T′*. (*S*,*S′*)∈*state-wl-l None* ∧ (*T*,*T′*)∈*state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧

try-to-forward-subsume-inv S′ cands C (i, break, T′) ∧ get-watched-wl T = get-watched-wl S ∧ literals-are-$\mathcal{L}_{in}$′ S)) ›

definition *try-to-forward-subsume-wl :: ‹nat ⇒ nat multiset ⇒ 'v twl-st-wl ⇒ 'v twl-st-wl nres›* **where**
‹try-to-forward-subsume-wl C cands S = do {
ASSERT (try-to-forward-subsume-wl-pre C cands S);
n ← RES {-::nat. True};
ebreak ← RES {-::bool. True};
(-, -, S) ← WHILE$_T$ try-to-forward-subsume-wl-inv S cands C
(λ(i, break, S). ¬break ∧ i < n)
(λ(i, break, S). do {
(S, subs) ← forward-subsumption-one-wl C cands S;
ebreak ← RES {-::bool. True};
RETURN (i+1, subs ∨ ebreak, S)
})
(0, ebreak, S);
RETURN S
}
›

lemma *cdcl-twl-inprocessing-l-dom-get-clauses-mono: ‹cdcl-twl-inprocessing-l S T ⟹*
set-mset (dom-m (get-clauses-l T)) ⊆ set-mset (dom-m (get-clauses-l S))›
by *(cases rule: cdcl-twl-inprocessing-l.cases)*
(auto simp: cdcl-twl-unitres-l.simps cdcl-twl-unitres-true-l.simps
cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps
cdcl-twl-pure-remove-l.simps dest: in-diffD)

lemma *rtranclp-cdcl-twl-inprocessing-l-dom-get-clauses-mono:*
*‹cdcl-twl-inprocessing-l** S T ⟹*
set-mset (dom-m (get-clauses-l T)) ⊆ set-mset (dom-m (get-clauses-l S))›
by *(induction rule: rtranclp-induct) (auto dest: cdcl-twl-inprocessing-l-dom-get-clauses-mono)*

lemma *try-to-forward-subsume-wl-inv-no-lost-clause-in-WLD:*
assumes
‹try-to-forward-subsume-wl-inv S cands C (i, break, T)›
shows *‹no-lost-clause-in-WL T›* **and** *‹literals-are-$\mathcal{L}_{in}$′ T›*
proof −
obtain *S′ T′* **where**
SS′: ‹(S, S′) ∈ state-wl-l None› **and**
TT′: ‹(T, T′) ∈ state-wl-l None› **and**
lost: ‹no-lost-clause-in-WL S›
‹try-to-forward-subsume-inv S′ cands C (i, break, T′)› **and**
*S′T′: ‹cdcl-twl-inprocessing-l** S′ T′›* **and**
w: ‹get-watched-wl T = get-watched-wl S› **and**
lits: ‹literals-are-$\mathcal{L}_{in}$′ S›
using *assms* **unfolding** *try-to-forward-subsume-wl-inv-def prod.simps*
try-to-forward-subsume-inv-def
by *blast*

have *‹set-mset (dom-m (get-clauses-wl T)) ⊆ set-mset (dom-m (get-clauses-wl S))›*
using *SS′ TT′ rtranclp-cdcl-twl-inprocessing-l-dom-get-clauses-mono[OF S′T′]* **by** *auto*
moreover have *eq: ‹set-mset (all-init-lits-of-l S′) = set-mset (all-init-lits-of-l T′)›*
using *S′T′ rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l* **by** *blast*
ultimately show *‹no-lost-clause-in-WL T›*
using *lost SS′ TT′ w* **unfolding** *no-lost-clause-in-WL-def*
by *(auto simp: watched-by-alt-def)*

show ‹*literals-are-*$\mathcal{L}_{in}$*′ T*›
using *eq SS′ TT′ rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l*[*OF S′T′*] *w lits*
unfolding *literals-are-*$\mathcal{L}_{in}$*′-def*
by (*auto simp*: *blits-in-*$\mathcal{L}_{in}$*′-def watched-by-alt-def*)
qed

lemma *try-to-forward-subsume-wl*:
assumes
SS′: ‹(*S*, *S′*) ∈ *state-wl-l None*› **and**
‹*no-lost-clause-in-WL S*› **and**
‹(*C*,*C′*)∈*nat-rel*› **and**
‹*literals-are-*$\mathcal{L}_{in}$*′ S*› **and**
‹(*cands*, *cands′*) ∈*Id*›
shows
‹*try-to-forward-subsume-wl C cands S* ≤ ⇓({(*T*, *T′*).
(*T*, *T′*) ∈ *state-wl-l None* ∧ *get-watched-wl T* = *get-watched-wl S*})
(*try-to-forward-subsume C′ cands′ S′*)›
proof −
have *CC′*: ‹*C′* = *C*›
using *assms* **by** *auto*
have [*refine0*]: ‹(*xs*, *ys*) ∈ *Id* ⟹ (*ebreak*, *ebreak′*) ∈ *Id* ⟹
((*xs*, *ebreak*, *S*), (*ys*, *ebreak′*, *S′*)) ∈ *Id* ×$_r$ *bool-rel* ×$_r$ {(*U*,*V*). (*U*,*V*)∈ *state-wl-l None* ∧ *get-watched-wl U* = *get-watched-wl S*}› (**is** ‹- ⟹ - ⟹ - ∈ *?loop*›) **for** *xs ys* :: ‹*nat*› **and** *ebreak ebreak′*
using *assms* **by** *auto*
show *?thesis*
unfolding *try-to-forward-subsume-wl-def try-to-forward-subsume-def CC′*
apply (*refine-vcg simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st*
forward-subsumption-one-wl)
subgoal using *assms* **unfolding** *try-to-forward-subsume-wl-pre-def* **by** *fast*
subgoal using *assms* **by** *auto*
subgoal for *xs xsa ebreak ebreak′ x x′*
using *assms* **unfolding** *try-to-forward-subsume-wl-inv-def case-prod-beta prod-rel-fst-snd-iff*
by (*rule-tac x*=*S′* **in** *exI*, *rule-tac x*=‹*snd* (*snd x′*)› **in** *exI*) (*auto*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto dest*: *try-to-forward-subsume-wl-inv-no-lost-clause-in-WLD*)
subgoal by (*auto dest*: *try-to-forward-subsume-wl-inv-no-lost-clause-in-WLD*)
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *forward-subsumption-all-wl-pre* :: ‹*′v twl-st-wl* ⇒ *bool*› **where**
‹*forward-subsumption-all-wl-pre S* =
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *forward-subsumption-all-pre T* ∧ *literals-are-*$\mathcal{L}_{in}$*′ S*)›

definition *forward-subsumption-all-wl-inv* :: ‹*′v twl-st-wl* ⇒ *nat multiset* ⇒ *nat multiset* × *′v twl-st-wl* ⇒ *bool*› **where**
‹*forward-subsumption-all-wl-inv* = (λ*S cands* (*xs*, *s*).
(∃ *T s′*. (*S*, *T*) ∈ *state-wl-l None* ∧ (*s*, *s′*) ∈ *state-wl-l None* ∧ *forward-subsumption-all-inv T* (*xs*, *s′*)
∧
no-lost-clause-in-WL S ∧ *get-watched-wl s* = *get-watched-wl S* ∧ *literals-are-*$\mathcal{L}_{in}$*′ S*))›

definition *forward-subsumption-wl-all-cands* **where**

```
  ‹forward-subsumption-wl-all-cands S xs ⟷ xs ⊆# dom-m (get-clauses-wl S) ∧
  (∀ C∈#xs. (∀ L∈set (get-clauses-wl S ∝ C). undefined-lit (get-trail-wl S) L) ∧
  length (get-clauses-wl S ∝ C) > 2)›

definition forward-subsumption-all-wl :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› where
  ‹forward-subsumption-all-wl = (λS. do {
  ASSERT (forward-subsumption-all-wl-pre S);
  xs ← SPEC (forward-subsumption-wl-all-cands S);
  (xs, S) ←
    WHILE_T forward-subsumption-all-wl-inv S xs  (λ(xs, S). xs ≠ {#} ∧ get-conflict-wl S = None)
    (λ(xs, S). do {
      C ← SPEC (λC. C ∈# xs);
      T ← try-to-forward-subsume-wl C xs S;
      ASSERT (∀ D∈#remove1-mset C xs. get-clauses-wl T ∝ D = get-clauses-wl S ∝ D);
      RETURN (remove1-mset C xs, T)
    })
    (xs, S);
  RETURN S
  }
)›


definition forward-subsume-wl-needed :: ‹'v twl-st-wl ⇒ bool nres› where
  ‹forward-subsume-wl-needed S = SPEC (λ-. True)›

definition forward-subsume-wl :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› where
  ‹forward-subsume-wl S = do {
    ASSERT (forward-subsumption-all-wl-pre S);
    b ← forward-subsume-wl-needed S;
    if b then forward-subsumption-all-wl S else RETURN S
}›

lemma
  assumes ‹forward-subsumption-all-wl-inv S cands (xs, T)›
  shows
    forward-subsumption-all-wl-inv-no-lost-clause-in-WLD: ‹no-lost-clause-in-WL T› and
    forward-subsumption-all-wl-inv-literals-are-ℒ_in'D: ‹literals-are-ℒ_in' T›
proof −
  obtain S' T' where
    SS': ‹(S, S') ∈ state-wl-l None› and
    TT': ‹(T, T') ∈ state-wl-l None› and
    lost: ‹no-lost-clause-in-WL S› and
    S'T': ‹cdcl-twl-inprocessing-l** S' T'› and
    w: ‹get-watched-wl T = get-watched-wl S› and
    lits: ‹literals-are-ℒ_in' S›
    using assms unfolding forward-subsumption-all-wl-inv-def prod.simps
      forward-subsumption-all-inv-def
    by blast

  have ‹set-mset (dom-m (get-clauses-wl T)) ⊆ set-mset (dom-m (get-clauses-wl S))›
    using SS' TT' rtranclp-cdcl-twl-inprocessing-l-dom-get-clauses-mono[OF S'T'] by auto
  moreover have eq: ‹set-mset (all-init-lits-of-l S') = set-mset (all-init-lits-of-l T')›
   using S'T' rtranclp-cdcl-twl-inprocessing-l-all-init-lits-of-l by blast
  ultimately show ‹no-lost-clause-in-WL T›
    using lost SS' TT' w unfolding no-lost-clause-in-WL-def
    by (auto simp: watched-by-alt-def)
```

show ‹*literals-are-$\mathcal{L}_{in}$′ T*›
using *eq SS′ TT′ rtranclp-cdcl-twl-inprocessing-l-all-learned-lits-of-l*[*OF S′T′*] *w lits*
unfolding *literals-are-$\mathcal{L}_{in}$′-def*
by (*auto simp*: *blits-in-$\mathcal{L}_{in}$′-def watched-by-alt-def*)
qed

lemma *forward-subsumption-all-wl-inv-alt-def*:
‹*forward-subsumption-all-wl-inv* = (λ*S cands* (*xs*, *s*).
(∃ *T s′*. (*S*, *T*) ∈ *state-wl-l None* ∧ (*s*, *s′*) ∈ *state-wl-l None* ∧ *forward-subsumption-all-inv T* (*xs*, *s′*)
∧
no-lost-clause-in-WL S ∧ *get-watched-wl s* = *get-watched-wl S* ∧ *literals-are-$\mathcal{L}_{in}$′ S* ∧ *literals-are-$\mathcal{L}_{in}$′ s*))›
using *forward-subsumption-all-wl-inv-literals-are-$\mathcal{L}_{in}$′D* **unfolding** *forward-subsumption-all-wl-inv-def*
by *fast*

lemma *forward-subsumption-all-wl*:
assumes
SS′: ‹(*S*, *S′*) ∈ *state-wl-l None*› **and**
lost: ‹*correct-watching′-leaking-bin S*› **and**
lits: ‹*literals-are-$\mathcal{L}_{in}$′ S*›
shows
‹*forward-subsumption-all-wl S* ≤ ⇓({(*T*, *T′*).
(*T*, *T′*) ∈ *state-wl-l None* ∧ *get-watched-wl T* = *get-watched-wl S* ∧ *no-lost-clause-in-WL T* ∧
literals-are-$\mathcal{L}_{in}$′ T})
(*forward-subsumption-all S′*)›
proof −

have *lost*: ‹*no-lost-clause-in-WL S*› **if** *pre*: ‹*forward-subsumption-all-pre S′*›
proof −
obtain *S″* **where**
S′S″: ‹(*S′*, *S″*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs S″*› **and**
‹*twl-list-invs S′*›
‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of S″*)› **and**
‹*set* (*get-all-mark-of-propagated* (*get-trail-l S′*)) ⊆ {*0*}› **and**
‹*clauses-to-update-l S′* = {#}› **and**
‹*count-decided* (*get-trail-l S′*) = *0*›
using *pre* **unfolding** *forward-subsumption-all-pre-def* **by** *fast*
have ‹*correct-watching′-nobin S*›
using *assms*(*2*) **by** (*cases S*) (*auto simp*: *correct-watching′-leaking-bin.simps*
correct-watching′-nobin.simps)
then show *?thesis*
using *correct-watching′-nobin-clauses-pointed-to0*[*OF assms*(*1*) - *assms*(*3*) *S′S″ struct-invs*]
by *fast*
qed
then have [*refine0*]: ‹(*xs*, *ys*) ∈ *Id* ⟹ *forward-subsumption-all-pre S′* ⟹ *forward-subsumption-all-inv*
S′ (*ys*, *S′*) ⟹
forward-subsumption-all-wl-inv S xs (*xs*, *S*)› **for** *xs ys* :: ‹*nat multiset*›
using *assms* **unfolding** *forward-subsumption-all-wl-inv-def case-prod-beta prod-rel-fst-snd-iff*
by (*rule-tac x*=*S′* **in** *exI*, *rule-tac x*=‹*S′*› **in** *exI*) *auto*
then have [*refine0*]: ‹(*xs*, *ys*) ∈ *Id* ⟹
((*xs*, *S*), (*ys*, *S′*)) ∈ *Id* ×$_f$ {(*U*,*V*). (*U*,*V*)∈ *state-wl-l None* ∧ *get-watched-wl U* = *get-watched-wl S*}› (**is** ‹- ⟹ - ∈ *?loop*›) **for** *xs ys* :: ‹*nat multiset*›
using *assms* **by** *auto*

show *?thesis*

unfolding *forward-subsumption-all-wl-def forward-subsumption-all-def*
apply (*refine-vcg simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st*
forward-subsumption-one-wl try-to-forward-subsume-wl WHILEIT-refine-with-all-loopinvariants)
subgoal using *assms* **unfolding** *forward-subsumption-all-wl-pre-def* **by** *fast*
subgoal using *assms* **unfolding** *forward-subsumption-all-cands-def forward-subsumption-wl-all-cands-def*
by *auto*
subgoal for *xs xsa x x'*
using *assms lost* **unfolding** *forward-subsumption-all-wl-inv-def case-prod-beta prod-rel-fst-snd-iff*
by (*rule-tac x=S' in exI*, *rule-tac x=‹snd x'› in exI*; *auto*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto dest*: *forward-subsumption-all-wl-inv-no-lost-clause-in-WLD*
forward-subsumption-all-wl-inv-literals-are-$\mathcal{L}_{in}$'D)
subgoal by (*auto dest*: *forward-subsumption-all-wl-inv-no-lost-clause-in-WLD*
forward-subsumption-all-wl-inv-literals-are-$\mathcal{L}_{in}$'D)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto dest*: *forward-subsumption-all-wl-inv-no-lost-clause-in-WLD*
forward-subsumption-all-wl-inv-literals-are-$\mathcal{L}_{in}$'D)
done
qed

lemma *forward-subsume-wl*:
assumes
SS': ‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
lost: ‹*correct-watching'-leaking-bin S*› **and**
lits: ‹*literals-are-$\mathcal{L}_{in}$' S*›
shows
‹*forward-subsume-wl S* ≤ ⇓({(*T*, *T'*).
(*T*, *T'*) ∈ *state-wl-l None* ∧ *get-watched-wl T* = *get-watched-wl S* ∧ *no-lost-clause-in-WL T* ∧
literals-are-$\mathcal{L}_{in}$' T})
(*forward-subsume-l S'*)›
proof −
have *lost*: ‹*no-lost-clause-in-WL S*› **if** *pre*: ‹*forward-subsumption-all-pre S'*›
proof −
obtain *S''* **where**
S'S'': ‹(*S'*, *S''*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs S''*› **and**
‹*twl-list-invs S'*›
‹*cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init* (*state$_W$-of S''*)› **and**
‹*set* (*get-all-mark-of-propagated* (*get-trail-l S'*)) ⊆ {*0*}› **and**
‹*clauses-to-update-l S'* = {#}› **and**
‹*count-decided* (*get-trail-l S'*) = *0*›
using *pre* **unfolding** *forward-subsumption-all-pre-def* **by** *fast*
have ‹*correct-watching'-nobin S*›
using *assms*(*2*) **by** (*cases S*) (*auto simp*: *correct-watching'-leaking-bin.simps*
correct-watching'-nobin.simps)
then show *?thesis*
using *correct-watching'-nobin-clauses-pointed-to0*[*OF assms*(*1*) - *assms*(*3*) *S'S''* *struct-invs*]
by *fast*
qed
show *?thesis*
unfolding *forward-subsume-wl-def forward-subsume-l-def*
forward-subsumption-needed-l-def forward-subsume-wl-needed-def

apply (*refine-vcg forward-subsumption-all-wl*)
subgoal using *assms* **unfolding** *forward-subsumption-all-wl-pre-def* **by** *fast*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms lost* **by** *simp*
done
qed

lemma *cdcl-twl-inprocessing-l-dom-get-clauses-l-mono*:
‹*cdcl-twl-inprocessing-l S T* ⟹ *dom-m* (*get-clauses-l T*) ⊆# *dom-m* (*get-clauses-l S*)›
by (*auto simp*: *cdcl-twl-inprocessing-l.simps cdcl-twl-unitres-true-l.simps cdcl-twl-unitres-l.simps cdcl-twl-subsumed-l.simps cdcl-twl-subresolution-l.simps add-mset-eq-add-mset cdcl-twl-pure-remove-l.simps dest*!: *multi-member-split*)

lemma *rtranclp-cdcl-twl-inprocessing-l-dom-get-clauses-l-mono*:
‹*cdcl-twl-inprocessing-l*** *S T* ⟹ *dom-m* (*get-clauses-l T*) ⊆# *dom-m* (*get-clauses-l S*)›
by (*induction rule*: *rtranclp-induct*) (*auto dest*: *cdcl-twl-inprocessing-l-dom-get-clauses-l-mono*)

6.5.10 Pure literal deletion

definition *propagate-pure-wl-pre*:: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*propagate-pure-wl-pre L S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *propagate-pure-l-pre L S'* ∧ *no-lost-clause-in-WL S* ∧
literals-are-$\mathcal{L}_{in}$*' S*)›

definition *propagate-pure-bt-wl* :: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*propagate-pure-bt-wl* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*). *do* {
ASSERT(*propagate-pure-wl-pre L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*));
M ← *cons-trail-propagate-l L 0 M*;
RETURN (*M*, *N*, *D*, *NE*, *UE*, *add-mset* {#*L*#} *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *add-mset* (−*L*) *Q*, *WS*)})›

lemma *propagate-pure-bt-wl-propagate-pure-bt-l*:
assumes ‹(*S*,*S'*)∈ *state-wl-l None*› **and** ‹*no-lost-clause-in-WL S*› ‹*literals-are-*$\mathcal{L}_{in}$*' S*› ‹(*L*,*L'*)∈ *Id*›
shows
‹*propagate-pure-bt-wl L S* ≤⇓{(*S*,*T*). *no-lost-clause-in-WL S* ∧ (*S*,*T*)∈ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}$*' S*} (*propagate-pure-bt-l L' S'*)›
proof −
have *K*: ‹*cons-trail-propagate-l L 0 a* = *cons-trail-propagate-l L 0 b* ⟹
cons-trail-propagate-l L 0 a ≤⇓{(*x*,*y*). *x* = *y* ∧ *y* = *Propagated L 0* # *a*} (*cons-trail-propagate-l L' 0 b*)› **for** *a b*
using *assms* **unfolding** *cons-trail-propagate-l-def*
by *refine-rcg auto*
show *?thesis*
unfolding *propagate-pure-bt-wl-def propagate-pure-bt-l-def*
apply *refine-rcg*
subgoal using *assms* **unfolding** *propagate-pure-wl-pre-def* **apply** − **by** (*rule exI*[*of* - *S'*]) *fast*
apply (*rule K*)
subgoal using *assms* **by** (*auto simp*: *state-wl-l-def no-lost-clause-in-WL-def*)
subgoal
using *assms*
by (*auto simp*: *state-wl-l-def propagate-pure-l-pre-def in-all-lits-of-mm-uminusD*

all-init-lits-of-wl-def all-init-lits-of-l-def get-init-clss-l-def
simp: *no-lost-clause-in-WL-def*
simp: *literals-are-$\mathcal{L}_{in}$'-def all-lits-of-mm-add-mset all-lits-of-m-add-mset*
all-learned-lits-of-wl-def blits-in-$\mathcal{L}_{in}$'-def)
done
qed

definition *pure-literal-deletion-wl-pre* **where**
‹*pure-literal-deletion-wl-pre S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *pure-literal-deletion-pre S'* ∧
no-lost-clause-in-WL S ∧ *literals-are-$\mathcal{L}_{in}$' S*)›

definition *pure-literal-deletion-candidates-wl* **where**
‹*pure-literal-deletion-candidates-wl S* = *SPEC* (λ*Ls*. *set-mset Ls* ⊆ *set-mset* (*all-init-atms-st S*))›

definition *pure-literal-deletion-wl-inv* **where**
‹*pure-literal-deletion-wl-inv S xs0* = (λ(*T*, *xs*).
∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧ *pure-literal-deletion-l-inv S' xs0* (*T'*, *xs*) ∧
no-lost-clause-in-WL T ∧ *literals-are-$\mathcal{L}_{in}$' T*)›

definition *pure-literal-deletion-wl* :: ‹(*'v literal* ⇒ *bool*) ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*pure-literal-deletion-wl occs S* = *do* {
ASSERT (*pure-literal-deletion-wl-pre S*);
let As = ⋃(*set-mset* ‘ *set-mset* (*mset* ‘# *get-init-clss-wl S*));
xs ← *pure-literal-deletion-candidates-wl S*;
(*S*, *xs*) ← *WHILE*$_T^{\textit{pure-literal-deletion-wl-inv } S \textit{ xs}}$ (λ(*S*, *xs*). *xs* ≠ {#})
(λ(*S*, *xs*). *do* {
L ← *SPEC* (λ*L*. *L* ∈# *xs*);
let A = (*if occs* (*Pos L*) ∧ ¬*occs* (*Neg L*) *then Pos L else Neg L*);
if ¬*occs* (−*A*) ∧ *undefined-lit* (*get-trail-wl S*) *A*
then do {*S* ← *propagate-pure-bt-wl A S*;
RETURN (*S*, *remove1-mset L xs*)}
else RETURN (*S*, *remove1-mset L xs*)
})
(*S*, *xs*);
RETURN S
}›

lemma *pure-literal-deletion-wl-pure-literal-deletion-l*:
assumes ‹(*S*,*S'*)∈ *state-wl-l None*› **and** ‹*no-lost-clause-in-WL S*› ‹*literals-are-$\mathcal{L}_{in}$' S*› ‹(*occs*, *occs'*) ∈ *Id*›
shows
‹*pure-literal-deletion-wl occs S* ≤⇓{(*S*,*T*). *no-lost-clause-in-WL S* ∧ (*S*,*T*)∈ *state-wl-l None* ∧ *literals-are-$\mathcal{L}_{in}$' S*} (*pure-literal-deletion-l2 occs' S'*)› (**is** ‹- ≤⇓ *?A* -›)
proof −
have [*refine0*]: ‹(*xs*, *ys*) ∈ *Id* ⟹ ((*S*, *xs*), (*S'*, *ys*)) ∈ *?A* ×$_f$ *Id*› **for** *xs ys*
by (*use assms* **in** *auto*)
have [*refine0*]: ‹*pure-literal-deletion-candidates-wl S* ≤ ⇓ *Id* (*pure-literal-deletion-candidates S'*)›
using *assms* **by** (*auto simp*: *pure-literal-deletion-candidates-wl-def pure-literal-deletion-candidates-def*
all-init-atms-st-def all-init-atms-alt-def)
show *?thesis*
unfolding *pure-literal-deletion-wl-def pure-literal-deletion-l2-def*
apply (*refine-rcg propagate-pure-bt-wl-propagate-pure-bt-l*)
subgoal using *assms* **unfolding** *pure-literal-deletion-wl-pre-def* **by** *blast*
subgoal for *xs xsa x x'*

using *assms* **unfolding** *pure-literal-deletion-wl-inv-def case-prod-beta*
by (*rule-tac x=S′* **in** *exI*, *rule-tac x=‹fst x′›* **in** *exI*, *case-tac x*, *case-tac x′*) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal
by (*subst twl-st-wl*, *use assms* **in** *auto*)
subgoal by *auto*
subgoal unfolding *pure-literal-deletion-wl-inv-def* **by** *blast*
subgoal by *simp*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *pure-literal-count-occs-clause-wl-invs* :: ‹*nat* ⇒ *′v twl-st-wl* ⇒ (*′v literal* ⇒ *bool*) ⇒ *nat* × (*′v literal* ⇒ *bool*) ⇒ *bool*› **where**
‹*pure-literal-count-occs-clause-wl-invs C S occs* = (λ(*i*, *occs2*).
∃ *S′*. (*S*,*S′*)∈*state-wl-l None* ∧ *pure-literal-count-occs-l-clause-invs C S′ occs* (*i*, *occs2*))›

definition *pure-literal-count-occs-clause-wl-pre* :: ‹*nat* ⇒ *′v twl-st-wl* ⇒ - ⇒ *bool*› **where**
‹*pure-literal-count-occs-clause-wl-pre C S occs* =
(∃ *S′*. (*S*,*S′*)∈*state-wl-l None* ∧ *pure-literal-count-occs-l-clause-pre C S′ occs*)›

definition *pure-literal-count-occs-clause-wl* :: ‹*nat* ⇒ *′v twl-st-wl* ⇒ -› **where**
‹*pure-literal-count-occs-clause-wl C S occs* = *do* {
ASSERT (*pure-literal-count-occs-clause-wl-pre C S occs*);
(*i*, *occs*) ← *WHILE*$_T^{\textit{pure-literal-count-occs-clause-wl-invs C S occs}}$ (λ(*i*, *occs*). *i* < *length* (*get-clauses-wl S* ∝ *C*))
(λ(*i*, *occs*). *do* {
let L = *get-clauses-wl S* ∝ *C* ! *i*;
let occs = *occs* (*L* := *True*);
RETURN (*i*+*1*, *occs*)
})
(*0*, *occs*);
RETURN occs
}›

lemma *pure-literal-count-occs-clause-wl-pure-literal-count-occs-l-clause*:
assumes ‹(*S*, *S′*) ∈ *state-wl-l None*› ‹(*C*,*C′*)∈*nat-rel*› ‹(*occs*, *occs′*) ∈ *Id*›
shows ‹*pure-literal-count-occs-clause-wl C S occs* ≤ ⇓*Id* (*pure-literal-count-occs-l-clause C′ S′ occs′*)›
proof −
have [*refine0*]: ‹((*0*, *occs*), *0*, *occs′*) ∈ *nat-rel* ×$_f$ *Id*›
using *assms* **by** *auto*
show *?thesis*
unfolding *pure-literal-count-occs-clause-wl-def pure-literal-count-occs-l-clause-def*
apply (*refine-vcg*)
subgoal using *assms* **unfolding** *pure-literal-count-occs-clause-wl-pre-def* **by** *fast*
subgoal using *assms* **unfolding** *pure-literal-count-occs-clause-wl-invs-def case-prod-beta prod.collapse*
by *fastforce*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
done

qed

definition *pure-literal-count-occs-wl-pre* :: ‹-› **where**
‹*pure-literal-count-occs-wl-pre S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL S ∧ *literals-are-*$\mathcal{L}_{in}$*' S* ∧ *pure-literal-count-occs-l-pre S'*)›

definition *pure-literal-count-occs-wl-inv* :: ‹-› **where**
‹*pure-literal-count-occs-wl-inv S T* ⟷
(∃ *S' T'*. (*S*,*S'*)∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧ *cdcl-twl-inprocessing-l S' T'*)›

definition *pure-literal-count-occs-wl* :: ‹*'v twl-st-wl* ⇒ -› **where**
‹*pure-literal-count-occs-wl S* = *do* {
ASSERT (*pure-literal-count-occs-wl-pre S*);
xs ← *SPEC* (λ*xs*. *distinct-mset xs* ∧ (∀ *C*∈#*dom-m* (*get-clauses-wl S*). *irred* (*get-clauses-wl S*) *C* ⟶ *C* ∈# *xs*));
abort ← *RES* (*UNIV* :: *bool set*);
let occs = (λ-. *False*);
(-, *occs*, *abort*) ← *WHILE*$_T$(λ(*A*, *occs*, *abort*). *A* ≠ {#} ∧ ¬*abort*)
(λ(*A*, *occs*, *abort*). *do* {
ASSERT (*A* ≠ {#});
C ← *SPEC* (λ*C*. *C* ∈# *A*);
if (*C* ∈# *dom-m* (*get-clauses-wl S*) ∧ *irred* (*get-clauses-wl S*) *C*) *then do* {
occs ← *pure-literal-count-occs-clause-wl C S occs*;
abort ← *RES* (*UNIV* :: *bool set*);
RETURN (*remove1-mset C A*, *occs*, *abort*)
} *else RETURN* (*remove1-mset C A*, *occs*, *abort*)
})
(*xs*, *occs*, *abort*);
RETURN (*abort*, *occs*)
}›

lemma *pure-literal-count-occs-wl-pure-literal-count-occs-l*:
assumes
‹(*S*, *S'*) ∈ *state-wl-l None*›
‹*no-lost-clause-in-WL S*›
‹*literals-are-*$\mathcal{L}_{in}$*' S*›
shows ‹*pure-literal-count-occs-wl S* ≤ ⇓*Id* (*pure-literal-count-occs-l S'*)›
proof −
have [*refine0*]: ‹(*xs*, *xsa*)∈ *Id* ⟹ (*abort*, *abort'*) ∈ *bool-rel* ⟹
((*xs*, λ-. *False*, *abort*), *xsa*, λ-. *False*, *abort'*) ∈ *Id* ×$_r$ *Id* ×$_r$ *bool-rel*› **for** *xs xsa abort abort'*
by *auto*
show *?thesis*
unfolding *pure-literal-count-occs-wl-def pure-literal-count-occs-l-def*
apply (*refine-vcg pure-literal-count-occs-clause-wl-pure-literal-count-occs-l-clause*)
subgoal using *assms* **unfolding** *pure-literal-count-occs-wl-pre-def* **by** *fast*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
done
qed

definition *pure-literal-elimination-round-wl-pre* **where**
‹*pure-literal-elimination-round-wl-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *pure-literal-elimination-round-pre T*)›

definition *pure-literal-elimination-round-wl* **where**
‹*pure-literal-elimination-round-wl S* = *do* {
ASSERT (*pure-literal-elimination-round-wl-pre S*);
S ← *simplify-clauses-with-units-st-wl S*;
if get-conflict-wl S = *None*
then do {
(*abort*, *occs*) ← *pure-literal-count-occs-wl S*;
if ¬*abort then pure-literal-deletion-wl occs S*
else RETURN S}
else RETURN S
}›

lemma *pure-literal-elimination-round-wl-pure-literal-elimination-round-l*:
assumes
‹(*S*, *S'*) ∈ *state-wl-l None*›
‹*no-lost-clause-in-WL S*›
‹*literals-are-*$\mathcal{L}_{in}$*' S*›
shows ‹*pure-literal-elimination-round-wl S* ≤
⇓{(*S*,*T*). *no-lost-clause-in-WL S* ∧ (*S*,*T*)∈ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}$*' S*} (*pure-literal-elimination-round S'*)›
proof −
show *?thesis*
unfolding *pure-literal-elimination-round-wl-def pure-literal-elimination-round-def*
apply (*refine-vcg*
simplify-clauses-with-units-st-wl-simplify-clause-with-units-st2
pure-literal-count-occs-wl-pure-literal-count-occs-l
pure-literal-deletion-wl-pure-literal-deletion-l)
subgoal using *assms* **unfolding** *pure-literal-elimination-round-wl-pre-def* **by** *fast*
subgoal using *assms* **by** *fast*
subgoal using *assms* **by** *fast*
subgoal using *assms* **by** *fast*
subgoal by *auto*
subgoal by *auto*
subgoal by *blast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *blast*
subgoal by *auto*
subgoal by *auto*
subgoal by *blast*
subgoal by *blast*
done
qed

definition *pure-literal-elimination-wl-pre* **where**
‹*pure-literal-elimination-wl-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *pure-literal-elimination-l-pre T* ∧ *no-lost-clause-in-WL S* ∧
literals-are-$\mathcal{L}_{in}$*′ S*)›

definition *pure-literal-elimination-wl-inv* **where**
‹*pure-literal-elimination-wl-inv S max-rounds* =
(λ(*U*,*m*,*abort*). ∃ *T U′*. (*S*, *T*) ∈ *state-wl-l None* ∧ (*U*, *U′*) ∈ *state-wl-l None* ∧
no-lost-clause-in-WL U ∧ *literals-are-*$\mathcal{L}_{in}$*′ U* ∧ *pure-literal-elimination-l-inv T max-rounds* (*U′*,*m*,*abort*))›

definition *pure-literal-elimination-wl* :: ‹-› **where**
‹*pure-literal-elimination-wl S* = *do* {
ASSERT (*pure-literal-elimination-wl-pre S*);
max-rounds ← *RES* (*UNIV* :: *nat set*);
(*S*, -, -) ← *WHILE*$_T^{\textit{pure-literal-elimination-wl-inv S max-rounds}}$ (λ(*S*, *m*, *abort*). *m* < *max-rounds* ∧
¬*abort*)
(λ(*S*, *m*, *abort*). *do* {
S ← *pure-literal-elimination-round-wl S*;
abort ← *RES* (*UNIV* :: *bool set*);
RETURN (*S*, *m*+*1*, *abort*)
})
(*S*, *0*, *False*);
RETURN S
}›

lemma *pure-literal-elimination-wl*:
assumes
‹(*S*, *S′*) ∈ *state-wl-l None*›
‹*no-lost-clause-in-WL S*›
‹*literals-are-*$\mathcal{L}_{in}$*′ S*›
shows ‹*pure-literal-elimination-wl S* ≤
⇓{(*S*,*T*). *no-lost-clause-in-WL S* ∧ (*S*,*T*)∈ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}$*′ S*} (*pure-literal-elimination-l S′*)›
proof −
have [*refine0*]: ‹*pure-literal-elimination-l-pre S′* ⟹
((*S*, *0*, *False*), *S′*, *0*, *False*) ∈ {(*S*,*T*). *no-lost-clause-in-WL S* ∧ (*S*,*T*)∈ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}$*′ S*} ×$_r$ *nat-rel* ×$_r$ *bool-rel*›
using *assms* **by** *auto*
show *?thesis*
unfolding *pure-literal-elimination-wl-def pure-literal-elimination-l-def*
apply (*refine-vcg pure-literal-elimination-round-wl-pure-literal-elimination-round-l*)
subgoal using *assms* **unfolding** *pure-literal-elimination-wl-pre-def* **by** *blast*
subgoal for *max-rounds max-roundsa x x′*
using *assms* **unfolding** *pure-literal-elimination-wl-inv-def case-prod-beta prod-rel-fst-snd-iff*
apply −
by (*rule exI*[*of* - *S′*], *rule exI*[*of* - ‹*fst x′*›]) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *pure-literal-eliminate-wl-needed* :: ‹-› **where**
‹*pure-literal-eliminate-wl-needed S* = *SPEC* (λ-. *True*)›

definition *pure-literal-eliminate-wl* :: ‹-› **where**
‹*pure-literal-eliminate-wl S* = *do* {
ASSERT (*pure-literal-elimination-wl-pre S*);
b ← *pure-literal-eliminate-wl-needed S*;
if b then pure-literal-elimination-wl S else RETURN S
}›

lemma *pure-literal-eliminate-wl*:
assumes
‹(*S*, *S'*) ∈ *state-wl-l None*›
‹*no-lost-clause-in-WL S*›
‹*literals-are-*$\mathcal{L}_{in}$' *S*›
shows ‹*pure-literal-eliminate-wl S* ≤
⇓{(*S*,*T*). *no-lost-clause-in-WL S* ∧ (*S*,*T*)∈ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}$' *S*} (*pure-literal-eliminate-l S'*)›
unfolding *pure-literal-eliminate-wl-def pure-literal-eliminate-l-def*
pure-literal-eliminate-wl-needed-def pure-literal-elimination-l-needed-def
apply (*refine-vcg pure-literal-elimination-wl*)
subgoal using *assms* **unfolding** *pure-literal-elimination-wl-pre-def* **by** *blast*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
done

end
theory *Watched-Literals-Initialisation*
imports *Watched-Literals-List*
begin

Chapter 7

Initialisation of Data structure

type-synonym *'v twl-st-init* = ‹*'v twl-st* × *'v clauses*›

fun *get-trail-init* :: ‹*'v twl-st-init* ⇒ (*'v*, *'v clause*) *ann-lit list*› **where**
‹*get-trail-init* ((*M*, -, -, -, -, -, -), -) = *M*›

fun *get-conflict-init* :: ‹*'v twl-st-init* ⇒ *'v cconflict*› **where**
‹*get-conflict-init* ((-, -, -, *D*, -, -, -, -), -) = *D*›

fun *literals-to-update-init* :: ‹*'v twl-st-init* ⇒ *'v clause*› **where**
‹*literals-to-update-init* ((-, -, -, -, -, -, -, -, -, -, -, *Q*), -) = *Q*›

fun *get-init-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v twl-cls multiset*› **where**
‹*get-init-clauses-init* ((-, *N*, -, -, -, -, -, -), -) = *N*›

fun *get-learned-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v twl-cls multiset*› **where**
‹*get-learned-clauses-init* ((-, -, *U*, -, -, -, -, -), -) = *U*›

fun *get-unit-init-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-unit-init-clauses-init* ((-, -, -, -, *NE*, -, -, -), -) = *NE*›

fun *get-unit-learned-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-unit-learned-clauses-init* ((-, -, -, -, -, *UE*, -, -), -) = *UE*›

fun *get-subsumed-init-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-subsumed-init-clauses-init* ((-, -, -, -, -, -, *NS*, *US*, -, -), -) = *NS*›

fun *get-subsumed-learned-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-subsumed-learned-clauses-init* ((-, -, -, -, -, -, *NS*, *US*, -, -), -) = *US*›

fun *get-subsumed-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-subsumed-clauses-init* ((-, -, -, -, -, -, *NS*, *US*, -, -), -) = *NS* + *US*›

fun *clauses-to-update-init* :: ‹*'v twl-st-init* ⇒ (*'v literal* × *'v twl-cls*) *multiset*› **where**
‹*clauses-to-update-init* ((-, -, -, -, -, -, -, -, -, -, *WS*, -), -) = *WS*›

fun *other-clauses-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*other-clauses-init* ((-, -, -, -, -, -, -), *OC*) = *OC*›

fun *get-init-clauses0-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-init-clauses0-init* ((-, -, -, -, -, -, *NS*, *US*, *N0*, *U0*,-, -), -) = *N0*›

fun *get-learned-clauses0-init* :: ‹*'v twl-st-init* ⇒ *'v clauses*› **where**
‹*get-learned-clauses0-init* ((-, -, -, -, -, -, *NS*, *US*, *N0*, *U0*,-, -), -) = *U0*›

fun *add-to-init-clauses* :: ‹*'v clause-l* ⇒ *'v twl-st-init* ⇒ *'v twl-st-init*› **where**
‹*add-to-init-clauses C* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*), *OC*) =
(*M*, *add-mset* (*twl-clause-of C*) *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*), *OC*)›

fun *add-to-unit-init-clauses* :: ‹*'v clause* ⇒ *'v twl-st-init* ⇒ *'v twl-st-init*› **where**
‹*add-to-unit-init-clauses C* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*), *OC*) =
(*M*, *N*, *U*, *D*, *add-mset C NE*, *UE*, *NS*, *US*, *WS*, *Q*), *OC*)›

fun *set-conflict-init* :: ‹*'v clause-l* ⇒ *'v twl-st-init* ⇒ *'v twl-st-init*› **where**
‹*set-conflict-init C* ((*M*, *N*, *U*, -, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =
(*M*, *N*, *U*, *Some* (*mset C*), *add-mset* (*mset C*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#}), *OC*)›

fun *propagate-unit-init* :: ‹*'v literal* ⇒ *'v twl-st-init* ⇒ *'v twl-st-init*› **where**
‹*propagate-unit-init L* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =
(*Propagated L* {#*L*#} # *M*, *N*, *U*, *D*, *add-mset* {#*L*#} *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* (−*L*) *Q*), *OC*)›

fun *add-empty-conflict-init* :: ‹*'v twl-st-init* ⇒ *'v twl-st-init*› **where**
‹*add-empty-conflict-init* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =
(*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, *NS*, *US*, *add-mset* {#} *N0*, *U0*, *WS*, {#}), *OC*)›

fun *add-to-clauses-init* :: ‹*'v clause-l* ⇒ *'v twl-st-init* ⇒ *'v twl-st-init*› **where**
‹*add-to-clauses-init C* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*), *OC*) =
(*M*, *add-mset* (*twl-clause-of C*) *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*), *OC*)›

fun *add-to-tautology-init* :: ‹*'v clause* ⇒ *'v twl-st-init* ⇒ *'v twl-st-init*› **where**
add-to-tautology-init-def[*simp del*]:
‹*add-to-tautology-init C* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =
(*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset* (*remdups-mset C*) *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*)›

type-synonym *'v twl-st-l-init* = ‹*'v twl-st-l* × *'v clauses*›

fun *get-trail-l-init* :: ‹*'v twl-st-l-init* ⇒ (*'v*, *nat*) *ann-lit list*› **where**
‹*get-trail-l-init* ((*M*, -, -, -, -, -, -), -) = *M*›

fun *get-conflict-l-init* :: ‹*'v twl-st-l-init* ⇒ *'v cconflict*› **where**
‹*get-conflict-l-init* ((-, -, *D*, -, -, -, -), -) = *D*›

fun *get-unit-clauses-l-init* :: ‹*'v twl-st-l-init* ⇒ *'v clauses*› **where**
‹*get-unit-clauses-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), -) = *NE* + *NEk* + *UE* + *UEk*›

fun *get-kept-unit-clauses-l-init* :: ‹*'v twl-st-l-init* ⇒ *'v clauses*› **where**
‹*get-kept-unit-clauses-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), -) = *NEk* + *UEk*›

fun *get-learned-unit-clauses-l-init* :: ‹*'v twl-st-l-init* ⇒ *'v clauses*› **where**
‹*get-learned-unit-clauses-l-init* ((*M*, *N*, *D*, *NEk*, *UEk*, *NE*, *UE*, *WS*, *Q*), -) = *UE* + *UEk*›

fun *get-clauses-l-init* :: ‹*'v twl-st-l-init* ⇒ *'v clauses-l*› **where**
‹*get-clauses-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NS*, *US*, *WS*, *Q*), -) = *N*›

fun *literals-to-update-l-init* :: ‹*'v twl-st-l-init* ⇒ *'v clause*› **where**

‹*literals-to-update-l-init* ((-, -, -, -, -, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, -, *Q*), -) = *Q*›

fun *clauses-to-update-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses-to-update-l*› **where**
‹*clauses-to-update-l-init* ((-, -, -, -, -, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, -), -) = *WS*›

fun *other-clauses-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*other-clauses-l-init* ((-, -, -, -, -, -, -), *OC*) = *OC*›

fun $pstate_W$*-of-init* :: '*v twl-st-init* ⇒ '*v prag-st* **where**
‹$pstate_W$*-of-init* ((*M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*) =
(*M*, *clause* '# *N* + *OC*, *clause* '# *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*)›

fun $state_W$*-of-init* :: '*v twl-st-init* ⇒ '*v* $cdcl_W$*-restart-mset* **where**
$state_W$*-of-init* (*S*) = *state-of* ($pstate_W$*-of-init S*)

fun *get-subsumed-init-clauses-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*get-subsumed-init-clauses-l-init* ((-, -, -, -, -, *NEk*, *UEk*, *NS*, *US*, -, -), -) = *NS*›

fun *get-subsumed-learned-clauses-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*get-subsumed-learned-clauses-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), -) = *US*›

fun *get-subsumed-clauses-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*get-subsumed-clauses-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), -) = *NS*+*US*›

fun *get-init-clauses0-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*get-init-clauses0-l-init* ((-, -, -, -, -, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, -, -), -) = *N0*›

fun *get-learned-clauses0-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*get-learned-clauses0-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*,*WS*, *Q*), -) = *U0*›

fun *get-clauses0-l-init* :: ‹'*v twl-st-l-init* ⇒ '*v clauses*› **where**
‹*get-clauses0-l-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*,*WS*, *Q*), -) = *N0*+*U0*›

named-theorems *twl-st-init* ‹*Convertion for inital theorems*›

lemma [*twl-st-init*]:
‹*get-conflict-init* (*S*, *QC*) = *get-conflict S*›
‹*get-trail-init* (*S*, *QC*) = *get-trail S*›
‹*clauses-to-update-init* (*S*, *QC*) = *clauses-to-update S*›
‹*literals-to-update-init* (*S*, *QC*) = *literals-to-update S*›
by (*solves* ‹*cases S*; *auto*›)+

lemma [*twl-st-init*]:
‹*clauses-to-update-init* (*add-to-unit-init-clauses* (*mset C*) *T*) = *clauses-to-update-init T*›
‹*literals-to-update-init* (*add-to-unit-init-clauses* (*mset C*) *T*) = *literals-to-update-init T*›
‹*get-conflict-init* (*add-to-unit-init-clauses* (*mset C*) *T*) = *get-conflict-init T*›
apply (*cases T*; *auto simp*: *twl-st-inv.simps*; *fail*)+
done
lemma [*twl-st-init*]:
‹*twl-st-inv* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *twl-st-inv* (*fst T*)›
‹*valid-enqueued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *valid-enqueued* (*fst T*)›
‹*no-duplicate-queued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *no-duplicate-queued* (*fst T*)›
‹*distinct-queued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *distinct-queued* (*fst T*)›
‹*confl-cands-enqueued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *confl-cands-enqueued* (*fst T*)›
‹*propa-cands-enqueued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *propa-cands-enqueued* (*fst T*)›

‹*twl-st-exception-inv* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *twl-st-exception-inv* (*fst T*)›
apply (*cases T*; *auto simp*: *twl-st-inv.simps*; *fail*)+
apply (*cases* ‹*get-conflict-init T*›; *cases T*;
auto simp: *twl-st-inv.simps twl-exception-inv.simps*; *fail*)+
done

lemma [*twl-st-init*]:
‹*trail* (*state*$_W$*-of-init T*) = *get-trail-init T*›
‹*get-trail* (*fst T*) = *get-trail-init* (*T*)›
‹*conflicting* (*state*$_W$*-of-init T*) = *get-conflict-init T*›
‹*init-clss* (*state*$_W$*-of-init T*) = *clauses* (*get-init-clauses-init T*) + *get-unit-init-clauses-init T* +
get-subsumed-init-clauses-init T + *get-init-clauses0-init T* + *other-clauses-init T*›
‹*learned-clss* (*state*$_W$*-of-init T*) = *clauses* (*get-learned-clauses-init T*) +
get-unit-learned-clauses-init T + *get-subsumed-learned-clauses-init T* + *get-learned-clauses0-init T*›
‹*conflicting* (*state*$_W$*-of* (*fst T*)) = *conflicting* (*state*$_W$*-of-init T*)›
‹*trail* (*state*$_W$*-of* (*fst T*)) = *trail* (*state*$_W$*-of-init T*)›
‹*clauses-to-update* (*fst T*) = *clauses-to-update-init T*›
‹*get-conflict* (*fst T*) = *get-conflict-init T*›
‹*literals-to-update* (*fst T*) = *literals-to-update-init T*›
‹*subsumed-learned-clss* (*fst T*) = *get-subsumed-learned-clauses-init T*›
by (*cases T*; *auto simp*: *cdcl*$_W$*-restart-mset-state*; *fail*)+

definition *twl-st-l-init* :: ‹(*'v twl-st-l-init* × *'v twl-st-init*) *set*› **where**
‹*twl-st-l-init* = {(((*M*, *N*, *C*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*),
((*M'*, *N'*, *C'*, *NE'*, *UE'*, *NS'*, *US'*, *N0'*, *U0'*, *WS'*, *Q'*), *OC'*)).
(*M* , *M'*) ∈ *convert-lits-l N* (*NEk*+*UEk*) ∧
((*N'*, *C'*, *NE'*, *UE'*, *NS'*, *US'*, *N0'*, *U0'*, *WS'*, *Q'*), *OC'*) =
((*twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*,
C, *NE* + *NEk*, *UE* + *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*), *OC*)}›

lemma *twl-st-l-init-alt-def*:
‹(*S*, *T*) ∈ *twl-st-l-init* ⟷
(*fst S*, *fst T*) ∈ *twl-st-l None* ∧ *other-clauses-l-init S* = *other-clauses-init T*›
by (*cases S*; *cases T*) (*auto simp*: *twl-st-l-init-def twl-st-l-def*)

lemma [*twl-st-init*]:
assumes ‹(*S*, *T*) ∈ *twl-st-l-init*›
shows
‹*get-conflict-init T* = *get-conflict-l-init S*›
‹*get-conflict* (*fst T*) = *get-conflict-l-init S*›
‹*literals-to-update-init T* = *literals-to-update-l-init S*›
‹*clauses-to-update-init T* = {#}›
‹*other-clauses-init T* = *other-clauses-l-init S*›
‹*lits-of-l* (*get-trail-init T*) = *lits-of-l* (*get-trail-l-init S*)›
‹*lit-of* ‘# *mset* (*get-trail-init T*) = *lit-of* ‘# *mset* (*get-trail-l-init S*)›
by (*use assms* **in** ‹*solves* ‹*cases S*; *auto simp*: *twl-st-l-init-def*››)+

definition *twl-struct-invs-init* :: ‹*'v twl-st-init* ⇒ *bool*› **where**
‹*twl-struct-invs-init S* ⟷
(*twl-st-inv* (*fst S*) ∧
valid-enqueued (*fst S*) ∧
pcdcl-all-struct-invs (*pstate*$_W$*-of-init S*) ∧
cdcl$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of-init S*) ∧
twl-st-exception-inv (*fst S*) ∧
no-duplicate-queued (*fst S*) ∧
distinct-queued (*fst S*) ∧

confl-cands-enqueued (*fst S*) ∧
propa-cands-enqueued (*fst S*) ∧
(*get-conflict-init S* ≠ *None* ⟶ *clauses-to-update-init S* = {#} ∧ *literals-to-update-init S* = {#}) ∧
clauses-to-update-inv (*fst S*) ∧
past-invs (*fst S*))
›

lemma *state*$_W$*-of-state*$_W$*-of-init*:
‹*other-clauses-init W* = {#} ⟹ *state*$_W$*-of* (*fst W*) = *state*$_W$*-of-init W*›
by (*cases W*) *auto*

lemma *twl-struct-invs-init-twl-struct-invs*:
‹*other-clauses-init W* = {#} ⟹ *twl-struct-invs-init W* ⟹ *twl-struct-invs* (*fst W*)›
unfolding *twl-struct-invs-def twl-struct-invs-init-def*
apply (*subst state*$_W$*-of-state*$_W$*-of-init*; *assumption?*)+
apply (*intro iffI impI conjI*)
by (*cases W*; *clarsimp simp*: *twl-st-init*)+

fun *add-empty-conflict-init-l* :: ‹'*v twl-st-l-init* ⇒ '*v twl-st-l-init*› **where**
add-empty-conflict-init-l-def[*simp del*]:
‹*add-empty-conflict-init-l* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =
((*M*, *N*, *Some* {#}, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *add-mset* {#} *N0*, *U0*, *WS*, {#}), *OC*)›

fun *propagate-unit-init-l* :: ‹'*v literal* ⇒ '*v twl-st-l-init* ⇒ ('*v twl-st-l-init*) *nres*› **where**
propagate-unit-init-l-def[*simp del*]:
‹*propagate-unit-init-l L* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) = *do* {
M ← *cons-trail-propagate-l L 0 M*;
RETURN ((*M*, *N*, *D*, *NE*, *UE*, *add-mset* {#*L*#} *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* (−*L*) *Q*), *OC*)
}›

fun *already-propagated-unit-init-l* :: ‹'*v clause* ⇒ '*v twl-st-l-init* ⇒ '*v twl-st-l-init*› **where**
already-propagated-unit-init-l-def[*simp del*]:
‹*already-propagated-unit-init-l C* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), *OC*) =
((*M*, *N*, *D*, *NE*, *UE*, *add-mset C NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), *OC*)›

fun *set-conflict-init-l* :: ‹'*v clause-l* ⇒ '*v twl-st-l-init* ⇒ '*v twl-st-l-init*› **where**
set-conflict-init-l-def[*simp del*]:
‹*set-conflict-init-l C* ((*M*, *N*, -, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =
((*M*, *N*, *Some* (*mset C*), *add-mset* (*mset C*) *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, {#}), *OC*)›

fun *add-to-clauses-init-l* :: ‹'*v clause-l* ⇒ '*v twl-st-l-init* ⇒ '*v twl-st-l-init nres*› **where**
add-to-clauses-init-l-def[*simp del*]:
‹*add-to-clauses-init-l C* ((*M*, *N*, -, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), *OC*) = *do* {
i ← *get-fresh-index N*;
RETURN ((*M*, *fmupd i* (*C*, *True*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *WS*, *Q*), *OC*)
}›

fun *add-to-tautology-init-l* :: ‹'*v clause-l* ⇒ '*v twl-st-l-init* ⇒ '*v twl-st-l-init*› **where**
add-to-tautology-init-l-def[*simp del*]:
‹*add-to-tautology-init-l C* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*) =

((M, N, D, NE, UE, NEk, UEk, add-mset (remdups-mset (mset C)) NS, US, N0, U0, WS, Q), OC)›

fun *add-to-other-init* **where**
‹*add-to-other-init C (S, OC) = (S, add-mset (remdups-mset (mset C)) OC)›

lemma *fst-add-to-other-init* [*simp*]: ‹*fst* (*add-to-other-init a T*) = *fst T*›
by (*cases T*) *auto*

definition *remdups-clause* **where**
‹*remdups-clause C* = *SPEC* (λ*C'*. *mset C'* = *remdups-mset* (*mset C*))›

definition *init-dt-step* :: ‹'*v clause-l* ⇒ '*v twl-st-l-init* ⇒ '*v twl-st-l-init nres*› **where**
‹*init-dt-step C S* =
(*case get-conflict-l-init S of*
None ⇒
if tautology (*mset C*)
then RETURN (*add-to-tautology-init-l C S*)
else do {
C ← *remdups-clause C*;
if length C = *0*
then RETURN (*add-empty-conflict-init-l S*)
else if length C = *1*
then
let L = *hd C in*
if undefined-lit (*get-trail-l-init S*) *L*
then propagate-unit-init-l L S
else if L ∈ *lits-of-l* (*get-trail-l-init S*)
then RETURN (*already-propagated-unit-init-l* (*mset C*) *S*)
else RETURN (*set-conflict-init-l C S*)
else
add-to-clauses-init-l C S
}
| *Some D* ⇒
RETURN (*add-to-other-init C S*))›

definition *init-dt* :: ‹'*v clause-l list* ⇒ '*v twl-st-l-init* ⇒ '*v twl-st-l-init nres*› **where**
‹*init-dt CS S* = *nfoldli CS* (λ-. *True*) *init-dt-step S*›

definition *init-dt-pre* :: ‹'*v clause-l list* ⇒ -› **where**
‹*init-dt-pre CS SOC* ⟷
(∃ *T*. (*SOC*, *T*) ∈ *twl-st-l-init* ∧
twl-struct-invs-init T ∧
clauses-to-update-l-init SOC = {#} ∧
(∀ *s*∈*set* (*get-trail-l-init SOC*). ¬*is-decided s*) ∧
(*get-conflict-l-init SOC* = *None* ⟶
literals-to-update-l-init SOC = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init SOC*)) ∧
twl-list-invs (*fst SOC*) ∧
twl-stgy-invs (*fst T*) ∧
(*other-clauses-l-init SOC* ≠ {#} ⟶ *get-conflict-l-init SOC* ≠ *None*) ∧
(∀ *C*∈#*ran-mf* (*get-clauses-l-init SOC*). ¬*tautology* (*mset C*)))›

lemma *init-dt-pre-ConsD*: ‹*init-dt-pre* (*a* # *CS*) *SOC* ⟹ *init-dt-pre CS SOC*›
unfolding *init-dt-pre-def*
apply *normalize-goal*+
by *fastforce*

definition *init-dt-spec* **where**
‹*init-dt-spec CS SOC SOC'* ⟷
(∃ *T'*. (*SOC'*, *T'*) ∈ *twl-st-l-init* ∧
twl-struct-invs-init T' ∧
clauses-to-update-l-init SOC' = {#} ∧
(∀ *s*∈*set* (*get-trail-l-init SOC'*). ¬*is-decided s*) ∧
(*get-conflict-l-init SOC'* = *None* ⟶
literals-to-update-l-init SOC' = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init SOC'*)) ∧
(*remdups-mset* ‘# *mset* ‘# *mset CS* + *mset* ‘# *ran-mf* (*get-clauses-l-init SOC*) + *other-clauses-l-init SOC* +
get-unit-clauses-l-init SOC + *get-subsumed-init-clauses-l-init SOC*+ *get-init-clauses0-l-init SOC* =
mset ‘# *ran-mf* (*get-clauses-l-init SOC'*) + *other-clauses-l-init SOC'* +
get-unit-clauses-l-init SOC' + *get-subsumed-init-clauses-l-init SOC'* +
get-init-clauses0-l-init SOC') ∧
learned-clss-lf (*get-clauses-l-init SOC*) = *learned-clss-lf* (*get-clauses-l-init SOC'*) ∧
get-learned-unit-clauses-l-init SOC' = *get-learned-unit-clauses-l-init SOC* ∧
get-subsumed-learned-clauses-l-init SOC' = *get-subsumed-learned-clauses-l-init SOC* ∧
get-learned-clauses0-l-init SOC' = *get-learned-clauses0-l-init SOC* ∧
twl-list-invs (*fst SOC'*) ∧
twl-stgy-invs (*fst T'*) ∧
(*other-clauses-l-init SOC'* ≠ {#} ⟶ *get-conflict-l-init SOC'* ≠ *None*) ∧
({#} ∈# *mset* ‘# *mset CS* ⟶ *get-conflict-l-init SOC'* ≠ *None*) ∧
(*get-conflict-l-init SOC* ≠ *None* ⟶ *get-conflict-l-init SOC* = *get-conflict-l-init SOC'*)›

lemma *twl-struct-invs-init-add-to-other-init*:
assumes
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*›
shows
‹*twl-struct-invs-init* (*add-to-other-init a T*)›
(**is** *?twl-struct-invs-init*)
proof −
let *?a* = ‹*remdups a*›
obtain *M N U D NE UE Q OC WS NS US N0 U0* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*)›
by (*cases T*) *auto*
have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *clauses N* + *NE* + *NS* + *N0* + *OC*, *clauses U* + *UE* + *US* + *U0*, *D*)› **and**
smaller: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *NE* + *NS* + *N0* + *OC*, *clauses U* + *UE* + *US* + *U0*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** (*auto simp*: *ac-simps*)
then have
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *add-mset* (*mset ?a*) (*clauses N* + *NE* + *NS* + *N0* + *OC*),
clauses U + *UE* + *US* + *U0*, *D*)›
by (*auto simp*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.no-strange-atm-def cdcl*$_W$*-restart-mset-state*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def all-decomposition-implies-def*
clauses-def cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
cdcl$_W$*-restart-mset.reasons-in-clauses-def*)
then have ‹*pcdcl-all-struct-invs* (*M*, *add-mset* (*mset ?a*) (*clauses N* + *OC*), *clauses U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*)›
using *invs* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def*

by (*auto simp*: *ac-simps entailed-clss-inv-def psubsumed-invs-def clauses0-inv-def*)

moreover have
‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*M*, *add-mset* (*mset ?a*) (*clauses N* + *NE* + *NS* + *N0* + *OC*),
clauses U + *UE* + *US* + *U0*, *D*)›
using *lev smaller*
by (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def cdcl*$_W$*-restart-mset-state clauses-def T count-decided-0-iff*)
ultimately show *?twl-struct-invs-init*
using *invs*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv add-to-other-init.simps state*$_W$*-of-init.simps get-conflict.simps*
by (*clarsimp simp*: *ac-simps*)
qed

lemma *clauses-to-update-inv-add-subsumed*[*simp*]:
‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset a NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
clauses-to-update-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset a US*, *N0*, *U0*, *WS*, *Q*) =
clauses-to-update-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto*; *fail*)+

lemma *confl-cands-enqueued-add-subsumed*[*simp*]:
‹*confl-cands-enqueued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset a NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ›
by (*cases D*; *auto*; *fail*)+

lemma *propa-cands-enqueued-add-subsumed*[*simp*]:
‹*propa-cands-enqueued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset a NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
propa-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ›
by (*cases D*; *auto*; *fail*)+

lemma *twl-exception-inv-add-subsumed*[*simp*]:
‹*twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset a NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
twl-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset a US*, *N0*, *U0*, *WS*, *Q*) =
twl-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto simp*: *twl-exception-inv.simps*; *fail*)+

lemma *past-invs-add-subsumed*[*simp*]:
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset a NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
past-invs (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset a US*, *N0*, *U0*, *WS*, *Q*) =
past-invs (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto simp*: *past-invs.simps*; *fail*)+

lemma *twl-st-inv-add-subsumed*[*simp*]:
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *add-mset a NS*, *US*, *N0*, *U0*, *WS*, *Q*) =
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *add-mset a US*, *N0*, *U0*, *WS*, *Q*) =
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto simp*: *twl-st-inv.simps*; *fail*)+

lemma *twl-struct-invs-init-add-to-tautology-init*:
assumes

tauto: ‹*tautology a*› **and**
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*›
shows
‹*twl-struct-invs-init* (*add-to-tautology-init a T*)›
(**is** *?twl-struct-invs-init*)
proof −
let *?a* = ‹*remdups-mset a*›
obtain *M N U D NE UE Q OC WS NS US N0 U0* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*)›
by (*cases T*) *auto*
have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *clauses N* + *NE* + *NS* + *N0* + *OC*, *clauses U* + *UE* + *US* + *U0*, *D*)› **and**
smaller: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *NE* + *NS* + *N0* + *OC*, *clauses U* + *UE* + *US* + *U0*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** (*auto simp*: *ac-simps*)
then have
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *add-mset ?a* (*clauses N* + *NE* + *NS* + *N0* + *OC*), *clauses U* + *UE* + *US* + *U0*, *D*)›
by (*auto simp*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.no-strange-atm-def cdcl*$_W$*-restart-mset-state*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def all-decomposition-implies-def*
clauses-def cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
cdcl$_W$*-restart-mset.reasons-in-clauses-def*)
then have ‹*pcdcl-all-struct-invs* (*M*, (*clauses N* + *OC*), *clauses U*, *D*, *NE*, *UE*, *add-mset ?a NS*, *US*, *N0*, *U0*)›
using *invs tauto* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def*
by (*auto simp*: *ac-simps entailed-clss-inv-def psubsumed-invs-def clauses0-inv-def*)

moreover have
‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*M*, *add-mset ?a* (*clauses N* + *NE* + *NS* + *N0* + *OC*), *clauses U* + *UE* + *US* + *U0*, *D*)›
using *lev smaller*
by (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def cdcl*$_W$*-restart-mset-state*
clauses-def T count-decided-0-iff)
ultimately show *?twl-struct-invs-init*
using *invs*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv state*$_W$*-of-init.simps get-conflict.simps add-to-tautology-init-def*
by (*clarsimp simp*: *ac-simps*)
qed

lemma *invariants-init-state*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
wf: ‹∀ *C* ∈# *get-clauses* (*fst T*). *struct-wf-twl-cls C*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
n-d: ‹*no-dup* (*get-trail-init T*)›
shows ‹*propa-cands-enqueued* (*fst T*)› **and** ‹*confl-cands-enqueued* (*fst T*)› **and** ‹*twl-st-inv* (*fst T*)›
‹*clauses-to-update-inv* (*fst T*)› **and** ‹*past-invs* (*fst T*)› **and** ‹*distinct-queued* (*fst T*)› **and**
‹*valid-enqueued* (*fst T*)› **and** ‹*twl-st-exception-inv* (*fst T*)› **and** ‹*no-duplicate-queued* (*fst T*)›
proof −
obtain *M N U NE UE OC D NS US N0 U0* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *uminus* ‘# *lit-of* ‘# *mset M*), *OC*)›

```
  using MQ WS by (cases T) auto
let ?Q = ‹uminus `# lit-of `# mset M›

have [iff]: ‹M = M' @ Decided K # Ma ⟷ False› for M' K Ma
  using lev by (auto simp: count-decided-0-iff T)

have struct: ‹struct-wf-twl-cls C› if ‹C ∈# N + U› for C
  using wf that by (simp add: T twl-st-inv.simps)
let ?T = ‹fst T›
have [simp]: ‹propa-cands-enqueued ?T› if D: ‹D = None›
  unfolding propa-cands-enqueued.simps Ball-def T fst-conv D
  apply − apply (intro conjI impI allI)
  subgoal for x C
    using struct[of C]
    apply (case-tac C; auto simp: uminus-lit-swap lits-of-def size-2-iff
        true-annots-true-cls-def-iff-negation-in-model Ball-def remove1-mset-add-mset-If
        all-conj-distrib conj-disj-distribR ex-disj-distrib
        split: if-splits)
    done
  done
then show ‹propa-cands-enqueued ?T›
  by (cases D) (auto simp: T)

have [simp]: ‹confl-cands-enqueued ?T› if D: ‹D = None›
  unfolding confl-cands-enqueued.simps Ball-def T D fst-conv
  apply − apply (intro conjI impI allI)
  subgoal for x
    using struct[of x]
    by (case-tac x; case-tac ‹watched x›; auto simp: uminus-lit-swap lits-of-def)
  done
then show ‹confl-cands-enqueued ?T›
  by (cases D) (auto simp: T)
have [simp]: ‹get-level M L = 0› for L
  using lev by (auto simp: T count-decided-0-iff)
show [simp]: ‹twl-st-inv ?T›
  unfolding T fst-conv twl-st-inv.simps Ball-def
  apply − apply (intro conjI impI allI)
  subgoal using wf by (auto simp: T)
  subgoal for C
    by (cases C)
      (auto simp: T twl-st-inv.simps twl-lazy-update.simps twl-is-an-exception-def
        lits-of-def uminus-lit-swap)
  subgoal for C
    using lev by (cases C)
      (auto simp: T twl-st-inv.simps twl-lazy-update.simps)
  done
have [simp]: ‹{#C ∈# N. clauses-to-update-prop {#− lit-of x. x ∈# mset M#} M (L, C)#} = {#}›
  for L N
  by (auto simp: filter-mset-empty-conv clauses-to-update-prop.simps lits-of-def
      uminus-lit-swap)
have ‹clauses-to-update-inv ?T› if D: ‹D = None›
  unfolding T D
  by (auto simp: filter-mset-empty-conv lits-of-def uminus-lit-swap)
then show ‹clauses-to-update-inv (fst T)›
  by (cases D) (auto simp: T)
```

show ‹*past-invs ?T*›
by (*auto simp*: *T past-invs.simps*)

show ‹*distinct-queued ?T*›
using *WS n-d* **by** (*auto simp*: *T no-dup-distinct-uminus*)
show ‹*valid-enqueued ?T*›
using *lev* **by** (*auto simp*: *T lits-of-def*)

show ‹*twl-st-exception-inv* (*fst T*)›
unfolding *T fst-conv twl-st-exception-inv.simps Ball-def*
apply − **apply** (*intro conjI impI allI*)
apply (*case-tac x*; *cases D*)
by (*auto simp*: *T twl-exception-inv.simps lits-of-def uminus-lit-swap*)

show ‹*no-duplicate-queued* (*fst T*)›
by (*auto simp*: *T*)
qed

lemma *twl-struct-invs-init-init-state*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
wf: ‹∀ *C* ∈# *get-clauses* (*fst T*). *struct-wf-twl-cls C*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
struct-invs: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of-init T*)› **and**
‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$ *-of-init T*)› **and**
‹*get-conflict-init T* ≠ *None* ⟶ *clauses-to-update-init T* = {#} ∧ *literals-to-update-init T* = {#}›
shows ‹*twl-struct-invs-init T*›
proof −
have *n-d*: ‹*no-dup* (*get-trail-init T*)›
using *struct-invs* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def pcdcl-all-struct-invs-def*
by (*cases T*) (*auto simp*: *trail.simps*)
then show *?thesis*
using *invariants-init-state*[*OF lev wf MQ WS n-d*] *assms* **unfolding** *twl-struct-invs-init-def*
by *fast*+
qed

lemma *twl-struct-invs-init-add-to-unit-init-clauses*:
assumes
dist: ‹*distinct a*› **and**
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
ex: ‹∃ *L* ∈ *set a*. *L* ∈ *lits-of-l* (*get-trail-init T*)›
shows
‹*twl-struct-invs-init* (*add-to-unit-init-clauses* (*mset a*) *T*)›
(**is** *?all-struct*)
proof −
have [*simp*]: ‹*remdups-mset* (*mset a*) = *mset a*›
using *dist* **by** (*simp add*: *mset-set-set remdups-mset-def*)
obtain *M N U D NE UE Q NS US N0 U0 OC WS* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*), *OC*)›
by (*cases T*) *auto*
have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *D*)›

using *invs* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** (*auto simp*: *ac-simps*)
then have [*simp*]:
‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv* (*M*, *add-mset* (*mset a*) (*OC* + *clauses N* + *NE* + *NS* + *N0*),
clauses U + *UE* + *US* + *U0*, *D*)›
using *twl-struct-invs-init-add-to-other-init*[*OF lev invs*, *of a*] *dist*
unfolding *T twl-struct-invs-init-def pcdcl-all-struct-invs-def*
by (*simp add*: *ac-simps*)
then have *invs'*: ‹*pcdcl-all-struct-invs*
(*M*, *clauses N* + *OC*, *clauses U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*)›
using *invs count-decided-ge-get-level*[*of M*] *lev ex* **unfolding** *twl-struct-invs-init-def pcdcl-all-struct-invs-def*
by (*auto simp*: *psubsumed-invs-def entailed-clss-inv-def T ac-simps clauses0-inv-def*)
have ‹*cdcl$_W$-restart-mset.no-smaller-propa* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹*cdcl$_W$-restart-mset.no-smaller-propa* (*M*, *add-mset* (*mset a*) (*clauses N* + *OC* + *NE* + *NS* + *N0*),
clauses U + *UE* + *US* + *U0*, *D*)›
using *lev*
by (*auto simp*: *cdcl$_W$-restart-mset.no-smaller-propa-def cdcl$_W$-restart-mset-state*
clauses-def T count-decided-0-iff)
have [*simp*]: ‹*confl-cands-enqueued* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*propa-cands-enqueued* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
propa-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹⋀*x*. *twl-exception-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) *x* ⟷
twl-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) *x*›
‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
clauses-to-update-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
‹*past-invs* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
past-invs (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
by (*cases D*; *auto simp*: *twl-st-inv.simps twl-exception-inv.simps past-invs.simps*; *fail*)+
show *?all-struct*
using *invs ex invs'*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv add-to-other-init.simps state$_W$-of-init.simps get-conflict.simps*
by (*clarsimp simp del*: *entailed-clss-inv.simps*)
qed

lemma *twl-struct-invs-init-set-conflict-init*:
assumes
dist: ‹*distinct C*› **and**
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
ex: ‹∀ *L* ∈ *set C*. −*L* ∈ *lits-of-l* (*get-trail-init T*)› **and**
nempty: ‹*C* ≠ []› **and**
confl: ‹*get-conflict-init T* = *None*›
shows
‹*twl-struct-invs-init* (*set-conflict-init C T*)›
(**is** *?all-struct*)
proof −
obtain *M N U D NE UE Q OC WS NS US N0 U0* **where**

T: ‹$T = ((M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q),\ OC)$›
by (*cases* T) *auto*
have ‹$cdcl_W$-*restart-mset.cdcl*$_W$-*all-struct-inv* (M, *clauses* N + OC + NE + NS + $N0$, *clauses* U + UE + US + $U0$, D)› **and**
pinvs: ‹*pcdcl-all-struct-invs* (*pstate*$_W$-*of-init* (($M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$), OC))›
using *invs* **unfolding** T *twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** *auto*
then have
‹$cdcl_W$-*restart-mset.cdcl*$_W$-*all-struct-inv* (M, *add-mset* (*mset* C) (*clauses* N + OC+ NE + NS + $N0$),
clauses U + UE + US + $U0$, *Some* (*mset* C))›
using *dist ex*
unfolding T *twl-struct-invs-init-def*
by (*auto 5 5 simp*: $cdcl_W$-*restart-mset.cdcl*$_W$-*all-struct-inv-def*
$cdcl_W$-*restart-mset.no-strange-atm-def* $cdcl_W$-*restart-mset-state*
$cdcl_W$-*restart-mset.cdcl*$_W$-*M-level-inv-def* $cdcl_W$-*restart-mset.cdcl*$_W$-*conflicting-def*
$cdcl_W$-*restart-mset.distinct-cdcl*$_W$-*state-def all-decomposition-implies-def*
clauses-def $cdcl_W$-*restart-mset.cdcl*$_W$-*learned-clause-alt-def*
true-annots-true-cls-def-iff-negation-in-model)
then have H: ‹*pcdcl-all-struct-invs* (*pstate*$_W$-*of-init* (($M,\ N,\ U$, *Some* (*mset* C), *add-mset* (*mset* C) $NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$), OC))›
using *pinvs ex count-decided-ge-get-level*[*of* M] *lev nempty confl*
unfolding *pcdcl-all-struct-invs-def entailed-clss-inv-def psubsumed-invs-def*
by (*auto simp*: *entailed-clss-inv-def* T *clauses0-inv-def*)

have ‹$cdcl_W$-*restart-mset.no-smaller-propa* (M, *clauses* N + OC + NE + NS + $N0$, *clauses* U + UE + US + $U0$, D)›
using *invs* **unfolding** T *twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹$cdcl_W$-*restart-mset.no-smaller-propa* (M, *add-mset* (*mset* C) (*clauses* N + OC + NE + NS + $N0$),
clauses U + UE + US + $U0$, *Some* (*mset* C))›
using *lev*
by (*auto simp*: $cdcl_W$-*restart-mset.no-smaller-propa-def* $cdcl_W$-*restart-mset-state*
clauses-def T *count-decided-0-iff*)
let *?T* = ‹($M,\ N,\ U$, *Some* (*mset* C), *add-mset* (*mset* C) $NE,\ UE,\ NS,\ US,\ N0,\ U0$, {#}, {#})›

have [*simp*]: ‹*confl-cands-enqueued ?T*›
‹*propa-cands-enqueued ?T*›
‹*twl-st-inv* ($M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$) ⟹ *twl-st-inv ?T*›
‹$\bigwedge x.$ *twl-exception-inv* ($M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$) x ⟹ *twl-exception-inv ?T x*›
‹*clauses-to-update-inv* ($M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$) ⟹ *clauses-to-update-inv ?T*›
‹*past-invs* ($M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q$) ⟹ *past-invs ?T*›
by (*auto simp*: *twl-st-inv.simps twl-exception-inv.simps past-invs.simps*; *fail*)+

show *?all-struct*
using *invs ex H*
unfolding *twl-struct-invs-init-def* T
unfolding *fst-conv add-to-other-init.simps state*$_W$-*of-init.simps get-conflict.simps*
by (*clarsimp simp del*: *entailed-clss-inv.simps*)
qed

lemma *twl-struct-invs-init-propagate-unit-init*:
assumes

lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
undef: ‹*undefined-lit* (*get-trail-init T*) *L*› **and**
confl: ‹*get-conflict-init T* = *None*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}›
shows
‹*twl-struct-invs-init* (*propagate-unit-init L T*)›
(**is** *?all-struct*)
proof −
obtain *M N U NE UE OC WS NS US N0 U0* **where**
T: ‹*T* = ((*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *uminus* ‘# *lit-of* ‘# *mset M*), *OC*)›
using *confl MQ* **by** (*cases T*) *auto*
let *?Q* = ‹*uminus* ‘# *lit-of* ‘# *mset M*›
have [*iff*]: ‹− *L* ∈ *lits-of-l M* ⟷ *False*›
using *undef* **by** (*auto simp*: *T Decided-Propagated-in-iff-in-lits-of-l*)
have [*simp*]: ‹*get-all-ann-decomposition M* = [([], *M*)]›
by (*rule no-decision-get-all-ann-decomposition*) (*use lev* **in** ‹*auto simp*: *T count-decided-0-iff*›)
have *H*: ‹*a* @ *Propagated L′ mark′* # *b* = *Propagated L mark* # *M* ⟷
(*a* = [] ∧ *L* = *L′* ∧ *mark* = *mark′* ∧ *b* = *M*) ∨
(*a* ≠ [] ∧ *hd a* = *Propagated L mark* ∧ *tl a* @ *Propagated L′ mark′* # *b* = *M*)›
for *a mark mark′ L′ b*
using *undef* **by** (*cases a*) (*auto simp*: *T atm-of-eq-atm-of*)
have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *None*)› **and**
excep: ‹*twl-st-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *?Q*)› **and**
st-inv: ‹*twl-st-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *?Q*)› **and**
pinvs: ‹*pcdcl-all-struct-invs* (*pstate*$_W$*-of-init T*)›
using *invs confl* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** *auto*
then have [*simp*]:
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *add-mset* {#*L*#} (*clauses N* + *OC* + *NE* + *NS* + *N0*),
clauses U + *UE* + *US* + *U0*, *None*)› **and**
n-d: ‹*no-dup M*›
by (*auto simp*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def T*
cdcl$_W$*-restart-mset.no-strange-atm-def cdcl*$_W$*-restart-mset-state*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def all-decomposition-implies-def*
clauses-def cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
cdcl$_W$*-restart-mset.reasons-in-clauses-def*)
then have
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*Propagated L* {#*L*#} # *M*,
add-mset {#*L*#} (*clauses N* + *OC* + *NE* + *NS* + *N0*), *clauses U* + *UE* + *US* + *U0*, *None*)›
using *undef* **by** (*auto simp*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def T H*
cdcl$_W$*-restart-mset.no-strange-atm-def cdcl*$_W$*-restart-mset-state*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def all-decomposition-implies-def*
clauses-def cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
consistent-interp-insert-iff)
then have *pinvs′*: ‹*pcdcl-all-struct-invs* (*pstate*$_W$*-of-init* (*propagate-unit-init L T*))›
using *pinvs lev count-decided-ge-get-level*[*of* ‹*Propagated L* {#*L*#} # *M*›] **unfolding** *pcdcl-all-struct-invs-def T*
by (*auto simp*: *entailed-clss-inv-def psubsumed-invs-def clauses0-inv-def*)
have [*iff*]: ‹*Propagated L* {#*L*#} # *M* = *M′* @ *Decided K* # *Ma* ⟷ *False*› **for** *M′ K Ma*
using *lev* **by** (*cases M′*) (*auto simp*: *count-decided-0-iff T*)
have ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* +

UE + *US* + *U0*, *None*)›
  using *invs confl* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
 then have [*simp*]:
  ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*Propagated L* {#*L*#} # *M*, *add-mset* {#*L*#} (*clauses N* + *NE* + *NS* + *N0* + *OC*),
    clauses U + *UE* + *US* + *U0*, *None*)›
  using *lev*
  by (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def cdcl*$_W$*-restart-mset-state*
    clauses-def T count-decided-0-iff)

 have ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *None*)›
  using *invs confl* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** *auto*
 then have [*simp*]:
  ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*Propagated L* {#*L*#} # *M*, *add-mset* {#*L*#} (*clauses N* + *OC* + *NE* + *NS* + *N0*),
    clauses U + *UE* + *US* + *U0*, *None*)›
  using *lev*
  by (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def cdcl*$_W$*-restart-mset-state*
    clauses-def T count-decided-0-iff)
 let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *?Q*)›
 let *?T* = ‹(*Propagated L* {#*L*#} # *M*, *N*, *U*, *None*, *add-mset* {#*L*#} *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *add-mset* (−*L*) *?Q*)›

 have *struct*: ‹*struct-wf-twl-cls C*› **if** ‹*C* ∈# *N* + *U*› **for** *C*
  using *st-inv that* **by** (*simp add*: *twl-st-inv.simps*)
 show ‹*twl-struct-invs-init* (*propagate-unit-init L T*)›
  apply (*rule twl-struct-invs-init-init-state*)
  subgoal using *lev* **by** (*auto simp*: *T*)
  subgoal using *struct* **by** (*auto simp*: *T*)
  subgoal using *MQ* **by** (*auto simp*: *T*)
  subgoal using *WS* **by** (*auto simp*: *T*)
  subgoal using *pinvs′* **by** (*simp add*: *T*)
  subgoal by (*auto simp*: *T*)
  subgoal by (*auto simp*: *T*)
  done
qed

named-theorems *twl-st-l-init*
lemma [*twl-st-l-init*]:
 ‹*clauses-to-update-l-init* (*already-propagated-unit-init-l C S*) = *clauses-to-update-l-init S*›
 ‹*get-trail-l-init* (*already-propagated-unit-init-l C S*) = *get-trail-l-init S*›
 ‹*get-conflict-l-init* (*already-propagated-unit-init-l C S*) = *get-conflict-l-init S*›
 ‹*other-clauses-l-init* (*already-propagated-unit-init-l C S*) = *other-clauses-l-init S*›
 ‹*clauses-to-update-l-init* (*already-propagated-unit-init-l C S*) = *clauses-to-update-l-init S*›
 ‹*literals-to-update-l-init* (*already-propagated-unit-init-l C S*) = *literals-to-update-l-init S*›
 ‹*get-clauses-l-init* (*already-propagated-unit-init-l C S*) = *get-clauses-l-init S*›
 ‹*get-unit-clauses-l-init* (*already-propagated-unit-init-l C S*) = *add-mset C* (*get-unit-clauses-l-init S*)›
 ‹*get-learned-unit-clauses-l-init* (*already-propagated-unit-init-l C S*) =
    get-learned-unit-clauses-l-init S›
 ‹*get-subsumed-learned-clauses-l-init* (*already-propagated-unit-init-l C S*) =
    get-subsumed-learned-clauses-l-init S›
 ‹*get-subsumed-init-clauses-l-init* (*already-propagated-unit-init-l C S*) =
    get-subsumed-init-clauses-l-init S›
 ‹*get-learned-clauses0-l-init* (*already-propagated-unit-init-l C S*) = *get-learned-clauses0-l-init S*›
 ‹*get-init-clauses0-l-init* (*already-propagated-unit-init-l C S*) = *get-init-clauses0-l-init S*›

‹get-conflict-l-init (T, OC) = get-conflict-l T›
by (solves ‹cases S; cases T; auto simp: already-propagated-unit-init-l-def›)+

lemma [twl-st-l-init]:
‹clauses-to-update-l-init (add-to-tautology-init-l C S) = clauses-to-update-l-init S›
‹get-trail-l-init (add-to-tautology-init-l C S) = get-trail-l-init S›
‹get-conflict-l-init (add-to-tautology-init-l C S) = get-conflict-l-init S›
‹other-clauses-l-init (add-to-tautology-init-l C S) = other-clauses-l-init S›
‹clauses-to-update-l-init (add-to-tautology-init-l C S) = clauses-to-update-l-init S›
‹literals-to-update-l-init (add-to-tautology-init-l C S) = literals-to-update-l-init S›
‹get-clauses-l-init (add-to-tautology-init-l C S) = get-clauses-l-init S›
‹get-unit-clauses-l-init (add-to-tautology-init-l C S) = (get-unit-clauses-l-init S)›
‹get-learned-unit-clauses-l-init (add-to-tautology-init-l C S) =
get-learned-unit-clauses-l-init S›
‹get-subsumed-learned-clauses-l-init (add-to-tautology-init-l C S) =
get-subsumed-learned-clauses-l-init S›
‹get-subsumed-init-clauses-l-init (add-to-tautology-init-l C S) =
add-mset (remdups-mset (mset C)) (get-subsumed-init-clauses-l-init S)›
‹get-learned-clauses0-l-init (add-to-tautology-init-l C S) = get-learned-clauses0-l-init S›
‹get-init-clauses0-l-init (add-to-tautology-init-l C S) = get-init-clauses0-l-init S›
by (solves ‹cases S; auto simp: add-to-tautology-init-l-def›)+

lemma [twl-st-l-init]:
‹(V, W) ∈ twl-st-l-init ⟹
count-decided (get-trail-init W) = count-decided (get-trail-l-init V)›
by (auto simp: twl-st-l-init-def)

lemma [twl-st-l-init]:
‹(V, W) ∈ twl-st-l-init ⟹get-subsumed-learned-clauses-init W = get-subsumed-learned-clauses-l-init V›
by (cases V, cases W, auto simp: twl-st-l-init-def)

lemma [twl-st-l-init]:
‹get-conflict-l (fst T) = get-conflict-l-init T›
‹literals-to-update-l (fst T) = literals-to-update-l-init T›
‹clauses-to-update-l (fst T) = clauses-to-update-l-init T›
‹get-subsumed-learned-clauses-l (fst T) = get-subsumed-learned-clauses-l-init T›
‹get-subsumed-init-clauses-l (fst T) = get-subsumed-init-clauses-l-init T›
‹get-subsumed-clauses-l (fst T) = get-subsumed-clauses-l-init T›
‹get-conflict-l (fst T) = get-conflict-l-init T›
by (cases T; auto; fail)+

lemma entailed-clss-inv-add-to-unit-init-clauses:
‹count-decided (get-trail-init T) = 0 ⟹ C ≠ [] ⟹ hd C ∈ lits-of-l (get-trail-init T) ⟹
entailed-clss-inv (pstate$_W$-of-init T) ⟹ entailed-clss-inv (pstate$_W$-of-init (add-to-unit-init-clauses (mset C) T))›
using count-decided-ge-get-level[of ‹get-trail-init T›]
by (cases T; cases C; auto simp: twl-st-inv.simps twl-exception-inv.simps entailed-clss-inv-def)

lemma convert-lits-l-no-decision-iff: ‹(S, T) ∈ convert-lits-l M N ⟹
(∀ s∈set T. ¬ is-decided s) ⟷
(∀ s∈set S. ¬ is-decided s)›
unfolding convert-lits-l-def
by (induction rule: list-rel-induct)
(auto simp: dest!: p2relD)

lemma *twl-st-l-init-no-decision-iff*:
  ‹(*S*, *T*) ∈ *twl-st-l-init* ⟹
    (∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*) ⟷
    (∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*)›
  by (*subst convert-lits-l-no-decision-iff*[*of* - - ‹*get-clauses-l-init S*›
    ‹*get-kept-unit-clauses-l-init S*›])
   (*auto simp*: *twl-st-l-init-def*)

lemma *twl-st-l-init-defined-lit*[*twl-st-l-init*]:
  ‹(*S*, *T*) ∈ *twl-st-l-init* ⟹
    defined-lit (*get-trail-init T*) = *defined-lit* (*get-trail-l-init S*)›
  by (*auto simp*: *twl-st-l-init-def*)

lemma [*twl-st-l-init*]:
  ‹(*S*, *T*) ∈ *twl-st-l-init* ⟹ *get-learned-clauses-init T* = {#} ⟷ *learned-clss-l* (*get-clauses-l-init S*) = {#}›
  ‹(*S*, *T*) ∈ *twl-st-l-init* ⟹ *get-unit-learned-clauses-init T* = {#} ⟷ *get-learned-unit-clauses-l-init S* = {#}
   ›
  by (*cases S*; *cases T*; *auto simp*: *twl-st-l-init-def*; *fail*)+

lemma *init-dt-pre-already-propagated-unit-init-l*:
  assumes
    hd-C: ‹*hd C* ∈ *lits-of-l* (*get-trail-l-init S*)› **and**
    pre: ‹*init-dt-pre CS S*› **and**
    nempty: ‹*C* ≠ []› **and**
    dist-C: ‹*distinct C*› **and**
    lev: ‹*count-decided* (*get-trail-l-init S*) = *0*› **and**
    C': ‹*mset* (*remdups C'*) = *mset C*›
  shows
    ‹*init-dt-pre CS* (*already-propagated-unit-init-l* (*mset C*) *S*)› (**is** *?pre*) **and**
    ‹*init-dt-spec* [*C'*] *S* (*already-propagated-unit-init-l* (*mset C*) *S*)› (**is** *?spec*)
proof −
  obtain *T* **where**
    SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
    inv: ‹*twl-struct-invs-init T*› **and**
    WS: ‹*clauses-to-update-l-init S* = {#}› **and**
    dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
    in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
     literals-to-update-l-init S = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init S*)› **and**
    add-inv: ‹*twl-list-invs* (*fst S*)› **and**
    stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
    OC'-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*› **and**
    tauto: ‹∀ *C*∈#*ran-mf* (*get-clauses-l-init S*). ¬ *tautology* (*mset C*)›
    using *pre* **unfolding** *init-dt-pre-def*
    apply −
    apply *normalize-goal+*
    by *presburger*
  obtain *M N D NE UE NEk UEk Q U OC NS US N0 U0* **where**
    S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*)›
    by (*cases S*) *auto*
  have [*simp*]: ‹*twl-list-invs* (*fst* (*already-propagated-unit-init-l* (*mset C*) *S*))›
    using *add-inv* **by** (*auto simp*: *already-propagated-unit-init-l-def S*
      twl-list-invs-def)

have [*simp*]: ‹(*already-propagated-unit-init-l* (*mset C*) *S*, *add-to-unit-init-clauses* (*mset C*) *T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*)
(*auto simp*: *twl-st-l-init-def already-propagated-unit-init-l-def*
convert-lits-l-extend-mono)
have *dec'*: ‹∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*›
using *SOC-T dec* **by** (*subst twl-st-l-init-no-decision-iff*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*))›
using *stgy-inv dec'* **unfolding** *twl-stgy-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def*
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0-def cdcl*$_W$*-restart-mset.no-smaller-confl-def*
by (*cases T*)
(*auto simp*: *cdcl*$_W$*-restart-mset-state clauses-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*add-to-unit-init-clauses* (*mset C*) *T*)›
apply (*rule twl-struct-invs-init-add-to-unit-init-clauses*)
using *inv hd-C nempty dist-C lev SOC-T dec'*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff intro*: *bexI*[*of* - ‹*hd C*›])
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*add-to-unit-init-clauses* (*mset C*) *T*›])
using *WS dec in-literals-to-update OC'-empty tauto* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*add-to-unit-init-clauses* (*mset C*) *T*›])
using *WS dec in-literals-to-update OC'-empty nempty dist-C C'*
by (*auto simp*: *twl-st-init twl-st-l-init*; *fail*)+
qed

lemma *init-dt-pre-add-to-tautology-init-l*:
assumes
pre: ‹*init-dt-pre CS S*› **and**
tautology: ‹*tautology* (*mset C*)› **and**
lev: ‹*count-decided* (*get-trail-l-init S*) = *0*›
shows
‹*init-dt-pre CS* (*add-to-tautology-init-l C S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [*C*] *S* (*add-to-tautology-init-l C S*)› (**is** *?spec*)
proof −
obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC'-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*›**and**
tauto: ‹∀ *C*∈#*ran-mf* (*get-clauses-l-init S*). ¬ *tautology* (*mset C*)›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal*+
by *presburger*
obtain *M N D NE UE NEk UEk Q U OC NS US N0 U0* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*)›
by (*cases S*) *auto*

let *?S* = ‹*add-to-tautology-init-l C S*›
have [*simp*]: ‹*twl-list-invs* (*fst ?S*)›
using *add-inv* **by** (*auto simp*: *add-to-tautology-init-l-def S twl-list-invs-def*)
have [*simp*]: ‹(*add-to-tautology-init-l C S*, *add-to-tautology-init* (*mset C*) *T*) ∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*)
(*auto simp*: *twl-st-l-init-def add-to-tautology-init-l-def add-to-tautology-init-def convert-lits-l-extend-mono*)
have *dec'*: ‹∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*›
using *SOC-T dec* **by** (*subst twl-st-l-init-no-decision-iff*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*add-to-tautology-init* (*mset C*) *T*))›
using *stgy-inv dec'* **unfolding** *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def* $cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def* $cdcl_W$*-restart-mset.no-smaller-confl-def*
by (*cases T*)
(*auto simp*: $cdcl_W$*-restart-mset-state clauses-def add-to-tautology-init-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*add-to-tautology-init* (*mset C*) *T*)›
apply (*rule twl-struct-invs-init-add-to-tautology-init*)
using *inv tautology lev SOC-T dec'*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff intro*: *bexI*[*of* - ‹*hd C*›])
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*add-to-tautology-init* (*mset C*) *T*›])
using *WS dec in-literals-to-update OC'-empty tauto* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*add-to-tautology-init* (*mset C*) *T*›])
using *WS dec in-literals-to-update OC'-empty tautology*
by (*auto simp*: *twl-st-init twl-st-l-init*; *fail*)+
qed

lemma (**in** −) *twl-stgy-invs-backtrack-lvl-0*:
‹*count-decided* (*get-trail T*) = *0* ⟹ *twl-stgy-invs T*›
using *count-decided-ge-get-level*[*of* ‹*get-trail T*›]
by (*cases T*)
(*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def* $cdcl_W$*-restart-mset.no-smaller-confl-def* $cdcl_W$*-restart-mset-state* $cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)

lemma *init-dt-pre-propagate-unit-init*:
assumes
hd-C: ‹*undefined-lit* (*get-trail-l-init S*) *L*› **and**
pre: ‹*init-dt-pre CS S*› **and**
lev: ‹*count-decided* (*get-trail-l-init S*) = *0*› **and**
confl: ‹*get-conflict-l-init S* = *None*› **and**
C': ‹*remdups C'* = [*L*]›
shows
‹*propagate-unit-init-l L S* ≤ *SPEC*(*init-dt-pre CS*)› (**is** *?pre*) **and**
‹*propagate-unit-init-l L S* ≤ *SPEC*(*init-dt-spec* [*C'*] *S*)› (**is** *?spec*)
proof −
obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**

in-literals-to-update: ‹*get-conflict-l-init S = None* ⟶
literals-to-update-l-init S = uminus ‘# lit-of ‘# mset (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC′-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal+*
by *presburger*
obtain *M N D NE UE NEk UEk Q U OC NS US N0 U0* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*)›
by (*cases S*) *auto*
have *1*: ‹*propagate-unit-init-l L S* ≤ *SPEC*(λ*S′*. (*S′*, *propagate-unit-init L T*)
∈ *twl-st-l-init*)›
using *SOC-T assms* **by** (*auto simp*: *twl-st-l-init-def propagate-unit-init-l-def*
convert-lit.simps cons-trail-propagate-l-def S convert-lits-l-extend-mono
intro!: *ASSERT-refine-right ASSERT-leI*)
have *dec′*: ‹∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*›
using *SOC-T dec* **by** (*subst twl-st-l-init-no-decision-iff*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*propagate-unit-init L T*))›
apply (*rule twl-stgy-invs-backtrack-lvl-0*)
using *lev SOC-T*
by (*cases S*) (*auto simp*: *cdcl*$_W$*-restart-mset-state clauses-def twl-st-l-init-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have *2*: ‹*twl-struct-invs-init* (*propagate-unit-init L T*)›
apply (*rule twl-struct-invs-init-propagate-unit-init*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff uminus-lit-of-image-mset*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff uminus-lit-of-image-mset*)
done
have *3*: ‹*propagate-unit-init-l L S* ≤ *SPEC*(λ*S*. *twl-list-invs* (*fst* (*S*)))›
using *add-inv assms*
by (*auto simp*: *S twl-list-invs-def propagate-unit-init-l-def cons-trail-propagate-l-def*
intro!: *ASSERT-leI*)
show *?pre*
using *assms 3 2 1*
unfolding *init-dt-pre-def cons-trail-propagate-l-def*
propagate-unit-init-l-def S
apply (*simp only*: *S get-trail-l-init.simps not-False-eq-True assert.ASSERT-simps*
nres-monad3 nres-monad1 nres-order-simps mem-Collect-eq)
apply (*rule exI*[*of* - ‹*propagate-unit-init L T*›])

using *WS dec in-literals-to-update OC′-empty confl*
by (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
using *assms 1 2 3*
unfolding *init-dt-spec-def cons-trail-propagate-l-def propagate-unit-init-l-def S*
apply (*simp only*: *S get-trail-l-init.simps not-False-eq-True assert.ASSERT-simps*
nres-monad3 nres-monad1 nres-order-simps mem-Collect-eq)
apply (*rule exI*[*of* - ‹*propagate-unit-init L T*›])
using *WS dec in-literals-to-update OC′-empty confl C′*
by (*auto simp*: *twl-st-init twl-st-l-init S*
simp flip: *mset-remdups-remdups-mset*)
qed

lemma [*twl-st-l-init*]:
‹*get-trail-l-init* (*set-conflict-init-l C S*) = *get-trail-l-init S*›
‹*literals-to-update-l-init* (*set-conflict-init-l C S*) = {#}›
‹*clauses-to-update-l-init* (*set-conflict-init-l C S*) = {#}›
‹*get-conflict-l-init* (*set-conflict-init-l C S*) = *Some* (*mset C*)›
‹*get-unit-clauses-l-init* (*set-conflict-init-l C S*) = *add-mset* (*mset C*) (*get-unit-clauses-l-init S*)›
‹*get-subsumed-init-clauses-l-init* (*set-conflict-init-l C S*) = *get-subsumed-init-clauses-l-init S*›
‹*get-subsumed-learned-clauses-l-init* (*set-conflict-init-l C S*) = *get-subsumed-learned-clauses-l-init S*›
‹*get-learned-unit-clauses-l-init* (*set-conflict-init-l C S*) = *get-learned-unit-clauses-l-init S*›
‹*get-learned-clauses0-l-init* (*set-conflict-init-l C S*) = *get-learned-clauses0-l-init S*›
‹*get-init-clauses0-l-init* (*set-conflict-init-l C S*) = *get-init-clauses0-l-init S*›
‹*get-clauses-l-init* (*set-conflict-init-l C S*) = *get-clauses-l-init S*›
‹*other-clauses-l-init* (*set-conflict-init-l C S*) = *other-clauses-l-init S*›
by (*cases S*; *auto simp*: *set-conflict-init-l-def*; *fail*)+

lemma *init-dt-pre-set-conflict-init-l*:
assumes
[*simp*]: ‹*get-conflict-l-init S* = *None*› **and**
pre: ‹*init-dt-pre* (*C′* # *CS*) *S*› **and**
false: ‹∀ *L* ∈ *set C*. −*L* ∈ *lits-of-l* (*get-trail-l-init S*)› **and**
nempty: ‹*C* ≠ []› **and**
C′: ‹*mset* (*remdups C′*) = *mset C*›
shows
‹*init-dt-pre CS* (*set-conflict-init-l C S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [*C′*] *S* (*set-conflict-init-l C S*)› (**is** *?spec*)
proof −
have *dist-C*: ‹*distinct C*›
using *C′* **using** *same-mset-distinct-iff* **by** *blast*
obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC′-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*› **and**
tauto: ‹∀ *C*∈#*ran-mf* (*get-clauses-l-init S*). ¬ *tautology* (*mset C*)›
using *pre C′* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal*+
by *force*

obtain *M N D NE UE NEk UEk Q U OC NS US N0 U0* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*)›
by (*cases S*) *auto*
have [*simp*]: ‹*twl-list-invs* (*fst* (*set-conflict-init-l C S*))›
using *add-inv* **by** (*auto simp*: *set-conflict-init-l-def S*
twl-list-invs-def)
have [*simp*]: ‹(*set-conflict-init-l C S*, *set-conflict-init C T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*) (*auto simp*: *twl-st-l-init-def set-conflict-init-l-def convert-lit.simps*
convert-lits-l-extend-mono)
have *dec'*: ‹*count-decided* (*get-trail-init T*) = *0*›
apply (*subst count-decided-0-iff*)
apply (*subst twl-st-l-init-no-decision-iff*)
using *SOC-T dec SOC-T* **by** (*auto simp*: *twl-st-l-init twl-st-init convert-lits-l-def*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*set-conflict-init C T*))›
using *stgy-inv dec' nempty count-decided-ge-get-level*[*of* ‹*get-trail-init T*›]
unfolding *twl-stgy-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant-def*
cdcl$_W$*-restart-mset.conflict-non-zero-unless-level-0-def cdcl*$_W$*-restart-mset.no-smaller-confl-def*
by (*cases T*; *cases C*)
(*auto 5 5 simp*: *cdcl*$_W$*-restart-mset-state clauses-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*set-conflict-init C T*)›
apply (*rule twl-struct-invs-init-set-conflict-init*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec' false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
done
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*set-conflict-init C T*›])
using *WS dec in-literals-to-update OC'-empty tauto* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*set-conflict-init C T*›])
using *WS dec in-literals-to-update OC'-empty C'* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
qed

lemma [*twl-st-init*]:
‹*get-trail-init* (*add-empty-conflict-init T*) = *get-trail-init T*›
‹*get-conflict-init* (*add-empty-conflict-init T*) = *Some* {#}›
‹*clauses-to-update-init* (*add-empty-conflict-init T*) = *clauses-to-update-init T*›

‹*literals-to-update-init* (*add-empty-conflict-init T*) = {#}›
by (*cases T*; *auto simp*:; *fail*)+

lemma [*twl-st-l-init*]:
‹*get-trail-l-init* (*add-empty-conflict-init-l T*) = *get-trail-l-init T*›
‹*get-conflict-l-init* (*add-empty-conflict-init-l T*) = *Some* {#}›
‹*clauses-to-update-l-init* (*add-empty-conflict-init-l T*) = *clauses-to-update-l-init T*›
‹*literals-to-update-l-init* (*add-empty-conflict-init-l T*) = {#}›
‹*get-unit-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-unit-clauses-l-init T*›
‹*get-subsumed-init-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-subsumed-init-clauses-l-init T*›
‹*get-subsumed-learned-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-subsumed-learned-clauses-l-init T*›
‹*get-learned-unit-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-learned-unit-clauses-l-init T*›
‹*get-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-clauses-l-init T*›
‹*get-init-clauses0-l-init* (*add-empty-conflict-init-l T*) = *add-mset* {#} (*get-init-clauses0-l-init T*)›
‹*get-learned-clauses0-l-init* (*add-empty-conflict-init-l T*) = *get-learned-clauses0-l-init T*›
‹*other-clauses-l-init* (*add-empty-conflict-init-l T*) = (*other-clauses-l-init T*)›
by (*cases T*; *auto simp*: *add-empty-conflict-init-l-def*; *fail*)+

lemma *twl-struct-invs-init-add-empty-conflict-init-l*:
assumes
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-init T* = {#}›
shows ‹*twl-struct-invs-init* (*add-empty-conflict-init T*)›
(**is** *?all-struct*)
proof −
obtain *M N U D NE UE Q OC NS US N0 U0* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*), *OC*)›
using *WS* **by** (*cases T*) *auto*
let *?T* = ‹(*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, *NS*, *US*, *add-mset* {#} *N0*, *U0*, {#}, {#})›
have ‹$cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *D*)› **and**
pinvs: ‹*pcdcl-all-struct-invs* ($pstate_W$-*of-init* ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*), *OC*))›
using *invs* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** *auto*
then have
‹$cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv* (*M*, *add-mset* {#} (*clauses N* + *OC* + *NE* + *NS* + *N0*), *clauses U* + *UE* + *US* + *U0*, *Some* {#})›
unfolding *T twl-struct-invs-init-def*
by (*auto 5 5 simp*: $cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv-def*
$cdcl_W$-*restart-mset.no-strange-atm-def* $cdcl_W$-*restart-mset-state*
$cdcl_W$-*restart-mset*.$cdcl_W$-*M-level-inv-def* $cdcl_W$-*restart-mset*.$cdcl_W$-*conflicting-def*
$cdcl_W$-*restart-mset.distinct-cdcl*$_W$-*state-def all-decomposition-implies-def*
clauses-def $cdcl_W$-*restart-mset*.$cdcl_W$-*learned-clause-alt-def*
true-annots-true-cls-def-iff-negation-in-model)
then have *pinvs*: ‹*pcdcl-all-struct-invs* ($pstate_W$-*of-init* (*?T*, *OC*))›
using *pinvs count-decided-ge-get-level*[*of M*] *lev*
unfolding *pcdcl-all-struct-invs-def*
by (*auto simp*: *T psubsumed-invs-def entailed-clss-inv-def clauses0-inv-def*)
have ‹$cdcl_W$-*restart-mset.no-smaller-propa* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹$cdcl_W$-*restart-mset.no-smaller-propa* (*M*, *add-mset* {#} (*clauses N* + *OC* + *NE* + *NS* + *N0*), *clauses U* + *UE* + *US* + *U0*, *Some* {#})›

using *lev*
by (*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def* $cdcl_W$*-restart-mset-state clauses-def T count-decided-0-iff*)

have [*simp*]: ‹*confl-cands-enqueued ?T*›
‹*propa-cands-enqueued ?T*›
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟹ *twl-st-inv ?T*›
‹⋀*x*. *twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) *x* ⟹ *twl-exception-inv ?T x*›
‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟹ *clauses-to-update-inv ?T*›
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*) ⟹ *past-invs ?T*›
by (*auto simp*: *twl-st-inv.simps twl-exception-inv.simps past-invs.simps*; *fail*)+

show *?all-struct*
using *invs pinvs*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv add-to-other-init.simps* $state_W$*-of-init.simps get-conflict.simps*
by (*clarsimp simp del*: *entailed-clss-inv.simps*)
qed

lemma *init-dt-pre-add-empty-conflict-init-l*:
assumes
confl[*simp*]: ‹*get-conflict-l-init S* = *None*› **and**
pre: ‹*init-dt-pre* ([] # *CS*) *S*›
shows
‹*init-dt-pre CS* (*add-empty-conflict-init-l S*)› (**is** *?pre*)
‹*init-dt-spec* [[]] *S* (*add-empty-conflict-init-l S*)› (**is** *?spec*)
proof −
obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC'-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*› **and**
tauto: ‹∀ *C*∈#*ran-mf* (*get-clauses-l-init S*). ¬ *tautology* (*mset C*)›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal+*
by *force*
obtain *M N D NE UE NEk UEk Q U OC NS US N0 U0* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*)›
by (*cases S*) *auto*
have [*simp*]: ‹*twl-list-invs* (*fst* (*add-empty-conflict-init-l S*))›
using *add-inv* **by** (*auto simp*: *add-empty-conflict-init-l-def S twl-list-invs-def*)
have [*simp*]: ‹(*add-empty-conflict-init-l S*, *add-empty-conflict-init T*) ∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*) (*auto simp*: *twl-st-l-init-def add-empty-conflict-init-l-def*)
have *dec'*: ‹*count-decided* (*get-trail-init T*) = *0*›
apply (*subst count-decided-0-iff*)
apply (*subst twl-st-l-init-no-decision-iff*)

using *SOC-T dec SOC-T* **by** (*auto simp*: *twl-st-l-init twl-st-init convert-lits-l-def*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*add-empty-conflict-init T*))›
using *stgy-inv dec′ count-decided-ge-get-level*[*of* ‹*get-trail-init T*›]
unfolding *twl-stgy-invs-def cdcl$_W$-restart-mset.cdcl$_W$-stgy-invariant-def*
cdcl$_W$-restart-mset.conflict-non-zero-unless-level-0-def cdcl$_W$-restart-mset.no-smaller-confl-def
by (*cases T*)
(*auto 5 5 simp*: *cdcl$_W$-restart-mset-state clauses-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*add-empty-conflict-init T*)›
apply (*rule twl-struct-invs-init-add-empty-conflict-init-l*)
using *inv SOC-T dec′ WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*add-empty-conflict-init T*›])
using *WS dec in-literals-to-update OC′-empty tauto* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*add-empty-conflict-init T*›])
using *WS dec in-literals-to-update OC′-empty* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
qed

lemma [*twl-st-l-init*]:
‹*get-trail* (*fst* (*add-to-clauses-init a T*)) = *get-trail-init T*›
by (*cases T*; *auto*; *fail*)

lemma [*twl-st-l-init*]:
‹*other-clauses-l-init* (*T*, *OC*) = *OC*›
‹*clauses-to-update-l-init* (*T*, *OC*) = *clauses-to-update-l T*›
by (*cases T*; *auto*; *fail*)+

lemma *twl-struct-invs-init-add-to-clauses-init*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
confl: ‹*get-conflict-init T* = *None*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
dist-C: ‹*distinct C*› **and**
le-2: ‹*length C* ≥ *2*›
shows
‹*twl-struct-invs-init* (*add-to-clauses-init C T*)›
(**is** *?all-struct*)
proof −
obtain *M N U NE UE OC WS NS US N0 U0* **where**
T: ‹*T* = ((*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *uminus* ‘# *lit-of* ‘# *mset M*), *OC*)›
using *confl MQ* **by** (*cases T*) *auto*
let *?Q* = ‹*uminus* ‘# *lit-of* ‘# *mset M*›

let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *WS*, *?Q*)›
have [*simp*]: ‹*get-all-ann-decomposition M* = [([], *M*)]›
by (*rule no-decision-get-all-ann-decomposition*) (*use lev* **in** ‹*auto simp*: *T count-decided-0-iff*›)
have ‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv* (*M*, (*clauses N* + *OC* + *NE* + *NS* + *N0*), *clauses U* + *UE* + *US* + *U0*, *None*)› **and**
excep: ‹*twl-st-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *?Q*)› **and**
st-inv: ‹*twl-st-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *?Q*)› **and**

pinvs: ‹*pcdcl-all-struct-invs*
(*pstate*$_W$ *-of-init* ((*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *uminus* ‘# *lit-of* ‘# *mset M*), *OC*))›
using *invs confl* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def* **by** *auto*
then have
‹*cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-all-struct-inv* (*M*, *add-mset* (*mset C*) (*clauses N* + *OC* + *NE* + *NS* + *N0*),
clauses U + *UE* + *US* + *U0*, *None*)› **and**
n-d: ‹*no-dup M*›
using *dist-C*
by (*auto simp*: *cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-all-struct-inv-def*
cdcl$_W$ *-restart-mset.no-strange-atm-def cdcl*$_W$ *-restart-mset-state*
cdcl$_W$ *-restart-mset.cdcl*$_W$ *-M-level-inv-def cdcl*$_W$ *-restart-mset.cdcl*$_W$ *-conflicting-def*
cdcl$_W$ *-restart-mset.distinct-cdcl*$_W$ *-state-def all-decomposition-implies-def*
clauses-def cdcl$_W$ *-restart-mset.cdcl*$_W$ *-learned-clause-alt-def*)
then have *pinvs'*: ‹*pcdcl-all-struct-invs* (*pstate*$_W$ *-of-init* (*add-to-clauses-init C T*))›
using *invs* **unfolding** *T twl-struct-invs-init-def pcdcl-all-struct-invs-def*
by (*auto simp*: *entailed-clss-inv-def psubsumed-invs-def mset-take-mset-drop-mset′ clauses0-inv-def*)
have ‹*cdcl*$_W$ *-restart-mset.no-smaller-propa* (*M*, *clauses N* + *OC* + *NE* + *NS* + *N0*, *clauses U* + *UE* + *US* + *U0*, *None*)›
using *invs confl* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹*cdcl*$_W$ *-restart-mset.no-smaller-propa* (*M*, *add-mset* (*mset C*) (*clauses N* + *OC* + *NE* + *NS* + *N0*),
clauses U + *UE* + *US* + *U0*, *None*)›
using *lev*
by (*auto simp*: *cdcl*$_W$ *-restart-mset.no-smaller-propa-def cdcl*$_W$ *-restart-mset-state*
clauses-def T count-decided-0-iff)

have *struct*: ‹*struct-wf-twl-cls C*› **if** ‹*C* ∈# *N* + *U*› **for** *C*
using *st-inv that* **by** (*simp add*: *twl-st-inv.simps*)

show ‹*twl-struct-invs-init* (*add-to-clauses-init C T*)›
apply (*rule twl-struct-invs-init-init-state*)
subgoal using *lev* **by** (*auto simp*: *T*)
subgoal using *struct dist-C le-2* **by** (*auto simp*: *T mset-take-mset-drop-mset′*)
subgoal using *MQ* **by** (*auto simp*: *T*)
subgoal using *WS* **by** (*auto simp*: *T*)
subgoal using *pinvs'* **by** (*simp add*: *T mset-take-mset-drop-mset′*)
subgoal by (*auto simp*: *T mset-take-mset-drop-mset′*)
subgoal by (*auto simp*: *T*)
done
qed

lemma *get-trail-init-add-to-clauses-init*[*simp*]:
‹*get-trail-init* (*add-to-clauses-init a T*) = *get-trail-init T*›
by (*cases T*) *auto*

lemma *init-dt-pre-add-to-clauses-init-l*:
assumes
D: ‹*get-conflict-l-init S* = *None*› **and**
a: ‹*length a* ≠ *Suc 0*› ‹*a* ≠ []› **and**
pre: ‹*init-dt-pre* (*a′* # *CS*) *S*› **and**
‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
C′: ‹*mset* (*remdups a′*) = *mset a*› **and**
not-tauto: ‹¬*tautology* (*mset a′*)›

```
  shows
    ‹add-to-clauses-init-l a S ≤ SPEC (init-dt-pre CS)› (is ?pre) and
    ‹add-to-clauses-init-l a S ≤ SPEC (init-dt-spec [a'] S)› (is ?spec)
proof −
  have not-tauto': ‹¬tautology (mset a)›
    by (metis C' consistent-interp-tautology not-tauto set-remdups)
  obtain T where
    SOC-T: ‹(S, T) ∈ twl-st-l-init› and
    inv: ‹twl-struct-invs-init T› and
    WS: ‹clauses-to-update-l-init S = {#}› and
    dec: ‹∀ s∈set (get-trail-l-init S). ¬ is-decided s› and
    in-literals-to-update: ‹get-conflict-l-init S = None ⟶
     literals-to-update-l-init S = uminus `# lit-of `# mset (get-trail-l-init S)› and
    add-inv: ‹twl-list-invs (fst S)› and
    stgy-inv: ‹twl-stgy-invs (fst T)› and
    OC'-empty: ‹other-clauses-l-init S ≠ {#} ⟶ get-conflict-l-init S ≠ None› and
    tauto: ‹∀ C∈#ran-mf (get-clauses-l-init S). ¬ tautology (mset C)›
    using pre unfolding init-dt-pre-def
    apply −
    apply normalize-goal+
    by force
  have dec': ‹∀ L ∈ set (get-trail-init T). ¬is-decided L›
    using SOC-T dec apply −
    apply (rule twl-st-l-init-no-decision-iff[THEN iffD2])
    using SOC-T dec SOC-T by (auto simp: twl-st-l-init twl-st-init convert-lits-l-def)
  obtain M N NE UE NEk UEk Q OC NS US N0 U0 where
    S: ‹S = ((M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), OC)›
    using D WS by (cases S) auto
  have le-2: ‹length a ≥ 2›
    using a by (cases a) auto
  have
    ‹init-dt-pre CS ((M, fmupd i (a, True) N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q),
OC)› (is ?pre1) and
    ‹init-dt-spec [a'] S
        ((M, fmupd i (a, True) N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), OC)› (is
?spec1)
    if
      i-0: ‹0 < i› and
      i-dom: ‹i ∉# dom-m N›
    for i :: ‹nat›
  proof −
    let ?S = ‹((M, fmupd i (a, True) N, None, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), OC)›
    have ‹Propagated L i ∉ set M› for L
      using add-inv i-dom i-0 unfolding S
      by (auto simp: twl-list-invs-def)
    then have ‹(?S, add-to-clauses-init a T) ∈ twl-st-l-init›
      using SOC-T i-dom
      by (auto simp: S twl-st-l-init-def init-clss-l-mapsto-upd-notin
          learned-clss-l-mapsto-upd-notin-irrelev convert-lit.simps
          intro!: convert-lits-l-extend-mono[of - - N ‹NEk+UEk› ‹fmupd i (a, True) N›])
    moreover have ‹twl-struct-invs-init (add-to-clauses-init a T)›
      apply (rule twl-struct-invs-init-add-to-clauses-init)
      subgoal
        apply (subst count-decided-0-iff)
        apply (subst twl-st-l-init-no-decision-iff)
        using SOC-T dec SOC-T by (auto simp: twl-st-l-init twl-st-init convert-lits-l-def)
```

subgoal by (*use dec SOC-T in-literals-to-update* **in**
⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv*⟩)
subgoal by (*use dec SOC-T in-literals-to-update* **in**
⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv*⟩)
subgoal by (*use dec SOC-T in-literals-to-update* **in**
⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv*⟩)
subgoal by (*use dec SOC-T in-literals-to-update* **in**
⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv*⟩)
subgoal using *C′ same-mset-distinct-iff* **by** *blast*
subgoal by (*use dec SOC-T in-literals-to-update* **in**
⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv*⟩)
done
moreover have ⟨*twl-list-invs* (*M*, *fmupd i* (*a*, *True*) *N*, *None*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)⟩
using *add-inv i-dom i-0 not-tauto′* **by** (*auto simp*: *S twl-list-invs-def ran-m-mapsto-upd-notin*)
moreover have ⟨*twl-stgy-invs* (*fst* (*add-to-clauses-init a T*))⟩
by (*rule twl-stgy-invs-backtrack-lvl-0*)
(*use dec′ SOC-T* **in** ⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init twl-st-l-init-def*⟩)
ultimately show *?pre1 ?spec1*
unfolding *init-dt-pre-def init-dt-spec-def* **apply** −
subgoal
apply (*rule exI*[*of* - ⟨*add-to-clauses-init a T*⟩])
using *dec OC′-empty in-literals-to-update tauto not-tauto′* **by** (*auto simp*: *S ran-m-mapsto-upd-notin i-dom*)
subgoal
apply (*rule exI*[*of* - ⟨*add-to-clauses-init a T*⟩])
using *dec OC′-empty in-literals-to-update i-dom i-0 a C′*
by (*auto simp*: *S learned-clss-l-mapsto-upd-notin-irrelev ran-m-mapsto-upd-notin remdups-mset-idem*)
done
qed
then show *?pre ?spec*
by (*auto simp*: *S add-to-clauses-init-l-def get-fresh-index-def RES-RETURN-RES*)
qed

lemma *init-dt-pre-init-dt-step*:
assumes *pre*: ⟨*init-dt-pre* (*a* # *CS*) *SOC*⟩
shows ⟨*init-dt-step a SOC* ≤ *SPEC* (λ*SOC′*. *init-dt-pre CS SOC′* ∧ *init-dt-spec* [*a*] *SOC SOC′*)⟩
proof −
obtain *S OC* **where** *SOC*: ⟨*SOC* = (*S*, *OC*)⟩
by (*cases SOC*) *auto*
obtain *T* **where**
SOC-T: ⟨((*S*, *OC*), *T*) ∈ *twl-st-l-init*⟩ **and**
inv: ⟨*twl-struct-invs-init T*⟩ **and**
WS: ⟨*clauses-to-update-l-init* (*S*, *OC*) = {#}⟩ **and**
dec: ⟨∀ *s*∈*set* (*get-trail-l-init* (*S*, *OC*)). ¬ *is-decided s*⟩ **and**
in-literals-to-update: ⟨*get-conflict-l-init* (*S*, *OC*) = *None* ⟶
literals-to-update-l-init (*S*, *OC*) = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init* (*S*, *OC*))⟩ **and**
add-inv: ⟨*twl-list-invs* (*fst* (*S*, *OC*))⟩ **and**
stgy-inv: ⟨*twl-stgy-invs* (*fst T*)⟩ **and**
OC′-empty: ⟨*other-clauses-l-init* (*S*, *OC*) ≠ {#} ⟶ *get-conflict-l-init* (*S*, *OC*) ≠ *None*⟩ **and**
tauto: ⟨∀ *C*∈#*ran-mf* (*get-clauses-l-init* (*S*, *OC*)). ¬ *tautology* (*mset C*)⟩
using *pre* **unfolding** *SOC init-dt-pre-def*
apply −
apply *normalize-goal*+

```
    by presburger
  have dec': ‹∀ s∈set (get-trail-init T). ¬ is-decided s›
    using SOC-T dec by (rule twl-st-l-init-no-decision-iff[THEN iffD2])

  obtain M N D NE UE NEk UEk NS US N0 U0 Q where
    S: ‹SOC = ((M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), OC)›
    using WS by (cases SOC) (auto simp: SOC)
  then have S': ‹S = (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q)›
    using S unfolding SOC by auto
  show ?thesis
  proof (cases ‹get-conflict-l (fst SOC)›)
    case None
    then show ?thesis
      using pre dec
      unfolding init-dt-step-def remdups-clause-def
      by refine-vcg
        (auto simp add: Let-def count-decided-0-iff SOC twl-st-l-init twl-st-init
          remdups-clause-def
          true-annot-iff-decided-or-true-lit length-list-Suc-0
          init-dt-step-def get-fresh-index-def RES-RETURN-RES
          intro: init-dt-pre-already-propagated-unit-init-l init-dt-pre-set-conflict-init-l
            init-dt-pre-propagate-unit-init init-dt-pre-add-empty-conflict-init-l
            init-dt-pre-add-to-clauses-init-l init-dt-pre-add-to-tautology-init-l
          intro!: SPEC-rule-conjI
          dest: init-dt-pre-ConsD in-lits-of-l-defined-litD tautology-length-ge2
          simp flip: mset-remdups-remdups-mset)
  next
    case (Some D')
    then have [simp]: ‹D = Some D'›
      by (auto simp: S)
    have [simp]:
      ‹(((M, N, Some D', NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), add-mset (remdups-mset
(mset a)) OC),
         add-to-other-init a T)
        ∈ twl-st-l-init› for a
      using SOC-T by (cases T; auto simp: S S' twl-st-l-init-def; fail)+
    have ‹init-dt-pre CS ((M, N, Some D', NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), add-mset
(remdups-mset (mset a)) OC)›
      unfolding init-dt-pre-def
      apply (rule exI[of - ‹add-to-other-init (a) T›])
      using inv WS dec' dec in-literals-to-update add-inv stgy-inv SOC-T tauto
      by (auto simp: S' count-decided-0-iff twl-st-init
          intro!: twl-struct-invs-init-add-to-other-init)
    moreover have ‹init-dt-spec [a] ((M, N, Some D', NE, UE, NEk, UEk, NS, US, N0, U0, {#},
Q), OC)
      ((M, N, Some D', NE, UE, NEk, UEk, NS, US, N0, U0, {#}, Q), add-mset (remdups-mset (mset
a)) OC)›
      unfolding init-dt-spec-def
      apply (rule exI[of - ‹add-to-other-init (a) T›])
      using inv WS dec dec' in-literals-to-update add-inv stgy-inv SOC-T
      by (auto simp: S' count-decided-0-iff twl-st-init
          intro!: twl-struct-invs-init-add-to-other-init)
    ultimately show ?thesis
      by (auto simp: S init-dt-step-def)
  qed
qed
```

lemma [*twl-st-l-init*]:
  ‹*get-trail-l-init* (*S*, *OC*) = *get-trail-l S*›
  ‹*literals-to-update-l-init* (*S*, *OC*) = *literals-to-update-l S*›
  by (*cases S*; *auto*; *fail*)+

lemma *init-dt-spec-append*:
  assumes
    spec1: ‹*init-dt-spec CS S T*› **and**
    spec: ‹*init-dt-spec CS′ T U*›
  shows ‹*init-dt-spec* (*CS* @ *CS′*) *S U*›
proof −
  obtain *T′* **where**
    TT′: ‹(*T*, *T′*) ∈ *twl-st-l-init*› **and**
    ‹*twl-struct-invs-init T′*› **and**
    ‹*clauses-to-update-l-init T* = {#}› **and**
    ‹∀ *s*∈*set* (*get-trail-l-init T*). ¬ *is-decided s*› **and**
    ‹*get-conflict-l-init T* = *None* ⟶
     literals-to-update-l-init T = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init T*)› **and**
   clss: ‹*remdups-mset* ‘# *mset* ‘# *mset CS* + *mset* ‘# *ran-mf* (*get-clauses-l-init S*) + *other-clauses-l-init S* +
        get-unit-clauses-l-init S +
        get-subsumed-init-clauses-l-init S +
        get-init-clauses0-l-init S =
        mset ‘# *ran-mf* (*get-clauses-l-init T*) + *other-clauses-l-init T* + *get-unit-clauses-l-init T* +
        get-subsumed-init-clauses-l-init T +
        get-init-clauses0-l-init T› **and**
    learned: ‹*learned-clss-lf* (*get-clauses-l-init S*) = *learned-clss-lf* (*get-clauses-l-init T*)› **and**
    unit-le:
      ‹*get-subsumed-learned-clauses-l-init S* = *get-subsumed-learned-clauses-l-init T*›
      ‹*get-learned-clauses0-l-init S* = *get-learned-clauses0-l-init T*›
      ‹*get-learned-unit-clauses-l-init S* = *get-learned-unit-clauses-l-init T*› **and**
    ‹*twl-list-invs* (*fst T*)› **and**
    ‹*twl-stgy-invs* (*fst T′*)› **and**
    ‹*other-clauses-l-init T* ≠ {#} ⟶ *get-conflict-l-init T* ≠ *None*› **and**
    empty: ‹{#} ∈# *mset* ‘# *mset CS* ⟶ *get-conflict-l-init T* ≠ *None*› **and**
    confl-kept: ‹*get-conflict-l-init S* ≠ *None* ⟶ *get-conflict-l-init S* = *get-conflict-l-init T*›
    using *spec1*
    unfolding *init-dt-spec-def* **apply** −
    apply *normalize-goal*+
    by *presburger*

  obtain *U′* **where**
    UU′: ‹(*U*, *U′*) ∈ *twl-st-l-init*› **and**
    struct-invs: ‹*twl-struct-invs-init U′*› **and**
    WS: ‹*clauses-to-update-l-init U* = {#}› **and**
    dec: ‹∀ *s*∈*set* (*get-trail-l-init U*). ¬ *is-decided s*› **and**
    confl: ‹*get-conflict-l-init U* = *None* ⟶
     literals-to-update-l-init U = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init U*)› **and**
   clss′: ‹*remdups-mset* ‘# *mset* ‘# *mset CS′* + *mset* ‘# *ran-mf* (*get-clauses-l-init T*) + *other-clauses-l-init T* +
     get-unit-clauses-l-init T + *get-subsumed-init-clauses-l-init T* + *get-init-clauses0-l-init T* =
     mset ‘# *ran-mf* (*get-clauses-l-init U*) + *other-clauses-l-init U* + *get-unit-clauses-l-init U* +
     get-subsumed-init-clauses-l-init U + *get-init-clauses0-l-init U*› **and**
    learned′: ‹*learned-clss-lf* (*get-clauses-l-init T*) = *learned-clss-lf* (*get-clauses-l-init U*)› **and**
    unit-le′: ‹*get-learned-unit-clauses-l-init U* = *get-learned-unit-clauses-l-init T*›

```
    ‹get-subsumed-learned-clauses-l-init U = get-subsumed-learned-clauses-l-init T›
    ‹get-learned-clauses0-l-init U = get-learned-clauses0-l-init T› and
  list-invs: ‹twl-list-invs (fst U)› and
  stgy-invs: ‹twl-stgy-invs (fst U')› and
  oth: ‹other-clauses-l-init U ≠ {#} ⟶ get-conflict-l-init U ≠ None› and
  empty': ‹{#} ∈# mset `# mset CS' ⟶ get-conflict-l-init U ≠ None› and
  confl-kept': ‹get-conflict-l-init T ≠ None ⟶ get-conflict-l-init T = get-conflict-l-init U›
  using spec
  unfolding init-dt-spec-def apply −
  apply normalize-goal+
  by metis
 have ‹remdups-mset `# mset `# mset (CS @ CS') + mset `# ran-mf (get-clauses-l-init S) +
other-clauses-l-init S +
  get-unit-clauses-l-init S + get-subsumed-init-clauses-l-init S + get-init-clauses0-l-init S =
   remdups-mset `# mset `# mset CS' + (remdups-mset `# mset `# mset CS + mset `# ran-mf
(get-clauses-l-init S) + other-clauses-l-init S +
  get-unit-clauses-l-init S + get-subsumed-init-clauses-l-init S + get-init-clauses0-l-init S)›
  by auto
 also have ‹... = remdups-mset `# mset `# mset CS' + mset `# ran-mf (get-clauses-l-init T) +
other-clauses-l-init T +
    get-unit-clauses-l-init T + get-subsumed-init-clauses-l-init T + get-init-clauses0-l-init T›
  unfolding clss by (auto simp: ac-simps)
 finally have eq: ‹remdups-mset `# mset `# mset (CS @ CS') + mset `# ran-mf (get-clauses-l-init
S) + other-clauses-l-init S +
  get-unit-clauses-l-init S +
  get-subsumed-init-clauses-l-init S +
  get-init-clauses0-l-init S =
  mset `# ran-mf (get-clauses-l-init U) + other-clauses-l-init U + get-unit-clauses-l-init U +
  get-subsumed-init-clauses-l-init U +
  get-init-clauses0-l-init U›
  unfolding clss' .
 show ?thesis
  unfolding init-dt-spec-def apply −
  apply (rule exI[of - U'])
  apply (intro conjI)
  subgoal using UU' .
  subgoal using struct-invs .
  subgoal using WS .
  subgoal using dec .
  subgoal using confl .
  subgoal using eq .
  subgoal using learned' learned by simp
  subgoal using unit-le unit-le' by simp
  subgoal using unit-le unit-le' learned by auto
  subgoal using unit-le unit-le' learned by auto
  subgoal using list-invs .
  subgoal using stgy-invs .
  subgoal using oth .
  subgoal using empty empty' oth confl-kept' by auto
  subgoal using confl-kept confl-kept' by auto
  done
qed

lemma init-dt-full:
 fixes CS :: ‹'v literal list list› and SOC :: ‹'v twl-st-l-init› and S'
 defines
```

```
    ‹S ≡ fst SOC› and
    ‹OC ≡ snd SOC›
  assumes
    ‹init-dt-pre CS SOC›
  shows
    ‹init-dt CS SOC ≤ SPEC (init-dt-spec CS SOC)›
  using assms unfolding S-def OC-def
proof (induction CS arbitrary: SOC)
  case Nil
  then obtain S OC where SOC: ‹SOC = (S, OC)›
    by (cases SOC) auto
  from Nil
  obtain T where
    T: ‹(SOC, T) ∈ twl-st-l-init›
      ‹twl-struct-invs-init T›
      ‹clauses-to-update-l-init SOC = {#}›
      ‹∀ s∈set (get-trail-l-init SOC). ¬ is-decided s›
      ‹get-conflict-l-init SOC = None ⟶
       literals-to-update-l-init SOC =
       uminus `# lit-of `# mset (get-trail-l-init SOC)›
      ‹twl-list-invs (fst SOC)›
      ‹twl-stgy-invs (fst T)›
      ‹other-clauses-l-init SOC ≠ {#} ⟶ get-conflict-l-init SOC ≠ None›
    unfolding init-dt-pre-def apply −
    apply normalize-goal+
    by auto

  then show ?case
    unfolding init-dt-def SOC init-dt-spec-def nfoldli-simps
    apply (intro RETURN-rule)
    unfolding prod.simps
    apply (rule exI[of - T])
    using T by (auto simp: SOC twl-st-init twl-st-l-init)
next
  case (Cons a CS) note IH = this(1) and pre = this(2)
  note init-dt-step-def[simp]
  have 1: ‹init-dt-step a SOC ≤ SPEC (λSOC'. init-dt-pre CS SOC' ∧ init-dt-spec [a] SOC SOC')›
    by (rule init-dt-pre-init-dt-step[OF pre])
  have 2: ‹init-dt-spec (a # CS) SOC UOC›
    if spec: ‹init-dt-spec CS T UOC› and
       spec': ‹init-dt-spec [a] SOC T› for T UOC
    using init-dt-spec-append[OF spec' spec] by simp
  show ?case
    unfolding init-dt-def nfoldli-simps if-True
    apply (rule specify-left)
     apply (rule 1)
    apply (rule order.trans)
    unfolding init-dt-def[symmetric]
     apply (rule IH)
     apply (solves ‹simp›)
    apply (rule SPEC-rule)
    by (rule 2) fast+
qed

lemma init-dt-pre-empty-state:
  ‹init-dt-pre [] (([], fmempty, None, {#}, {#}, {#}, {#}, {#}, {#}, {#},{#}, {#},{#}), {#})›
```

unfolding *init-dt-pre-def*
by (*auto simp*: *twl-st-l-init-def twl-struct-invs-init-def twl-st-inv.simps*
twl-struct-invs-def twl-st-inv.simps $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def* $cdcl_W$*-restart-mset.no-smaller-propa-def*
past-invs.simps clauses-def pcdcl-all-struct-invs-def clauses0-inv-def
$cdcl_W$*-restart-mset-state twl-list-invs-def psubsumed-invs-def*
twl-stgy-invs-def $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.no-smaller-confl-def entailed-clss-inv-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)

lemma *twl-init-invs*:
‹*twl-struct-invs-init* (([], {#}, {#}, *None*, {#}, {#}, {#}, {#}, {#},{#}, {#},{#}), {#})›
‹*twl-list-invs* ([], *fmempty*, *None*, {#}, {#}, {#}, {#}, {#}, {#}, {#},{#}, {#},{#})›
‹*twl-stgy-invs* ([], {#}, {#}, *None*, {#}, {#}, {#}, {#}, {#}, {#}, {#},{#})›
by (*auto simp*: *twl-struct-invs-init-def twl-st-inv.simps twl-list-invs-def twl-stgy-invs-def*
past-invs.simps
twl-struct-invs-def twl-st-inv.simps $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def* $cdcl_W$*-restart-mset.no-smaller-propa-def*
past-invs.simps clauses-def pcdcl-all-struct-invs-def clauses0-inv-def
$cdcl_W$*-restart-mset-state twl-list-invs-def psubsumed-invs-def*
twl-stgy-invs-def $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.no-smaller-confl-def entailed-clss-inv-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)
end
theory *Watched-Literals-Watch-List-Initialisation*
imports *Watched-Literals-Watch-List Watched-Literals-Initialisation*
begin

7.0.1 Initialisation

type-synonym *'v twl-st-wl-init'* = ‹((*'v*, *nat*) *ann-lits* × *'v clauses-l* ×
'v cconflict × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* × *'v clauses* ×
'v clauses × *'v clauses* ×*'v lit-queue-wl*)›

type-synonym *'v twl-st-wl-init* = ‹*'v twl-st-wl-init'* × *'v clauses*›
type-synonym *'v twl-st-wl-init-full* = ‹*'v twl-st-wl* × *'v clauses*›

fun *get-trail-init-wl* :: ‹*'v twl-st-wl-init* ⇒ (*'v*, *nat*) *ann-lit list*› **where**
‹*get-trail-init-wl* ((*M*, -, -, -, -, -, -, -), -) = *M*›

fun *get-clauses-init-wl* :: ‹*'v twl-st-wl-init* ⇒ *'v clauses-l*› **where**
‹*get-clauses-init-wl* ((-, *N*, -, -, -, -, -, -), *OC*) = *N*›

fun *get-conflict-init-wl* :: ‹*'v twl-st-wl-init* ⇒ *'v cconflict*› **where**
‹*get-conflict-init-wl* ((-, -, *D*, -, -, -, -, -), -) = *D*›

fun *literals-to-update-init-wl* :: ‹*'v twl-st-wl-init* ⇒ *'v clause*› **where**
‹*literals-to-update-init-wl* ((-, -, -, -, -, -, -, *NS*, *US*, -, -, *Q*), -) = *Q*›

fun *other-clauses-init-wl* :: ‹*'v twl-st-wl-init* ⇒ *'v clauses*› **where**
‹*other-clauses-init-wl* ((-, -, -, -, -, -), *OC*) = *OC*›

fun *add-empty-conflict-init-wl* :: ‹*'v twl-st-wl-init* ⇒ *'v twl-st-wl-init*› **where**
add-empty-conflict-init-wl-def[*simp del*]:
‹*add-empty-conflict-init-wl* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*) =
((*M*, *N*, *Some* {#}, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *add-mset* {#} *N0*, *U0*, {#}), *OC*)›

fun *propagate-unit-init-wl* :: ‹*'v literal* ⇒ *'v twl-st-wl-init* ⇒ (*'v twl-st-wl-init*) *nres*› **where**
propagate-unit-init-wl-def[*simp del*]:
‹*propagate-unit-init-wl L* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*) = *do* {
M ← *cons-trail-propagate-l L 0 M*;
RETURN ((*M*, *N*, *D*, *NE*, *UE*, *add-mset* {#*L*#} *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *add-mset* (−*L*) *Q*),
OC)}›

fun *already-propagated-unit-init-wl* :: ‹*'v clause* ⇒ *'v twl-st-wl-init* ⇒ *'v twl-st-wl-init*› **where**
already-propagated-unit-init-wl-def[*simp del*]:
‹*already-propagated-unit-init-wl C* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *Q*), *OC*) =
((*M*, *N*, *D*, *NE*, *UE*, *add-mset C NEk*, *UEk*, *Q*), *OC*)›

fun *set-conflict-init-wl* :: ‹*'v literal* ⇒ *'v twl-st-wl-init* ⇒ *'v twl-st-wl-init*› **where**
set-conflict-init-wl-def[*simp del*]:
‹*set-conflict-init-wl L* ((*M*, *N*, -, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*) =
((*M*, *N*, *Some* {#*L*#}, *add-mset* {#*L*#} *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}), *OC*)›

fun *add-to-tautology-init-wl* :: ‹*'v clause-l* ⇒ *'v twl-st-wl-init* ⇒ *'v twl-st-wl-init*› **where**
add-to-tautology-init-wl[*simp del*]:
‹*add-to-tautology-init-wl C* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*), *OC*) =
((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *add-mset* (*remdups-mset* (*mset C*)) *NS*, *US*, *N0*, *U0*, *Q*), *OC*)›

fun *add-to-clauses-init-wl* :: ‹*'v clause-l* ⇒ *'v twl-st-wl-init* ⇒ *'v twl-st-wl-init nres*› **where**
add-to-clauses-init-wl-def[*simp del*]:
‹*add-to-clauses-init-wl C* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *Q*), *OC*) = *do* {
i ← *get-fresh-index N*;
let b = (*length C* = *2*);
RETURN ((*M*, *fmupd i* (*C*, *True*) *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *Q*), *OC*)
}›

definition *init-dt-step-wl* :: ‹*'v clause-l* ⇒ *'v twl-st-wl-init* ⇒ *'v twl-st-wl-init nres*› **where**
‹*init-dt-step-wl C S* =
(*case get-conflict-init-wl S of*
None ⇒
if tautology (*mset C*)
then RETURN (*add-to-tautology-init-wl C S*)
else
do {
C ← *remdups-clause C*;
if length C = *0*
then RETURN (*add-empty-conflict-init-wl S*)
else if length C = *1*
then
let L = *hd C in*
if undefined-lit (*get-trail-init-wl S*) *L*
then propagate-unit-init-wl L S
else if L ∈ *lits-of-l* (*get-trail-init-wl S*)

```
      then RETURN (already-propagated-unit-init-wl (mset C) S)
      else RETURN (set-conflict-init-wl L S)
    else add-to-clauses-init-wl C S
  }
| Some D ⇒
    RETURN (add-to-other-init C S))›

fun st-l-of-wl-init :: ‹'v twl-st-wl-init' ⇒ 'v twl-st-l› where
  ‹st-l-of-wl-init (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q) = (M, N, D, NE, UE, NEk,
UEk, NS, US, N0, U0, {#}, Q)›

definition state-wl-l-init' where
  ‹state-wl-l-init' = {(S ,S'). S' = st-l-of-wl-init S}›

definition init-dt-wl :: ‹'v clause-l list ⇒ 'v twl-st-wl-init ⇒ 'v twl-st-wl-init nres› where
  ‹init-dt-wl CS = nfoldli CS (λ-. True) init-dt-step-wl›

definition state-wl-l-init :: ‹('v twl-st-wl-init × 'v twl-st-l-init) set› where
  ‹state-wl-l-init = {(S, S'). (fst S, fst S') ∈ state-wl-l-init' ∧
    other-clauses-init-wl S = other-clauses-l-init S'}›


fun all-blits-are-in-problem-init where
  [simp del]: ‹all-blits-are-in-problem-init (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) ⟷
    (∀ L. (∀ (i, K, b)∈#mset (W L). K ∈# all-lits-of-mm (mset `# ran-mf N + (NE + UE) + (NEk
+ UEk) + (NS + US) + (N0 + U0))))›
```

We assume that no clause has been deleted during initialisation. The definition is slightly redundant since $i \in\# \mathit{dom\text{-}m}\ N$ is already entailed by $\mathit{fst}\ `\#\ \mathit{mset}\ (W\ L) = \mathit{clause\text{-}to\text{-}update}\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\})$.

```
named-theorems twl-st-wl-init

lemma [twl-st-wl-init]:
  assumes ‹(S, S') ∈ state-wl-l-init›
  shows
    ‹get-conflict-l-init S' = get-conflict-init-wl S›
    ‹get-trail-l-init S' = get-trail-init-wl S›
    ‹other-clauses-l-init S' = other-clauses-init-wl S›
    ‹count-decided (get-trail-l-init S') = count-decided (get-trail-init-wl S)›
  using assms
  by (solves ‹cases S; cases S'; auto simp: state-wl-l-init-def state-wl-l-def
    state-wl-l-init'-def›)+

lemma in-clause-to-update-in-dom-mD:
  ‹bb ∈# clause-to-update L (a, aa, ab, ac, ad, NEk, UEk, NS, US, N0, U0, {#}, {#}) ⟹ bb ∈#
dom-m aa›
  unfolding clause-to-update-def
  by force

lemma init-dt-step-wl-init-dt-step:
  assumes S-S': ‹(S, S') ∈ state-wl-l-init›
  shows ‹init-dt-step-wl C S ≤ ⇓ state-wl-l-init
        (init-dt-step C S')›
   (is ‹- ≤ ⇓ ?A -›)
proof −
```

define *remdups-clause'* :: ‹*'a clause-l* ⇒ *'a clause-l nres*› **where**
‹*remdups-clause' C* = *remdups-clause C*› **for** *C* :: ‹*'a clause-l*›

have *remdups-clause*: ‹*remdups-clause' C* ≤⇓ {(*D*,*E*). *D* = *E* ∧ *distinct D*} (*remdups-clause C*)›
(**is** ‹- ≤ ⇓ *?C* -›)
unfolding *remdups-clause-def remdups-clause'-def*
by (*auto intro*!: *RES-refine intro*!: *distinct-mset-mset-distinct*[*THEN iffD1*]
simp del: *distinct-mset-remdups-mset distinct-mset-mset-distinct*)
fastforce

have *confl*: ‹(*get-conflict-init-wl S*, *get-conflict-l-init S'*) ∈ ⟨*Id*⟩*option-rel*›
using *S-S'* **by** (*auto simp*: *twl-st-wl-init*)
have *false*: ‹(*add-empty-conflict-init-wl S*, *add-empty-conflict-init-l S'*) ∈ *?A*›
using *S-S'*
apply (*cases S*; *cases S'*)
apply (*auto simp*: *add-empty-conflict-init-wl-def add-empty-conflict-init-l-def*
all-blits-are-in-problem-init.simps state-wl-l-init'-def
state-wl-l-init-def state-wl-l-def correct-watching.simps clause-to-update-def)
done
have [*refine*]: ‹*ab* = *ac* ⟹ *C*=*C'* ⟹ *cons-trail-propagate-l* (*hd C*) *0 ab*
≤ ⇓ *Id*
(*cons-trail-propagate-l* (*hd C'*) *0 ac*)›
for *C C' ab ac*
by *auto*
have *propa-unit*:
‹*propagate-unit-init-wl* (*hd C*) *S* ≤ ⇓ *?A* (*propagate-unit-init-l* (*hd C'*) *S'*)›
if ‹(*C*, *C'*) ∈ *?C*› **for** *C C'*
using *S-S'* **apply** (*cases S*; *cases S'*; *cases* ‹*fst S*›; *cases* ‹*fst S'*›; *hypsubst*)
unfolding *propagate-unit-init-wl-def propagate-unit-init-l-def fst-conv*
apply *hypsubst*
unfolding *propagate-unit-init-wl-def propagate-unit-init-l-def*
apply *refine-rcg*
using *that*
apply (*auto simp*: *propagate-unit-init-l-def propagate-unit-init-wl-def state-wl-l-init'-def*
state-wl-l-init-def state-wl-l-def clause-to-update-def cons-trail-propagate-l-def
all-lits-of-mm-add-mset all-lits-of-m-add-mset all-lits-of-mm-union)
done

have *already-propa*:
‹(*already-propagated-unit-init-wl* (*mset C*) *S*, *already-propagated-unit-init-l* (*mset C'*) *S'*) ∈ *?A*›
if ‹(*C*, *C'*) ∈ *?C*› **for** *C C'*
using *S-S' that*
by (*cases S*; *cases S'*)
(*auto simp*: *already-propagated-unit-init-wl-def already-propagated-unit-init-l-def*
state-wl-l-init-def state-wl-l-def clause-to-update-def
all-lits-of-mm-add-mset all-lits-of-m-add-mset state-wl-l-init'-def)
have *set-conflict*: ‹(*set-conflict-init-wl* (*hd C*) *S*, *set-conflict-init-l C' S'*) ∈ *?A*›
if ‹*C'* = [*hd C*]› **for** *C C'*
using *S-S' that*
by (*cases S*; *cases S'*)
(*auto simp*: *set-conflict-init-wl-def set-conflict-init-l-def*
state-wl-l-init-def state-wl-l-def clause-to-update-def state-wl-l-init'-def
all-lits-of-mm-add-mset all-lits-of-m-add-mset)
have *add-to-clauses-init-wl*: ‹*add-to-clauses-init-wl C S*
≤ ⇓ *state-wl-l-init*

```
        (add-to-clauses-init-l C' S')›
    if C: ‹length C ≥ 2› and conf: ‹get-conflict-l-init S' = None› and
      CC': ‹(C,C') ∈ ?C› for C C'
  proof −
    have [iff]: ‹C ! Suc 0 ∉ set (watched-l C) ⟷ False›
      ‹C ! 0 ∉ set (watched-l C) ⟷ False› and
      [dest!]: ‹⋀L. L ≠ C ! 0 ⟹ L ≠ C ! Suc 0 ⟹ L ∈ set (watched-l C) ⟹ False›
      using C by (cases C; cases ‹tl C›; auto)+
    have [dest!]: ‹C ! 0 = C ! Suc 0 ⟹ False›
      using C CC' by (cases C; cases ‹tl C›; auto)+
    show ?thesis
      using S-S' conf C CC'
      by (cases S; cases S')
        (auto 5 5 simp: add-to-clauses-init-wl-def add-to-clauses-init-l-def get-fresh-index-def
          state-wl-l-init-def state-wl-l-def clause-to-update-def
          all-lits-of-mm-add-mset all-lits-of-m-add-mset  state-wl-l-init'-def
          RES-RETURN-RES Let-def
          intro!: RES-refine filter-mset-cong2)
  qed
  have add-to-other-init':
    ‹(add-to-other-init C S, add-to-other-init C S') ∈ ?A› for C
    using S-S'
    by (cases S; cases S')
      (auto simp: state-wl-l-init-def state-wl-l-def clause-to-update-def
       all-lits-of-mm-add-mset all-lits-of-m-add-mset  state-wl-l-init'-def)
  have add-to-tautology-init-wl:
    ‹(add-to-tautology-init-wl (C) S, add-to-tautology-init-l C' S') ∈ ?A›
    if ‹(C, C') ∈ Id›
    for C C'
    using S-S' that
    by (cases S; cases S')
     (auto simp: add-to-tautology-init-wl.simps add-to-tautology-init-l-def
      state-wl-l-init-def state-wl-l-init'-def)

  show ?thesis
    unfolding init-dt-step-wl-def init-dt-step-def
    apply (subst remdups-clause'-def[symmetric])
    apply (refine-rcg confl false propa-unit already-propa set-conflict remdups-clause
      add-to-clauses-init-wl add-to-other-init' add-to-tautology-init-wl)
    subgoal by simp
    subgoal by simp
    subgoal using S-S' by (simp add: twl-st-wl-init)
    subgoal using S-S' by (simp add: twl-st-wl-init)
    subgoal using S-S' by (simp add: twl-st-wl-init)
    subgoal using S-S' by (simp add: twl-st-wl-init)
    subgoal for C Ca by (cases Ca) auto
    subgoal by linarith
    done
qed

lemma init-dt-wl-init-dt:
  assumes S-S': ‹(S, S') ∈ state-wl-l-init›
  shows ‹init-dt-wl C S ≤ ⇓ state-wl-l-init
        (init-dt C S')›
proof −
  have C: ‹(C, C) ∈ ⟨Id⟩list-rel›
```

by (*auto simp*: *list-rel-def list.rel-refl-strong*)
show *?thesis*
unfolding *init-dt-wl-def init-dt-def*
apply (*refine-vcg C S-S′*)
subgoal using *S-S′* **by** *fast*
subgoal by (*auto intro*!: *init-dt-step-wl-init-dt-step*)
done
qed

definition *init-dt-wl-pre* **where**
‹*init-dt-wl-pre C S* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *state-wl-l-init* ∧
init-dt-pre C S′)›

definition *init-dt-wl-spec* **where**
‹*init-dt-wl-spec C S T* ⟷
(∃ *S′ T′*. (*S*, *S′*) ∈ *state-wl-l-init* ∧ (*T*, *T′*) ∈ *state-wl-l-init* ∧
init-dt-spec C S′ T′)›

lemma *init-dt-wl-init-dt-wl-spec*:
assumes ‹*init-dt-wl-pre CS S*›
shows ‹*init-dt-wl CS S* ≤ *SPEC* (*init-dt-wl-spec CS S*)›
proof −
obtain *S′* **where**
SS′: ‹(*S*, *S′*) ∈ *state-wl-l-init*› **and**
pre: ‹*init-dt-pre CS S′*›
using *assms* **unfolding** *init-dt-wl-pre-def* **by** *blast*
show *?thesis*
apply (*rule order.trans*)
apply (*rule init-dt-wl-init-dt*[*OF SS′*])
apply (*rule order.trans*)
apply (*rule ref-two-step′*)
apply (*rule init-dt-full*[*OF pre*])
apply (*unfold conc-fun-SPEC*)
apply (*rule SPEC-rule*)
apply *normalize-goal*+
using *SS′ pre* **unfolding** *init-dt-wl-spec-def*
by *blast*
qed

fun *correct-watching-init* :: ‹*'v twl-st-wl* ⇒ *bool*› **where**
[*simp del*]: ‹*correct-watching-init* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ⟷
all-blits-are-in-problem-init (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) ∧
(∀ *L*.
distinct-watched (*W L*) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *i* ∈# *dom-m N* ∧ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)) ∧
fst ‘# *mset* (*W L*) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#},
{#}))›

lemma *correct-watching-init-correct-watching*:
‹*correct-watching-init T* ⟹ *correct-watching T*›
by (*cases T*)
(*fastforce simp*: *correct-watching.simps correct-watching-init.simps filter-mset-eq-conv*

all-blits-are-in-problem-init.simps
in-clause-to-update-in-dom-mD)

lemma *image-mset-Suc*: ‹*Suc* `# {#*C* ∈# *M*. *P C*#} = {#*C* ∈# *Suc* `# *M*. *P* (*C*−1)#}›
by (*induction M*) *auto*

lemma *correct-watching-init-add-unit*:
assumes ‹*correct-watching-init* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
shows ‹*correct-watching-init* (*M*, *N*, *D*, *add-mset C NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)›
proof −
have [*intro*!]: ‹(*a*, *x*) ∈ *set* (*W L*) ⟹ *a* ∈# *dom-m N* ⟹ *b* ∈ *set* (*N* ∝*a*) ⟹
b ∉# *all-lits-of-mm* {#*mset* (*fst x*). *x* ∈# *ran-m N*#} ⟹ *b* ∈# *all-lits-of-mm NE*›
for *x b F a L*
unfolding *ran-m-def*
by (*auto dest*!: *multi-member-split simp*: *all-lits-of-mm-add-mset in-clause-in-all-lits-of-m*)

show *?thesis*
using *assms*
unfolding *correct-watching-init.simps clause-to-update-def Ball-def*
by (*fastforce simp*: *correct-watching.simps all-lits-of-mm-add-mset*
all-lits-of-m-add-mset Ball-def all-conj-distrib clause-to-update-def
all-blits-are-in-problem-init.simps all-lits-of-mm-union
dest!:)
qed

lemma *correct-watching-init-propagate*:
‹*correct-watching-init* ((*L* # *M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*)) ⟷
correct-watching-init ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))›
‹*correct-watching-init* ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *add-mset C Q*, *W*)) ⟷
correct-watching-init ((*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*))›
unfolding *correct-watching-init.simps clause-to-update-def Ball-def*
by (*auto simp*: *correct-watching.simps all-lits-of-mm-add-mset*
all-lits-of-m-add-mset Ball-def all-conj-distrib clause-to-update-def
all-blits-are-in-problem-init.simps)

lemma *all-blits-are-in-problem-cons*[*simp*]:
‹*all-blits-are-in-problem-init* (*Propagated L i* # *a*, *aa*, *ab*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*)
⟷
all-blits-are-in-problem-init (*a*, *aa*, *ab*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*)›
‹*all-blits-are-in-problem-init* (*Decided L* # *a*, *aa*, *ab*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*) ⟷
all-blits-are-in-problem-init (*a*, *aa*, *ab*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*)›
‹*all-blits-are-in-problem-init* (*a*, *aa*, *ab*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *add-mset L ae*, *b*) ⟷
all-blits-are-in-problem-init (*a*, *aa*, *ab*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*)›
‹*NO-MATCH None y* ⟹ *all-blits-are-in-problem-init* (*a*, *aa*, *y*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*,
ae, *b*) ⟷
all-blits-are-in-problem-init (*a*, *aa*, *None*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*)›
‹*NO-MATCH* {#} *ae* ⟹ *all-blits-are-in-problem-init* (*a*, *aa*, *y*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*,
ae, *b*) ⟷
all-blits-are-in-problem-init (*a*, *aa*, *y*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, {#}, *b*)›
by (*auto simp*: *all-blits-are-in-problem-init.simps*)

lemma *correct-watching-init-cons*[*simp*]:
‹*NO-MATCH None y* ⟹ *correct-watching-init* ((*a*, *aa*, *y*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*,
b)) ⟷
correct-watching-init ((*a*, *aa*, *None*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*, *b*))›
‹*NO-MATCH* {#} *ae* ⟹ *correct-watching-init* ((*a*, *aa*, *y*, *ac*, *ad*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *ae*,

```
b)) ⟷
    correct-watching-init ((a, aa, y, ac, ad, NEk, UEk, NS, US, N0, U0, {#}, b))›
    apply (auto simp: correct-watching-init.simps clause-to-update-def)
  apply (subst (asm) all-blits-are-in-problem-cons(4))
 apply auto
  apply (subst all-blits-are-in-problem-cons(4))
 apply auto
  apply (subst (asm) all-blits-are-in-problem-cons(5))
 apply auto
  apply (subst all-blits-are-in-problem-cons(5))
 apply auto
 done


lemma clause-to-update-mapsto-upd-notin:
 assumes
  i: ‹i ∉# dom-m N›
 shows
 ‹clause-to-update L (M, N(i ↪ C'), C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q) =
  (if L ∈ set (watched-l C')
   then add-mset i (clause-to-update L (M, N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q))
   else (clause-to-update L (M, N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q)))›
 ‹clause-to-update L (M, fmupd i (C', b) N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q) =
  (if L ∈ set (watched-l C')
   then add-mset i (clause-to-update L (M, N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q))
   else (clause-to-update L (M, N, C, NE, UE, NEk, UEk, NS, US, N0, U0, WS, Q)))›
 using assms
 by (auto simp: clause-to-update-def intro!: filter-mset-cong)

lemma correct-watching-init-add-clause:
 assumes
  corr: ‹correct-watching-init ((a, aa, None, ac, ad, NEk, UEk, NS, US, N0, U0, Q, b))› and
  leC: ‹2 ≤ length C› and
  i-notin[simp]: ‹i ∉# dom-m aa› and
  dist[iff]: ‹C ! 0 ≠ C ! Suc 0›
 shows ‹correct-watching-init
       ((a, fmupd i (C, red) aa, None, ac, ad, NEk, UEk, NS, US, N0, U0, Q, b
         (C ! 0 := b (C ! 0) @ [(i, C ! Suc 0, length C = 2)],
          C ! Suc 0 := b (C ! Suc 0) @ [(i, C ! 0, length C = 2)])))›
proof −
 have [iff]: ‹C ! Suc 0 ≠ C ! 0›
  using ‹C ! 0 ≠ C ! Suc 0› by argo
 have [iff]: ‹C ! Suc 0 ∈# all-lits-of-m (mset C)› ‹C ! 0 ∈# all-lits-of-m (mset C)›
  ‹C ! Suc 0 ∈ set C› ‹ C ! 0 ∈ set C› ‹C ! 0 ∈ set (watched-l C)› ‹C ! Suc 0 ∈ set (watched-l C)›
  using leC by (force intro!: in-clause-in-all-lits-of-m nth-mem simp: in-set-conv-iff
      intro: exI[of - 0] exI[of - ‹Suc 0›])+
 have [dest!]: ‹⋀L. L ≠ C ! 0 ⟹ L ≠ C ! Suc 0 ⟹ L ∈ set (watched-l C) ⟹ False›
   by (cases C; cases ‹tl C›; auto)+
 have i: ‹i ∉ fst ‘ set (b L)› for L
  using corr i-notin  unfolding correct-watching-init.simps
  by force
 have [iff]: ‹(i,c, d) ∉ set (b L)› for L c d
  using i[of L] by (auto simp: image-iff)
 then show ?thesis
  using corr
  by (force simp: correct-watching-init.simps all-blits-are-in-problem-init.simps ran-m-mapsto-upd-notin
```

```
    all-lits-of-mm-add-mset all-lits-of-mm-union clause-to-update-mapsto-upd-notin correctly-marked-as-binary.simps
      split: if-splits)
qed

definition rewatch
  :: ‹'v clauses-l ⇒ ('v literal ⇒ 'v watched) ⇒ ('v literal ⇒ 'v watched) nres›
where
‹rewatch N W = do {
  xs ← SPEC(λxs. set-mset (dom-m N) ⊆ set xs ∧ distinct xs);
  nfoldli
    xs
    (λ-. True)
    (λi W. do {
      if i ∈# dom-m N
      then do {
        ASSERT(i ∈# dom-m N);
        ASSERT(length (N ∝ i) ≥ 2);
        let L1 = N ∝ i ! 0;
        let L2 = N ∝ i ! 1;
        let n = length (N ∝ i);
        let b = (n = 2);
        ASSERT(L1 ≠ L2);
        ASSERT(length (W L1) < size (dom-m N));
        let W = W(L1 := W L1 @ [(i, L2, b)]);
        ASSERT(length (W L2) < size (dom-m N));
        let W = W(L2 := W L2 @ [(i, L1, b)]);
        RETURN W
      }
      else RETURN W
    })
    W
  }›

lemma rewatch-correctness:
  assumes [simp]: ‹W = (λ-. [])› and
    H[dest]: ‹⋀x. x ∈# dom-m N ⟹ distinct (N ∝ x) ∧ length (N ∝ x) ≥ 2›
  shows
    ‹rewatch N W ≤ SPEC(λW. correct-watching-init (M, N, C, NE, UE, NEk, UEk, NS, US, N₀,
U₀, Q, W))›
proof −
  define I where
    ‹I ≡ λ(a :: nat list) (b :: nat list) W.
       correct-watching-init ((M, fmrestrict-set (set a) N, C, NE, UE, NEk, UEk, NS, US, N₀, U₀, Q,
W))›
  have I0: ‹set-mset (dom-m N) ⊆ set x ∧ distinct x ⟹ I [] x W› for x
    unfolding I-def by (auto simp: correct-watching-init.simps
      all-blits-are-in-problem-init.simps clause-to-update-def)
  have le: ‹length (σ L) < size (dom-m N)›
    if ‹correct-watching-init (M, fmrestrict-set (set l1) N, C, NE, UE, NEk, UEk, NS, US, N₀, U₀,
Q, σ)› and
      ‹set-mset (dom-m N) ⊆ set x ∧ distinct x› and
    ‹x = l1 @ xa # l2› ‹xa ∈# dom-m N›
    for L l1 σ xa l2 x
  proof −
    have 1: ‹card (set l1) ≤ length l1›
      by (auto simp: card-length)
```

```
    have ‹distinct-watched (σ L)› and ‹fst ‘ set (σ L) ⊆ set l1 ∩ set-mset (dom-m N)›
      using that by (fastforce simp: correct-watching-init.simps dom-m-fmrestrict-set')+
    then have ‹length (map fst (σ L)) ≤ card (set l1 ∩ set-mset (dom-m N))›
      using 1 by (subst distinct-card[symmetric])
       (auto simp: distinct-watched-alt-def intro!: card-mono intro: order-trans)
    also have ‹... < card (set-mset (dom-m N))›
      using that by (auto intro!: psubset-card-mono)
    also have ‹... = size (dom-m N)›
      by (simp add: distinct-mset-dom distinct-mset-size-eq-card)
    finally show ?thesis by simp
  qed
  show ?thesis
    unfolding rewatch-def
    apply (subst (3) Let-def)
    apply (refine-vcg
      nfoldli-rule[where I = ‹I›])
    subgoal by (rule I0)
    subgoal using assms unfolding I-def by auto
    subgoal for x xa l1 l2 σ  using H[of xa] unfolding I-def apply −
      by (rule, subst (asm)nth-eq-iff-index-eq)
        linarith+
    subgoal for x xa l1 l2 σ unfolding I-def by (rule le)
    subgoal for x xa l1 l2 σ unfolding I-def by (drule le[where L = ‹N ∝ xa ! 1›]) (auto simp: I-def
dest!: le)
    subgoal for x xa l1 l2 σ
      unfolding I-def
      by (cases ‹the (fmlookup N xa)›)
        (auto simp: dom-m-fmrestrict-set' intro!: correct-watching-init-add-clause)
    subgoal
      unfolding I-def by auto
    subgoal by auto
    subgoal unfolding I-def
      by (auto simp: fmlookup-restrict-set-id')
    done
qed

definition state-wl-l-init-full :: ‹('v twl-st-wl-init-full × 'v twl-st-l-init) set› where
  ‹state-wl-l-init-full = {(S, S'). (fst S, fst S') ∈ state-wl-l None ∧
      snd S = snd S'}›

definition added-only-watched  :: ‹('v twl-st-wl-init-full × 'v twl-st-wl-init) set› where
  ‹added-only-watched = {(((M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W), OC), ((M', N',
D', NE', UE', NEk', UEk', NS', US', N0', U0', Q'), OC')).
        (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q) = (M', N', D', NE', UE', NEk', UEk',
NS', US', N0', U0', Q') ∧ OC = OC'}›

definition init-dt-wl-spec-full
  :: ‹'v clause-l list ⇒ 'v twl-st-wl-init ⇒ 'v twl-st-wl-init-full ⇒ bool›
where
  ‹init-dt-wl-spec-full C S T'' ⟷
    (∃S' T T'. (S, S') ∈ state-wl-l-init ∧ (T :: 'v twl-st-wl-init, T') ∈ state-wl-l-init ∧
      init-dt-spec C S' T' ∧ correct-watching-init (fst T'') ∧ (T'', T) ∈ added-only-watched)›

definition init-dt-wl-full :: ‹'v clause-l list ⇒ 'v twl-st-wl-init ⇒ 'v twl-st-wl-init-full nres› where
  ‹init-dt-wl-full CS S = do{
    ((M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q), OC) ← init-dt-wl CS S;
```

```
    W ← rewatch N (λ-. []);
    RETURN ((M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W), OC)
  }›

lemma init-dt-wl-spec-rewatch-pre:
  assumes ‹init-dt-wl-spec CS S T› and ‹N = get-clauses-init-wl T› and ‹C ∈# dom-m N›
  shows ‹distinct (N ∝ C) ∧ length (N ∝ C) ≥ 2›
proof −
  obtain x xa xb where
    ‹N = get-clauses-init-wl T› and
    Sx: ‹(S, x) ∈ state-wl-l-init› and
    Txa: ‹(T, xa) ∈ state-wl-l-init› and
    xa-xb: ‹(xa, xb) ∈ twl-st-l-init› and
    struct-invs: ‹twl-struct-invs-init xb› and
    ‹clauses-to-update-l-init xa = {#}› and
    ‹∀ s∈set (get-trail-l-init xa). ¬ is-decided s› and
    ‹get-conflict-l-init xa = None ⟶
     literals-to-update-l-init xa = uminus `# lit-of `# mset (get-trail-l-init xa)› and
    ‹remdups-mset `# mset `# mset CS + mset `# ran-mf (get-clauses-l-init x) + other-clauses-l-init x
+
     get-unit-clauses-l-init x + get-subsumed-init-clauses-l-init x + get-init-clauses0-l-init x =
     mset `# ran-mf (get-clauses-l-init xa) + other-clauses-l-init xa +
     get-unit-clauses-l-init xa + get-subsumed-init-clauses-l-init xa + get-init-clauses0-l-init xa› and
    ‹learned-clss-lf (get-clauses-l-init x) =
     learned-clss-lf (get-clauses-l-init xa)› and
    ‹get-learned-unit-clauses-l-init xa = get-learned-unit-clauses-l-init x› and
    ‹get-subsumed-learned-clauses-l-init xa = get-subsumed-learned-clauses-l-init x›
    ‹get-learned-clauses0-l-init xa = get-learned-clauses0-l-init x›
    ‹twl-list-invs (fst xa)› and
    ‹twl-stgy-invs (fst xb)› and
    ‹other-clauses-l-init xa ≠ {#} ⟶ get-conflict-l-init xa ≠ None› and
    ‹{#} ∈# mset `# mset CS ⟶ get-conflict-l-init xa ≠ None› and
    ‹get-conflict-l-init x ≠ None ⟶ get-conflict-l-init x = get-conflict-l-init xa›
    using assms
    unfolding init-dt-wl-spec-def init-dt-spec-def apply −
    by normalize-goal+ presburger

  have ‹twl-st-inv (fst xb)›
    using struct-invs unfolding twl-struct-invs-init-def by fast
  then have ‹Multiset.Ball (get-clauses (fst xb)) struct-wf-twl-cls›
    by (cases xb) (auto simp: twl-st-inv.simps)
  with ‹C ∈# dom-m N› show ?thesis
    using Txa xa-xb assms by (cases T; cases ‹fmlookup N C›; cases ‹snd (the(fmlookup N C))›)
        (auto simp: state-wl-l-init-def twl-st-l-init-def conj-disj-distribR Collect-disj-eq
       Collect-conv-if mset-take-mset-drop-mset′
       state-wl-l-init′-def ran-m-def dest!: multi-member-split)
qed

lemma init-dt-wl-full-init-dt-wl-spec-full:
  assumes ‹init-dt-wl-pre CS S›
  shows ‹init-dt-wl-full CS S ≤ SPEC (init-dt-wl-spec-full CS S)›
proof −
  show ?thesis
    unfolding init-dt-wl-full-def
    apply (rule specify-left)
    apply (rule init-dt-wl-init-dt-wl-spec)
```

subgoal by (*rule assms*)
apply *clarify*
apply (*rule specify-left*)
apply (*rule-tac M* =*a* **and** *N*=*aa* **and** *C*=*ab* **and** *NE*=*ac* **and** *UE*=*ad* **and** *Q*=*b* **and**
NEk=*ae* **and** *UEk* = *af* **and** *NS*=*ag* **and** *US*=*ah* **and** N_0=*ai* **and** U_0=*aj* **in**
rewatch-correctness[*OF* - *init-dt-wl-spec-rewatch-pre*])
subgoal by *simp*
apply *assumption*
subgoal by *simp*
subgoal by *simp*
subgoal for *a aa ab ac ad ae af ag ah b ba W*
using *assms*
unfolding *init-dt-wl-spec-full-def init-dt-wl-pre-def init-dt-wl-spec-def*
by (*auto simp*: *added-only-watched-def state-wl-l-init-def state-wl-l-init'-def*)
done
qed

end
theory *CDCL-Conflict-Minimisation*
imports
Watched-Literals-Watch-List
More-Sepref.WB-More-Refinement
More-Sepref.WB-More-Refinement-List List−Index.List-Index
begin

Chapter 8

Conflict Minimisation

We implement the conflict minimisation as presented by Sörensson and Biere ("Minimizing Learned Clauses"').

We refer to the paper for further details, but the general idea is to produce a series of resolution steps such that eventually (i.e., after enough resolution steps) no new literals has been introduced in the conflict clause.

The resolution steps are only done with the reasons of the of literals appearing in the trail. Hence these steps are terminating: we are "shortening" the trail we have to consider with each resolution step. Remark that the shortening refers to the length of the trail we have to consider, not the levels.

The concrete proof was harder than we initially expected. Our first proof try was to certify the resolution steps. While this worked out, adding caching on top of that turned to be rather hard, since it is not obvious how to add resolution steps in the middle of the current proof if the literal has already been removed (basically we would have to prove termination and confluence of the rewriting system). Therefore, we worked instead directly on the entailment of the literals of the conflict clause (up to the point in the trail we currently considering, which is also the termination measure). The previous try is still present in our formalisation (see *minimize-conflict-support*, which we however only use for the termination proof).

The algorithm presented above does not distinguish between literals propagated at the same level: we cannot reuse information about failures to cut branches. There is a variant of the algorithm presented above that is able to do so (Van Gelder, "Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces"). The algorithm is however more complicated and has only be implemented in very few solvers (at least lingeling and cadical) and is especially not part of glucose nor cryptominisat. Therefore, we have decided to not implement it: It is probably not worth it and requires some additional data structures.

declare $cdcl_W$*-restart-mset-state*[*simp*]

type-synonym *out-learned* = ‹*nat clause-l*›

The data structure contains the (unique) literal of highest at position one. This is useful since this is what we want to have at the end (propagation clause) and we can skip the first literal when minimising the clause.

definition *out-learned* :: ‹(*nat*, *nat*) *ann-lits* ⇒ *nat clause option* ⇒ *out-learned* ⇒ *bool*› **where**
‹*out-learned M D out* ⟷
out ≠ [] ∧
(*D* = *None* ⟶ *length out* = *1*) ∧
(*D* ≠ *None* ⟶ *mset* (*tl out*) = *filter-mset* (λ*L*. *get-level M L* < *count-decided M*) (*the D*))›

definition *out-learned-confl* :: ‹(*nat*, *nat*) *ann-lits* ⇒ *nat clause option* ⇒ *out-learned* ⇒ *bool*› **where**
‹*out-learned-confl M D out* ⟷
out ≠ [] ∧ (*D* ≠ *None* ∧ *mset out* = *the D*)›

lemma *out-learned-Cons-None*[*simp*]:
‹*out-learned* (*L* # *aa*) *None ao* ⟷ *out-learned aa None ao*›
by (*auto simp*: *out-learned-def*)

lemma *out-learned-tl-None*[*simp*]:
‹*out-learned* (*tl aa*) *None ao* ⟷ *out-learned aa None ao*›
by (*auto simp*: *out-learned-def*)

definition *index-in-trail* :: ‹('*v*, '*a*) *ann-lits* ⇒ '*v literal* ⇒ *nat*› **where**
‹*index-in-trail M L* = *index* (*map* (*atm-of o lit-of*) (*rev M*)) (*atm-of L*)›

lemma *Propagated-in-trail-entailed*:
assumes
invs: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *N*, *U*, *D*)› **and**
in-trail: ‹*Propagated L C* ∈ *set M*›
shows
‹*M* ⊨*as CNot* (*remove1-mset L C*)› **and** ‹*L* ∈# *C*› **and** ‹*N* + *U* ⊨*pm C*› **and**
‹*K* ∈# *remove1-mset L C* ⟹ *index-in-trail M K* < *index-in-trail M L*› **and**
‹¬*tautology C*› **and** ‹*distinct-mset C*›
proof −
obtain *M2 M1* **where**
M: ‹*M* = *M2* @ *Propagated L C* # *M1*›
using *split-list*[*OF in-trail*] **by** *metis*
have ‹*a* @ *Propagated L mark* # *b* = *trail* (*M*, *N*, *U*, *D*) ⟶
b ⊨*as CNot* (*remove1-mset L mark*) ∧ *L* ∈# *mark*› **and**
dist: ‹*cdcl*$_W$*-restart-mset.distinct-cdcl*$_W$*-state* (*M*, *N*, *U*, *D*)›
for *L mark a b*
using *invs*
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
by *fast*+
then have *L-E*: ‹*L* ∈# *C*› **and** *M1-E*: ‹*M1* ⊨*as CNot* (*remove1-mset L C*)›
unfolding *M* **by** *force*+
then have *M-E*: ‹*M* ⊨*as CNot* (*remove1-mset L C*)›
unfolding *M* **by** (*simp add*: *true-annots-append-l*)
show ‹*M* ⊨*as CNot* (*remove1-mset L C*)› **and** ‹*L* ∈# *C*›
using *L-E M-E* **by** *fast*+
have ‹*set* (*get-all-mark-of-propagated* (*trail* (*M*, *N*, *U*, *D*)))
⊆ *set-mset* (*cdcl*$_W$*-restart-mset.clauses* (*M*, *N*, *U*, *D*))›
using *invs*
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
by *fast*
then have ‹*C* ∈# *N* + *U*›
using *in-trail cdcl*$_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*[*of C M*]
by (*auto simp*: *clauses-def*)
then show ‹*N* + *U* ⊨*pm C*› **by** *auto*

have *n-d*: ‹*no-dup M*›
using *invs*
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*

cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def
by *auto*
show ‹*index-in-trail M K < index-in-trail M L*› **if** *K-C*: ‹*K ∈# remove1-mset L C*›
proof −
have
KL: ‹*atm-of K ≠ atm-of L*› **and**
uK-M1: ‹*−K ∈ lits-of-l M1*› **and**
L: ‹*L ∉ lit-of ‘ (set M2 ∪ set M1)*› ‹*−L ∉ lit-of ‘ (set M2 ∪ set M1)*›
using *M1-E K-C n-d* **unfolding** *M true-annots-true-cls-def-iff-negation-in-model*
by (*auto dest*!: *multi-member-split simp*: *atm-of-eq-atm-of lits-of-def uminus-lit-swap*
Decided-Propagated-in-iff-in-lits-of-l)
have *L-M1*: ‹*atm-of L ∉ (atm-of ∘ lit-of) ‘ set M1*›
using *L* **by** (*auto simp*: *image-Un atm-of-eq-atm-of*)
have *K-M1*: ‹*atm-of K ∈ (atm-of ∘ lit-of) ‘ set M1*›
using *uK-M1* **by** (*auto simp*: *lits-of-def image-image comp-def uminus-lit-swap*)
show *?thesis*
using *KL L-M1 K-M1* **unfolding** *index-in-trail-def M* **by** (*auto simp*: *index-append*)
qed
have ‹*¬tautology*(*remove1-mset L C*)›
by (*rule consistent-CNot-not-tautology*[*of* ‹*lits-of-l M1*›])
(*use n-d M1-E* **in** ‹*auto dest*: *distinct-consistent-interp no-dup-appendD*
simp: *true-annots-true-cls M*›)
then show ‹*¬tautology C*›
using *multi-member-split*[*OF L-E*] *M1-E n-d*
by (*auto simp*: *tautology-add-mset true-annots-true-cls-def-iff-negation-in-model M*
dest!: *multi-member-split in-lits-of-l-defined-litD*)
show ‹*distinct-mset (C)*›
using *dist* ‹*C ∈# N + U*› **unfolding** *cdcl$_W$-restart-mset.distinct-cdcl$_W$-state-def*
by (*auto dest*: *multi-member-split*)
qed

This predicate corresponds to one resolution step.

inductive *minimize-conflict-support* :: ‹(*'v*, *'v clause*) *ann-lits* ⇒ *'v clause* ⇒ *'v clause* ⇒ *bool*›
for *M* **where**
resolve-propa:
‹*minimize-conflict-support M (add-mset (−L) C) (C + remove1-mset L E)*›
if ‹*Propagated L E ∈ set M*› |
remdups: ‹*minimize-conflict-support M (add-mset L C) C*›

lemma *index-in-trail-uminus*[*simp*]: ‹*index-in-trail M (−L) = index-in-trail M L*›
by (*auto simp*: *index-in-trail-def*)

This is the termination argument of the conflict minimisation: the multiset of the levels decreases (for the multiset ordering).

definition *minimize-conflict-support-mes* :: ‹(*'v*, *'v clause*) *ann-lits* ⇒ *'v clause* ⇒ *nat multiset*›
where
‹*minimize-conflict-support-mes M C = index-in-trail M ‘# C*›

context
fixes *M* :: ‹(*'v*, *'v clause*) *ann-lits*› **and** *N U* :: ‹*'v clauses*› **and**
D :: ‹*'v clause option*›
assumes *invs*: ‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv* (*M*, *N*, *U*, *D*)›
begin

private lemma
no-dup: ‹*no-dup M*› **and**
consistent: ‹*consistent-interp* (*lits-of-l M*)›
using *invs* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by *simp-all*

lemma *minimize-conflict-support-entailed-trail*:
assumes ‹*minimize-conflict-support M C E*› **and** ‹*M* ⊨*as CNot C*›
shows ‹*M* ⊨*as CNot E*›
using *assms*
proof (*induction rule*: *minimize-conflict-support.induct*)
case (*resolve-propa L E C*) **note** *in-trail* = *this*(*1*) **and** *M-C* = *this*(*2*)
then show *?case*
using *Propagated-in-trail-entailed*[*OF invs in-trail*] **by** (*auto dest*!: *multi-member-split*)
next
case (*remdups L C*)
then show *?case*
by *auto*
qed

lemma *rtranclp-minimize-conflict-support-entailed-trail*:
assumes ‹(*minimize-conflict-support M*)** *C E*› **and** ‹*M* ⊨*as CNot C*›
shows ‹*M* ⊨*as CNot E*›
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
subgoal by *fast*
subgoal using *minimize-conflict-support-entailed-trail* **by** *fast*
done

lemma *minimize-conflict-support-mes*:
assumes ‹*minimize-conflict-support M C E*›
shows ‹*minimize-conflict-support-mes M E* < *minimize-conflict-support-mes M C*›
using *assms* **unfolding** *minimize-conflict-support-mes-def*
proof (*induction rule*: *minimize-conflict-support.induct*)
case (*resolve-propa L E C*) **note** *in-trail* = *this*
let *?f* = ‹λ*xa*. *index* (*map* (λ*a*. *atm-of* (*lit-of a*)) (*rev M*)) *xa*›
have ‹*?f* (*atm-of x*) < *?f* (*atm-of L*)› **if** *x*: ‹*x* ∈# *remove1-mset L E*› **for** *x*
proof −
obtain *M2 M1* **where**
M: ‹*M* = *M2* @ *Propagated L E* # *M1*›
using *split-list*[*OF in-trail*] **by** *metis*
have ‹*a* @ *Propagated L mark* # *b* = *trail* (*M*, *N*, *U*, *D*) ⟶
b ⊨*as CNot* (*remove1-mset L mark*) ∧ *L* ∈# *mark*› **for** *L mark a b*
using *invs*
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
by *fast*
then have *L-E*: ‹*L* ∈# *E*› **and** *M-E*: ‹*M1* ⊨*as CNot* (*remove1-mset L E*)›
unfolding *M* **by** *force*+
then have ‹−*x* ∈ *lits-of-l M1*›
using *x* **unfolding** *true-annots-true-cls-def-iff-negation-in-model* **by** *auto*
then have ‹*?f* (*atm-of x*) < *length M1*›
using *no-dup*
by (*auto simp*: *M lits-of-def index-append Decided-Propagated-in-iff-in-lits-of-l*
uminus-lit-swap)
moreover have ‹*?f* (*atm-of L*) = *length M1*›

```
    using no-dup unfolding M by (auto simp: index-append Decided-Propagated-in-iff-in-lits-of-l
        atm-of-eq-atm-of lits-of-def)
   ultimately show ?thesis by auto
  qed

  then show ?case by (auto simp: comp-def index-in-trail-def)
next
  case (remdups L C)
  then show ?case by auto
qed

lemma wf-minimize-conflict-support:
  shows ‹wf {(C', C). minimize-conflict-support M C C'}›
  apply (rule wf-if-measure-in-wf[of ‹{(C', C). C' < C}› - ‹minimize-conflict-support-mes M›])
  subgoal using wf .
  subgoal using minimize-conflict-support-mes by auto
  done
end

lemma conflict-minimize-step:
  assumes
    ‹NU ⊨p add-mset L C› and
    ‹NU ⊨p add-mset (−L) D› and
    ‹⋀K'. K' ∈# C ⟹ NU ⊨p add-mset (−K') D›
  shows ‹NU ⊨p D›
proof −
  have ‹NU ⊨p D + C›
    using assms(1,2) true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or by blast
  then show ?thesis
    using assms(3)
  proof (induction C)
    case empty
    then show ?case
      using true-clss-cls-in true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or by fastforce
  next
    case (add x C) note IH =this(1) and NU-DC = this(2) and entailed = this(3)
    have ‹NU ⊨p D + C + D›
      using entailed[of x] NU-DC
        true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or[of NU ‹−x› ‹D + C› D]
      by auto
    then have ‹NU ⊨p D + C›
      by (smt (verit) total-over-m-sum total-over-m-union true-cls-union true-clss-cls-def)
    from IH[OF this] entailed show ?case by auto
  qed
qed
```

This function filters the clause by the levels up the level of the given literal. This is the part the conflict clause that is considered when testing if the given literal is redundant.

```
definition filter-to-poslev where
  ‹filter-to-poslev M L D = filter-mset (λK. index-in-trail M K < index-in-trail M L) D›

lemma filter-to-poslev-uminus[simp]:
  ‹filter-to-poslev M (−L) D = filter-to-poslev M L D›
  by (auto simp: filter-to-poslev-def)

lemma filter-to-poslev-empty[simp]:
```

‹*filter-to-poslev M L* {#} = {#}›
by (*auto simp*: *filter-to-poslev-def*)

lemma *filter-to-poslev-mono*:
‹*index-in-trail M K′* ≤ *index-in-trail M L* ⟹
filter-to-poslev M K′ D ⊆# *filter-to-poslev M L D*›
unfolding *filter-to-poslev-def*
by (*auto simp*: *multiset-filter-mono2*)

lemma *filter-to-poslev-mono-entailement*:
‹*index-in-trail M K′* ≤ *index-in-trail M L* ⟹
NU ⊨p *filter-to-poslev M K′ D* ⟹ *NU* ⊨p *filter-to-poslev M L D*›
by (*metis* (*full-types*) *filter-to-poslev-mono subset-mset.le-iff-add true-clss-cls-mono-r*)

lemma *filter-to-poslev-mono-entailement-add-mset*:
‹*index-in-trail M K′* ≤ *index-in-trail M L* ⟹
NU ⊨p *add-mset J* (*filter-to-poslev M K′ D*) ⟹ *NU* ⊨p *add-mset J* (*filter-to-poslev M L D*)›
by (*metis filter-to-poslev-mono mset-subset-eq-add-mset-cancel subset-mset.le-iff-add*
true-clss-cls-mono-r)

lemma *conflict-minimize-intermediate-step*:
assumes
‹*NU* ⊨p *add-mset L C*› **and**
K′-C: ‹⋀*K′*. *K′* ∈# *C* ⟹ *NU* ⊨p *add-mset* (−*K′*) *D* ∨ *K′* ∈# *D*›
shows ‹*NU* ⊨p *add-mset L D*›
proof −
have ‹*NU* ⊨p *add-mset L C* + *D*›
using *assms*(*1*) *true-clss-cls-mono-r* **by** *blast*
then show *?thesis*
using *assms*(*2*)
proof (*induction C*)
case *empty*
then show *?case*
using *true-clss-cls-in true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or* **by** *fastforce*
next
case (*add x C*) **note** *IH* =*this*(*1*) **and** *NU-DC* = *this*(*2*) **and** *entailed* = *this*(*3*)

have *1*: ‹*NU* ⊨p *add-mset x* (*add-mset L* (*D* + *C*))›
using *NU-DC* **by** (*auto simp*: *add-mset-commute ac-simps*)
moreover have *2*: ‹*remdups-mset* (*add-mset L* (*D* + *C* + *D*)) = *remdups-mset* (*add-mset L* (*C* +
D))›
by (*auto simp*: *remdups-mset-def*)
moreover have *3*: ‹*remdups-mset* (*D* + *C* + *D*) = *remdups-mset* (*D* + *C*)›
by (*auto simp*: *remdups-mset-def*)
moreover have ‹*x* ∈# *D* ⟹ *NU* ⊨p *add-mset L* (*D* + *C* + *D*)›
using *1*
apply (*subst* (*asm*) *true-clss-cls-remdups-mset*[*symmetric*])
apply (*subst true-clss-cls-remdups-mset*[*symmetric*])
by (*auto simp*: *2 3*)
ultimately have ‹*NU* ⊨p *add-mset L* (*D* + *C* + *D*)›
using *entailed*[*of x*] *NU-DC*
true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or[*of NU* ‹−*x*› ‹*add-mset L D* + *C*› *D*]
by *auto*
moreover have ‹*remdups-mset* (*D* + (*C* + *D*)) = *remdups-mset* (*D* + *C*)›
by (*auto simp*: *remdups-mset-def*)
ultimately have ‹*NU* ⊨p *add-mset L C* + *D*›

```
    apply (subst true-clss-cls-remdups-mset[symmetric])
    apply (subst (asm) true-clss-cls-remdups-mset[symmetric])
    by (auto simp add: 3 2 add.commute simp del: true-clss-cls-remdups-mset)
  from IH[OF this] entailed show ?case by auto
 qed
qed

lemma conflict-minimize-intermediate-step-filter-to-poslev:
 assumes
  lev-K-L: ‹⋀K'. K' ∈# C ⟹ index-in-trail M K' < index-in-trail M L› and
  NU-LC: ‹NU ⊨p add-mset L C› and
  K'-C: ‹⋀K'. K' ∈# C ⟹ NU ⊨p add-mset (−K') (filter-to-poslev M L D) ∨
   K' ∈# filter-to-poslev M L D›
 shows ‹NU ⊨p add-mset L (filter-to-poslev M L D)›
proof −
 have C-entailed: ‹K' ∈# C ⟹ NU ⊨p add-mset (−K') (filter-to-poslev M L D) ∨
  K' ∈# filter-to-poslev M L D› for K'
  using filter-to-poslev-mono[of M K' L D] lev-K-L[of K'] K'-C[of K']
   true-clss-cls-mono-r[of - ‹ add-mset (− K') (filter-to-poslev M K' D)› ]
  by (auto simp: mset-subset-eq-exists-conv)
 show ?thesis
  using conflict-minimize-intermediate-step[OF NU-LC C-entailed] by fast
qed

datatype minimize-status = SEEN-FAILED | SEEN-REMOVABLE | SEEN-UNKNOWN

instantiation minimize-status :: default
begin
 definition default-minimize-status where
  ‹default-minimize-status = SEEN-UNKNOWN›

instance by standard
end

type-synonym 'v conflict-min-analyse = ‹('v literal × 'v clause) list›
type-synonym 'v conflict-min-cach = ‹'v ⇒ minimize-status›

definition get-literal-and-remove-of-analyse
  :: ‹'v conflict-min-analyse ⇒ ('v literal × 'v conflict-min-analyse) nres› where
 ‹get-literal-and-remove-of-analyse analyse =
  SPEC(λ(L, ana). L ∈# snd (hd analyse) ∧ tl ana = tl analyse ∧ ana ≠ [] ∧
      hd ana = (fst (hd analyse), snd (hd (analyse)) − {#L#}))›

definition mark-failed-lits
 :: ‹- ⇒ 'v conflict-min-analyse ⇒ 'v conflict-min-cach ⇒ 'v conflict-min-cach nres›
where
 ‹mark-failed-lits NU analyse cach = SPEC(λcach'.
   (∀ L. cach' L = SEEN-REMOVABLE ⟶ cach L = SEEN-REMOVABLE))›


definition conflict-min-analysis-inv
 :: ‹('v, 'a) ann-lits ⇒ 'v conflict-min-cach ⇒ 'v clauses ⇒ 'v clause ⇒ bool›
where
 ‹conflict-min-analysis-inv M cach NU D ⟷
   (∀ L. L ∈ lits-of-l M ⟶ cach (atm-of L) = SEEN-REMOVABLE ⟶
    set-mset NU ⊨p add-mset L (filter-to-poslev M L D))›
```

lemma *conflict-min-analysis-inv-alt-def*:
‹*conflict-min-analysis-inv M cach NU D* ⟷
(∀ *L*. −*L* ∈ *lits-of-l M* ⟶ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ⟶
set-mset NU ⊨*p add-mset* (−*L*) (*filter-to-poslev M L D*))›
unfolding *conflict-min-analysis-inv-def filter-to-poslev-def index-in-trail-def*
apply (*auto dest*: *spec*[*of* ‹− ›] *intro*: *filter-mset-cong2 simp*: *atm-of-eq-atm-of*)
by (*metis* (*no-types*, *lifting*) *atm-of-eq-atm-of atm-of-uminus filter-mset-cong2*)

lemma *conflict-min-analysis-inv-update-removable*:
‹*no-dup M* ⟹ −*L* ∈ *lits-of-l M* ⟹
conflict-min-analysis-inv M (*cach*(*atm-of L* := *SEEN-REMOVABLE*)) *NU D* ⟷
conflict-min-analysis-inv M cach NU D ∧ *set-mset NU* ⊨*p add-mset* (−*L*) (*filter-to-poslev M L D*)›
by (*auto simp*: *conflict-min-analysis-inv-alt-def atm-of-eq-atm-of dest*: *no-dup-consistentD*)

lemma *conflict-min-analysis-inv-update-failed*:
‹*conflict-min-analysis-inv M cach NU D* ⟹
conflict-min-analysis-inv M (*cach*(*L* := *SEEN-FAILED*)) *NU D*›
by (*auto simp*: *conflict-min-analysis-inv-def*)

fun *conflict-min-analysis-stack*
:: ‹('*v*, '*a*) *ann-lits* ⇒ '*v clauses* ⇒ '*v clause* ⇒ '*v conflict-min-analyse* ⇒ *bool*›
where
‹*conflict-min-analysis-stack M NU D* [] ⟷ *True*› |
‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # []) ⟷ −*L* ∈ *lits-of-l M*› |
‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # (*L*′, *E*′) # *analyse*) ⟷
(∃ *C*. *set-mset NU* ⊨*p add-mset* (−*L*′) *C* ∧
(∀ *K*∈#*C*−*add-mset L E*′. *set-mset NU* ⊨*p* (*filter-to-poslev M L*′ *D*) + {#−*K*#} ∨
K ∈# *filter-to-poslev M L*′ *D*) ∧
(∀ *K*∈#*C*. *index-in-trail M K* < *index-in-trail M L*′) ∧
E′ ⊆# *C*) ∧
−*L*′ ∈ *lits-of-l M* ∧
−*L* ∈ *lits-of-l M* ∧
index-in-trail M L < *index-in-trail M L*′ ∧
conflict-min-analysis-stack M NU D ((*L*′, *E*′) # *analyse*)›

lemma *conflict-min-analysis-stack-change-hd*:
‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # *ana*) ⟹
conflict-min-analysis-stack M NU D ((*L*, *E*′) # *ana*)›
by (*cases ana*, *auto*)

lemma *conflict-min-analysis-stack-sorted*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹
sorted (*map* (*index-in-trail M o fst*) *analyse*)›
by (*induction rule*: *conflict-min-analysis-stack.induct*)
auto
lemma *conflict-min-analysis-stack-sorted-and-distinct*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹
sorted (*map* (*index-in-trail M o fst*) *analyse*) ∧
distinct (*map* (*index-in-trail M o fst*) *analyse*)›
by (*induction rule*: *conflict-min-analysis-stack.induct*)
auto

lemma *conflict-min-analysis-stack-distinct-fst*:
assumes ‹*conflict-min-analysis-stack M NU D analyse*›

shows ‹*distinct* (*map fst analyse*)› **and** ‹*distinct* (*map* (*atm-of o fst*) *analyse*)›
proof −
have *dist*: ‹*distinct* (*map* (*index-in-trail M* ∘ *fst*) *analyse*)›
using *conflict-min-analysis-stack-sorted-and-distinct*[*of M NU D analyse*, *OF assms*]
by *auto*
then show ‹*distinct* (*map fst analyse*)›
by (*auto simp*: *intro*!: *distinct-mapI*[*of* ‹(*index-in-trail M*)›])
show ‹*distinct* (*map* (*atm-of o fst*) *analyse*)›
proof (*rule ccontr*)
assume ‹¬*?thesis*›
from *not-distinct-decomp*[*OF this*]
obtain *xs L ys zs* **where** ‹*map* (*atm-of o fst*) *analyse* = *xs* @ *L* # *ys* @ *L* # *zs*›
by *auto*
then show *False*
using *dist*
by (*auto simp*: *map-eq-append-conv atm-of-eq-atm-of Int-Un-distrib image-Un*)
qed
qed

lemma *conflict-min-analysis-stack-neg*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹
M ⊨*as CNot* (*fst* ‘# *mset analyse*)›
by (*induction M NU D analyse rule*: *conflict-min-analysis-stack.induct*)
auto

fun *conflict-min-analysis-stack-hd*
:: ‹('*v*, '*a*) *ann-lits* ⇒ '*v clauses* ⇒ '*v clause* ⇒ '*v conflict-min-analyse* ⇒ *bool*›
where
‹*conflict-min-analysis-stack-hd M NU D* [] ⟷ *True*› |
‹*conflict-min-analysis-stack-hd M NU D* ((*L*, *E*) # -) ⟷
(∃ *C*. *set-mset NU* ⊨*p add-mset* (−*L*) *C* ∧
(∀ *K*∈#*C*. *index-in-trail M K* < *index-in-trail M L*) ∧ *E* ⊆# *C* ∧ −*L* ∈ *lits-of-l M* ∧
(∀ *K*∈#*C*−*E*. *set-mset NU* ⊨*p* (*filter-to-poslev M L D*) + {#−*K*#} ∨ *K* ∈# *filter-to-poslev M L D*))›
lemma *conflict-min-analysis-stack-tl*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹ *conflict-min-analysis-stack M NU D* (*tl analyse*)›
by (*cases* ‹(*M*, *NU*, *D*, *analyse*)› *rule*: *conflict-min-analysis-stack.cases*) *auto*

definition *lit-redundant-inv*
:: ‹('*v*, '*v clause*) *ann-lits* ⇒ '*v clauses* ⇒ '*v clause* ⇒ '*v conflict-min-analyse* ⇒
'*v conflict-min-cach* × '*v conflict-min-analyse* × *bool* ⇒ *bool*› **where**
‹*lit-redundant-inv M NU D init-analyse* = (λ(*cach*, *analyse*, *b*).
conflict-min-analysis-inv M cach NU D ∧
(*analyse* ≠ [] ⟶ *fst* (*hd init-analyse*) = *fst* (*last analyse*)) ∧
(*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analyse*))) = *SEEN-REMOVABLE*) ∧
conflict-min-analysis-stack M NU D analyse ∧
conflict-min-analysis-stack-hd M NU D analyse)›

definition *lit-redundant-rec-loop-inv* :: ‹('*v*, '*v clause*) *ann-lits* ⇒
'*v conflict-min-cach* × '*v conflict-min-analyse* × *bool* ⇒ *bool*› **where**
‹*lit-redundant-rec-loop-inv M* = (λ(*cach*, *analyse*, *b*).
(*uminus o fst*) ‘# *mset analyse* ⊆# *lit-of* ‘# *mset M* ∧
(∀ *L* ∈ *set analyse*. *cach* (*atm-of* (*fst L*)) = *SEEN-UNKNOWN*))›

definition *lit-redundant-rec* :: ‹('*v*, '*v clause*) *ann-lits* ⇒ '*v clauses* ⇒ '*v clause* ⇒

```
    'v conflict-min-cach ⇒ 'v conflict-min-analyse ⇒
    ('v conflict-min-cach × 'v conflict-min-analyse × bool) nres›
where
  ‹lit-redundant-rec M NU D cach analysis =
    WHILE_T^lit-redundant-rec-loop-inv M
     (λ(cach, analyse, b). analyse ≠ [])
     (λ(cach, analyse, b). do {
        ASSERT(analyse ≠ []);
        ASSERT(length analyse ≤ length M);
        ASSERT(−fst (hd analyse) ∈ lits-of-l M);
        if snd (hd analyse) = {#}
        then
          RETURN(cach (atm-of (fst (hd analyse)) := SEEN-REMOVABLE), tl analyse, True)
        else do {
          (L, analyse) ← get-literal-and-remove-of-analyse analyse;
          ASSERT(−L ∈ lits-of-l M);
          b ← RES UNIV;
          if (get-level M L = 0 ∨ cach (atm-of L) = SEEN-REMOVABLE ∨ L ∈# D)
          then RETURN (cach, analyse, False)
          else if b ∨ cach (atm-of L) = SEEN-FAILED
          then do {
            cach ← mark-failed-lits NU analyse cach;
            RETURN (cach, [], False)
          }
          else do {
            ASSERT(cach (atm-of L) = SEEN-UNKNOWN);
            C ← get-propagation-reason M (−L);
            case C of
              Some C ⇒ do {
    ASSERT (distinct-mset C ∧ ¬tautology C);
    RETURN (cach, (L, C − {#−L#}) # analyse, False)}
            | None ⇒ do {
              cach ← mark-failed-lits NU analyse cach;
              RETURN (cach, [], False)
            }
        }
      }
     })
     (cach, analysis, False)›

definition lit-redundant-rec-spec where
  ‹lit-redundant-rec-spec M NU D L =
    SPEC(λ(cach, analysis, b). (b ⟶ NU ⊨pm add-mset (−L) (filter-to-poslev M L D)) ∧
     conflict-min-analysis-inv M cach NU D)›

lemma WHILEIT-rule-stronger-inv-keepI':
  assumes
    ‹wf R› and
    ‹I s› and
    ‹I' s› and
    ‹⋀s. I s ⟹ I' s ⟹ b s ⟹ f s ≤ SPEC (λs'. I' s')› and
    ‹⋀s. I s ⟹ I' s ⟹ b s ⟹ f s ≤ SPEC (λs'. I' s' ⟶ (I s' ∧ (s', s) ∈ R))› and
    ‹⋀s. I s ⟹ I' s ⟹ ¬ b s ⟹ Φ s›
  shows ‹WHILE_T^I b f s ≤ SPEC Φ›
proof −
  have A[iff]: ‹f s ≤ SPEC (λv. I' v ∧ I v ∧ (v, s) ∈ R) ⟷ f s ≤ SPEC (λs'. I s' ∧ I' s' ∧ (s', s)
```

$\in R)\rangle$ **for** s
  by (*rule cong*[*of* ‹ $\lambda n.\ f\ s \leq n$›]) *auto*
 then have *H*: ‹$I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \leq SPEC\ (\lambda s'.\ I\ s' \wedge\ I'\ s' \wedge (s',\ s) \in R)$› **for** s
  using *SPEC-rule-conjI* [*OF assms*(*4*,*5*)[*of s*]] **by** *auto*
 have ‹$WHILE_T{}^I\ b\ f\ s \leq WHILE_T{}^{\lambda s.\ I\ s \wedge I'\ s}\ b\ f\ s$›
  by (*metis* (*mono-tags*, *lifting*) *WHILEIT-weaken*)

 also have ‹$WHILE_T{}^{\lambda s.\ I\ s \wedge I'\ s}\ b\ f\ s \leq SPEC\ \Phi$›
  by (*rule WHILEIT-rule*) (*use assms H* **in** ‹*auto simp*: ›)
 finally show *?thesis* **.**
qed

lemma *lit-redundant-rec-spec*:
 fixes L :: ‹'*v literal*›
 assumes *invs*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (M, $N + NE$, $U + UE$, D')›
 assumes
  init-analysis: ‹*init-analysis* = [(L, C)]› **and**
  in-trail: ‹*Propagated* ($-L$) (*add-mset* ($-L$) C) ∈ *set* M› **and**
  ‹*conflict-min-analysis-inv M cach* ($N + NE + U + UE$) D› **and**
  L-D: ‹$L \in\#\ D$› **and**
  M-D: ‹$M \models as\ CNot\ D$› **and**
  unknown: ‹*cach* (*atm-of* L) = *SEEN-UNKNOWN*›
 shows
  ‹*lit-redundant-rec M* ($N + U$) *D cach init-analysis* ≤
   lit-redundant-rec-spec M ($N + U + NE + UE$) *D L*›
proof −
 let *?N* = ‹$N + NE + U + UE$›
 obtain *M2 M1* **where**
  M: ‹M = *M2* @ *Propagated* (− L) (*add-mset* (− L) C) # *M1*›
  using *split-list*[*OF in-trail*] **by** (*auto 5 5*)
 have ‹*a* @ *Propagated L mark* # *b* = *trail* (M, $N + NE$, $U + UE$, D') ⟶
    $b \models as\ CNot$ (*remove1-mset L mark*) ∧ $L \in\#$ *mark*› **for** L *mark a b*
  using *invs*
  unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
   $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
  by *fast*
 then have ‹$M1 \models as\ CNot\ C$›
  by (*force simp*: M)
 then have *M-C*: ‹$M \models as\ CNot\ C$›
  unfolding M **by** (*simp add*: *true-annots-append-l*)
 have ‹*set* (*get-all-mark-of-propagated* (*trail* (M, $N + NE$, $U + UE$, D')))
  ⊆ *set-mset* ($cdcl_W$*-restart-mset.clauses* (M, $N + NE$, $U + UE$, D'))›
  using *invs*
  unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
   $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*
  by *fast*
 then have ‹*add-mset* ($-L$) $C \in\#$ *?N*›
  using *in-trail* $cdcl_W$*-restart-mset.in-get-all-mark-of-propagated-in-trail*[*of* ‹*add-mset* ($-L$) C› M]
  by (*auto simp*: *clauses-def*)
 then have *NU-C*: ‹*?N* ⊨*pm add-mset* (− L) C›
  by *auto*
 have *n-d*: ‹*no-dup* M›
  using *invs*
  unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
   $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*

```
  by auto

let ?f = ‹λanalysis. fold-mset (+) D (snd `# mset analysis)›
define I' where
  ‹I' = (λ(cach :: 'v conflict-min-cach, analysis :: 'v conflict-min-analyse, b::bool).
     lit-redundant-inv M ?N D init-analysis (cach, analysis, b) ∧ M ⊨as CNot (?f analysis) ∧
     distinct (map (atm-of o fst) analysis))›
define R where
  ‹R = {((cach :: 'v conflict-min-cach, analysis :: 'v conflict-min-analyse, b::bool),
        (cach' :: 'v conflict-min-cach, analysis' :: 'v conflict-min-analyse, b' :: bool)).
        (analysis' ≠ [] ∧ (minimize-conflict-support M) (?f analysis') (?f analysis)) ∨
        (analysis' ≠ [] ∧ analysis = tl analysis' ∧ snd (hd analysis') = {#}) ∨
        (analysis' ≠ [] ∧ analysis = [])}›
have wf-R: ‹wf R›
proof −
  have R: ‹R =
          {((cach, analysis, b), (cach', analysis', b')).
             analysis' ≠ [] ∧analysis = []} ∪
          ({((cach, analysis, b), (cach', analysis', b')).
              analysis' ≠ [] ∧ (minimize-conflict-support M) (?f analysis') (?f analysis)} ∪
          {((cach, analysis, b), (cach', analysis', b')).
              analysis' ≠ [] ∧ analysis = tl analysis' ∧ snd (hd analysis') = {#}})›
    (is ‹- = ?end ∪ (?Min ∪ ?ana)›)
    unfolding R-def by auto
  have 1: ‹wf {((cach:: 'v conflict-min-cach, analysis:: 'v conflict-min-analyse, b::bool),
      (cach':: 'v conflict-min-cach, analysis':: 'v conflict-min-analyse, b'::bool)).
     length analysis < length analysis'}›
    using wf-if-measure-f[of ‹measure length›, of ‹λ(-, xs, -). xs›] apply auto
    apply (rule subst[of - - wf])
     prefer 2 apply assumption
    apply auto
    done

  have 2: ‹wf {(C', C).minimize-conflict-support M C C'}›
    by (rule wf-minimize-conflict-support[OF invs])
  from wf-if-measure-f[OF this, of ?f]
  have 2: ‹wf {(C', C). minimize-conflict-support M (?f C) (?f C')}›
    by auto
  from wf-fst-wf-pair[OF this, where 'b = bool]
  have ‹wf {((analysis':: 'v conflict-min-analyse, - :: bool),
           (analysis:: 'v conflict-min-analyse,  -:: bool)).
        (minimize-conflict-support M) (?f analysis) (?f analysis')}›
    by blast
  from wf-snd-wf-pair[OF this, where 'b = ‹'v conflict-min-cach›]
  have ‹wf {((M' :: 'v conflict-min-cach, N'), Ma, N).
    (case N' of
     (analysis' :: 'v conflict-min-analyse, - :: bool) ⇒
       λ(analysis, -).
          minimize-conflict-support M (fold-mset (+) D (snd `# mset analysis))
            (fold-mset (+) D (snd `# mset analysis'))) N}›
    by blast
  then have wf-Min: ‹wf ?Min›
    apply (rule wf-subset)
    by auto
  have wf-ana: ‹wf ?ana›
    by (rule wf-subset[OF 1])  auto
```

```
  have wf: ‹wf (?Min ∪ ?ana)›
    apply (rule wf-union-compatible)
    subgoal by (rule wf-Min)
    subgoal by (rule wf-ana)
    subgoal by (auto elim!: neq-NilE)
    done
  have wf-end: ‹wf ?end›
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then obtain f where f: ‹(f (Suc i), f i) ∈ ?end› for i
      unfolding wf-iff-no-infinite-down-chain by auto
    have ‹fst (snd (f (Suc 0))) = []›
      using f[of 0] by auto
    moreover have ‹fst (snd (f (Suc 0))) ≠ []›
      using f[of 1] by auto
    ultimately show False by blast
  qed
  show ?thesis
    unfolding R
    apply (rule wf-Un)
    subgoal by (rule wf-end)
    subgoal by (rule wf)
    subgoal by auto
    done
qed
have uL-M: ‹- L ∈ lits-of-l M›
  using in-trail by (force simp: lits-of-def)
then have init-I: ‹lit-redundant-inv M ?N D init-analysis (cach, init-analysis, False)›
  using assms NU-C Propagated-in-trail-entailed[OF invs in-trail]
  unfolding lit-redundant-inv-def
  by (auto simp: ac-simps)

have ‹(minimize-conflict-support M) D (remove1-mset L (C + D))›
  using minimize-conflict-support.resolve-propa[OF in-trail, of ‹remove1-mset L D›] L-D
  by (auto simp: ac-simps)

then have init-I': ‹I' (cach, init-analysis, False)›
  using M-D L-D M-C init-I unfolding I'-def by (auto simp: init-analysis)

have hd-M: ‹- fst (hd analyse) ∈ lits-of-l M›
  if
    inv-I': ‹I' s› and
    s: ‹s = (cach, s')› ‹s' = (analyse, ba)› and
    nempty: ‹analyse ≠ []›
  for analyse s s' ba cach
proof -
  have
    cach: ‹conflict-min-analysis-inv M cach ?N D› and
    ana: ‹conflict-min-analysis-stack M ?N D analyse› and
    stack: ‹conflict-min-analysis-stack M ?N D analyse› and
    stack-hd: ‹conflict-min-analysis-stack-hd M ?N D analyse› and
    last-analysis: ‹analyse ≠ [] ⟶ fst (last analyse) = fst (hd init-analysis)› and
    b: ‹analyse = [] ⟶ ba ⟶ cach (atm-of (fst (hd init-analysis))) = SEEN-REMOVABLE›
    using inv-I' unfolding lit-redundant-inv-def s I'-def by auto
  show ?thesis
    using stack-hd nempty by (cases analyse) auto
```

qed

have *all-removed*: ‹*lit-redundant-inv M ?N D init-analysis*
(*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*), *tl analysis*, *True*)› (**is** *?I*) **and**
all-removed-I': ‹*I'* (*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*), *tl analysis*, *True*)›
(**is** *?I'*) **and**
all-removed-J: ‹*lit-redundant-rec-loop-inv M* (*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*),
tl analysis, *True*)› (**is** *?J*)
if
inv-I': ‹*I' s*› **and** *inv-J*: ‹*lit-redundant-rec-loop-inv M s*›
‹*case s of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
s: ‹*s* = (*cach*, *s'*)›
‹*s'* = (*analysis*, *b*)› **and**
nemtpy-stack: ‹*analysis* ≠ []› **and**
finished: ‹*snd* (*hd analysis*) = {#}›
for *s cach s' analysis b*
proof −
obtain *L ana'* **where** *analysis*: ‹*analysis* = (*L*, {#}) # *ana'*›
using *nemtpy-stack finished* **by** (*cases analysis*) *auto*
have
cach: ‹*conflict-min-analysis-inv M cach ?N D*› **and**
ana: ‹*conflict-min-analysis-stack M ?N D analysis*› **and**
stack: ‹*conflict-min-analysis-stack M ?N D analysis*› **and**
stack-hd: ‹*conflict-min-analysis-stack-hd M ?N D analysis*› **and**
last-analysis: ‹*analysis* ≠ [] ⟶ *fst* (*last analysis*) = *fst* (*hd init-analysis*)› **and**
b: ‹*analysis* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*› **and**
dist: ‹*distinct* (*map* (*atm-of o fst*) *analysis*)›
using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def* **by** *auto*
obtain *C* **where**
NU-C: ‹*?N* ⊨*pm add-mset* (−*L*) *C*› **and**
IH: ‹⋀*K*. *K* ∈# *C* ⟹ *?N* ⊨*pm add-mset* (−*K*) (*filter-to-poslev M L D*) ∨
K ∈# *filter-to-poslev M L D*› **and**
index-K: ‹*K*∈#*C* ⟹ *index-in-trail M K* < *index-in-trail M L*› **and**
L-M: ‹−*L* ∈ *lits-of-l M*› **for** *K*
using *stack-hd* **unfolding** *analysis* **by** *auto*

have *NU-D*: ‹*?N* ⊨*pm add-mset* (− *fst* (*hd analysis*)) (*filter-to-poslev M* (*fst* (*hd analysis*)) *D*)›
using *conflict-minimize-intermediate-step-filter-to-poslev*[*OF - NU-C*, *simplified*, *OF index-K*]
IH
unfolding *analysis* **by** *auto*
have *ana'*: ‹*conflict-min-analysis-stack M ?N D* (*tl analysis*)›
using *ana* **by** (*auto simp*: *conflict-min-analysis-stack-tl*)
have ‹−*fst* (*hd analysis*) ∈ *lits-of-l M*›
using *L-M* **by** (*auto simp*: *analysis I'-def s ana*)
then have *cach'*:
‹*conflict-min-analysis-inv M* (*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*)) *?N D*›
using *NU-D n-d* **by** (*auto simp*: *conflict-min-analysis-inv-update-removable cach*)
have *stack-hd'*: ‹*conflict-min-analysis-stack-hd M ?N D ana'*›
proof (*cases* ‹*ana'* = []›)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then obtain *L' C' ana''* **where** *ana''*: ‹*ana'* = (*L'*, *C'*) # *ana''*›
by (*cases ana'*; *cases* ‹*hd ana'*›) *auto*
then obtain *E'* **where**

NU-E': ‹*?N* ⊨*pm add-mset* (− *L'*) *E'*› **and**
‹∀ *K*∈#*E'* − *add-mset L C'*. *?N* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M L' D*) ∨
K ∈# *filter-to-poslev M L' D*› **and**
index-C': ‹∀ *K*∈#*E'*. *index-in-trail M K* < *index-in-trail M L'*› **and**
index-L'-L: ‹*index-in-trail M L* < *index-in-trail M L'*› **and**
C'-E': ‹*C'* ⊆# *E'*› **and**
uL'-M: ‹− *L'* ∈ *lits-of-l M*›
using *stack* **by** (*auto simp*: *analysis ana''*)
moreover have ‹*?N* ⊨*pm add-mset* (− *L*) (*filter-to-poslev M L D*)›
using *NU-D analysis* **by** *auto*
moreover have ‹*K* ∈# *E'* − *C'* ⟹ *K* ∈# *E'* − *add-mset L C'* ∨ *K* = *L*› **for** *K*
by (*cases* ‹*L* ∈# *E'*›)
(*fastforce simp*: *minus-notin-trivial dest*!: *multi-member-split*[*of L*]
dest: *in-remove1-msetI*)+
moreover have ‹*K* ∈# *E'* − *C'* ⟹ *index-in-trail M K* ≤ *index-in-trail M L'*› **for** *K*
by (*meson in-diffD index-C' less-or-eq-imp-le*)
ultimately have ‹*K* ∈# *E'* − *C'* ⟹ *?N* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M L' D*) ∨
K ∈# *filter-to-poslev M L' D*› **for** *K*
using *filter-to-poslev-mono-entailement-add-mset*[*of M K L'*]
filter-to-poslev-mono[*of M L L'*]
by *fastforce*
then show *?thesis*
using *NU-E' uL'-M index-C' C'-E'* **unfolding** *ana''* **by** (*auto intro*!: *exI*[*of - E'*])
qed

have ‹*fst* (*hd init-analysis*) = *fst* (*last* (*tl analysis*))› **if** ‹*tl analysis* ≠ []›
using *last-analysis tl-last*[*symmetric*, *OF that*] *that* **unfolding** *ana'* **by** *auto*
then show *?I*
using *ana' cach' last-analysis stack-hd' dist* **unfolding** *lit-redundant-inv-def*
by (*cases ana'*; *auto simp*: *analysis atm-of-eq-atm-of split*: *if-splits*)
then show *I'*: *?I'*
using *inv-I'* **unfolding** *I'-def s* **by** (*auto simp*: *analysis*)
have ‹*distinct* (*map* (λ*x*. − *fst x*) (*tl analysis*))›
using *dist distinct-mapI*[*of* ‹*atm-of o uminus*› ‹*map* (*uminus o fst*) (*tl analysis*)›]
conflict-min-analysis-stack-neg[*OF ana'*] **by** (*auto simp*: *comp-def map-tl*
simp flip: *distinct-mset-image-mset*)
then show *?J*
using *inv-J* **unfolding** *lit-redundant-rec-loop-inv-def prod.case s*
apply (*subst distinct-subseteq-iff*[*symmetric*])
using *conflict-min-analysis-stack-neg*[*OF ana'*] *no-dup-distinct*[*OF n-d*] *dist*
by (*auto simp*: *comp-def entails-CNot-negate-ann-lits negate-ann-lits-def*
analysis ana' rev-image-eqI
simp flip: *distinct-mset-image-mset*)
qed
have *all-removed-R*:
‹((*cach*(*atm-of* (*fst* (*hd analyse*)) := *SEEN-REMOVABLE*), *tl analyse*, *True*), *s*) ∈ *R*›
if
s: ‹*s* = (*cach*, *s'*)› ‹*s'* = (*analyse*, *b*)› **and**
nempty: ‹*analyse* ≠ []› **and**
finished: ‹*snd* (*hd analyse*) = {#}›
for *s cach s' analyse b*
using *nempty finished* **unfolding** *R-def s* **by** *auto*
have
seen-removable-inv: ‹*lit-redundant-inv M ?N D init-analysis* (*cach*, *ana*, *False*)› (**is** *?I*) **and**
seen-removable-I': ‹*I'* (*cach*, *ana*, *False*)› (**is** *?I'*) **and**
seen-removable-R: ‹((*cach*, *ana*, *False*), *s*) ∈ *R*› (**is** *?R*) **and**

seen-removable-J: ‹*lit-redundant-rec-loop-inv M* (*cach*, *ana*, *False*)› (**is** *?J*)
if
inv-I': ‹*I' s*› **and** *inv-J*: ‹*lit-redundant-rec-loop-inv M s*› **and**
cond: ‹*case s of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
s: ‹*s* = (*cach*, *s'*)› ‹*s'* = (*analyse*, *b*)› ‹*x* = (*L*, *ana*)› **and**
nemtpy-stack: ‹*analyse* ≠ []› **and**
‹*snd* (*hd analyse*) ≠ {#}› **and**
next-lit: ‹*case x of*
(*L*, *ana*) ⇒ *L* ∈# *snd* (*hd analyse*) ∧ *tl ana* = *tl analyse* ∧ *ana* ≠ [] ∧
hd ana = (*fst* (*hd analyse*), *remove1-mset L* (*snd* (*hd analyse*)))› **and**
lev0-removable: ‹*get-level M L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ∨ *L* ∈# *D*›
for *s cach s' analyse b x L ana*
proof −
obtain *K C ana'* **where** *analysis*: ‹*analyse* = (*K*, *C*) # *ana'*›
using *nemtpy-stack* **by** (*cases analyse*) *auto*
have *ana'*: ‹*ana* = (*K*, *remove1-mset L C*) # *ana'*› **and** *L-C*: ‹*L* ∈# *C*›
using *next-lit* **unfolding** *s* **by** (*cases ana*; *auto simp*: *analysis*)+
have
cach: ‹*conflict-min-analysis-inv M cach* (*?N*) *D*› **and**
ana: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack-hd: ‹*conflict-min-analysis-stack-hd M ?N D analyse*› **and**
last-analysis: ‹*analyse* ≠ [] ⟶ *fst* (*last analyse*) = *fst* (*hd init-analysis*)› **and**
b: ‹*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*› **and**
dist: ‹*distinct* (*map* (*atm-of* ∘ *fst*) *analyse*)›
using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def prod.case* **by** *auto*

have *last-analysis'*: ‹*ana* ≠ [] ⟹ *fst* (*hd init-analysis*) = *fst* (*last ana*)›
using *last-analysis next-lit* **unfolding** *analysis s*
by (*cases ana*) (*auto split*: *if-splits*)
have *uL-M*: ‹−*L* ∈ *lits-of-l M*›
using *inv-I' L-C* **unfolding** *analysis ana s I'-def*
by (*auto dest*!: *multi-member-split*)
have *uK-M*: ‹− *K* ∈ *lits-of-l M*›
using *stack-hd* **unfolding** *analysis* **by** *auto*
consider
(*lev0*) ‹*get-level M L* = *0*› |
(*Removable*) ‹*cach* (*atm-of L*) = *SEEN-REMOVABLE*› |
(*in-D*) ‹*L* ∈# *D*›
using *lev0-removable* **by** *fast*
then have *H*: ‹∃ *CK*. *?N* ⊨*pm add-mset* (− *K*) *CK* ∧
(∀ *Ka*∈#*CK* − *remove1-mset L C*. *?N* ⊨*pm* (*filter-to-poslev M K D*) + {#− *Ka*#} ∨
Ka ∈# *filter-to-poslev M K D*) ∧
(∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*) ∧
remove1-mset L C ⊆# *CK*›
(**is** ‹∃ *C*. *?P C*›)
proof *cases*
case *Removable*
then have *L*: ‹*?N* ⊨*pm add-mset* (− *L*) (*filter-to-poslev M L D*)›
using *cach uL-M* **unfolding** *conflict-min-analysis-inv-def* **by** *auto*
obtain *CK* **where**
‹*?N* ⊨*pm add-mset* (− *K*) *CK*› **and**
‹∀ *K'*∈#*CK* − *C*. *?N* ⊨*pm* (*filter-to-poslev M K D*) + {#− *K'*#} ∨ *K'* ∈# *filter-to-poslev M K D*› **and**
index-CK: ‹∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*› **and**
C-CK: ‹*C* ⊆# *CK*›

using *stack-hd* **unfolding** *analysis* **by** *auto*
moreover have ‹*remove1-mset L C* ⊆# *CK*›
using *C-CK* **by** (*meson diff-subset-eq-self subset-mset.dual-order.trans*)
moreover have ‹*index-in-trail M L* < *index-in-trail M K*›
using *index-CK C-CK L-C* **unfolding** *analysis ana'* **by** *auto*
moreover have *index-CK'*: ‹∀ *Ka*∈#*CK*. *index-in-trail M Ka* ≤ *index-in-trail M K*›
using *index-CK* **by** *auto*
ultimately have ‹*?P CK*›
using *filter-to-poslev-mono-entailement-add-mset*[*of M - -*]
filter-to-poslev-mono[*of M K L*]
using *L L-C C-CK* **by** (*auto simp*: *minus-remove1-mset-if*)
then show *?thesis* **by** *blast*
next
assume *lev0*: ‹*get-level M L* = *0*›
have ‹*M* ⊨*as CNot* (*?f analyse*)›
using *inv-I'* **unfolding** *I'-def s* **by** *auto*
then have ‹−*L* ∈ *lits-of-l M*›
using *next-lit* **unfolding** *analysis s* **by** (*auto dest*: *multi-member-split*)
then have ‹*?N* ⊨*pm* {#−*L*#}›
using *lev0 cdcl*$_W$*-restart-mset.literals-of-level0-entailed*[*OF invs*, *of* ‹−*L*›]
by (*auto simp*: *clauses-def ac-simps*)
moreover obtain *CK* **where**
‹*?N* ⊨*pm add-mset* (− *K*) *CK*› **and**
‹∀ *K'*∈#*CK* − *C*. *?N* ⊨*pm* (*filter-to-poslev M K D*) + {#− *K'*#} ∨ *K'* ∈# *filter-to-poslev M K D*› **and**
‹∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*› **and**
C-CK: ‹*C* ⊆# *CK*›
using *stack-hd* **unfolding** *analysis* **by** *auto*
moreover have ‹*remove1-mset L C* ⊆# *CK*›
using *C-CK* **by** (*meson diff-subset-eq-self subset-mset.order-trans*)
ultimately have ‹*?P CK*›
by (*auto simp*: *minus-remove1-mset-if intro*: *conflict-minimize-intermediate-step*)
then show *?thesis* **by** *blast*
next
case *in-D*
obtain *CK* **where**
‹*?N* ⊨*pm add-mset* (− *K*) *CK*› **and**
‹∀ *Ka*∈#*CK* − *C*. *?N* ⊨*pm* (*filter-to-poslev M K D*) + {#− *Ka*#} ∨ *Ka* ∈# *filter-to-poslev M K D*› **and**
index-CK: ‹∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*› **and**
C-CK: ‹*C* ⊆# *CK*›
using *stack-hd* **unfolding** *analysis* **by** *auto*
moreover have ‹*remove1-mset L C* ⊆# *CK*›
using *C-CK* **by** (*meson diff-subset-eq-self subset-mset.order-trans*)
moreover have ‹*L* ∈# *filter-to-poslev M K D*›
using *in-D L-C index-CK C-CK* **by** (*fastforce simp*: *filter-to-poslev-def*)

ultimately have ‹*?P CK*›
using *in-D*
using *filter-to-poslev-mono-entailement-add-mset*[*of M L K*]
by (*auto simp*: *minus-remove1-mset-if dest*!:
intro: *conflict-minimize-intermediate-step*)
then show *?thesis* **by** *blast*
qed note *H* = *this*
have *stack'*: ‹*conflict-min-analysis-stack M ?N D ana*›
using *stack* **unfolding** *ana' analysis* **by** (*cases ana'*) *auto*

```
  have stack-hd': ‹conflict-min-analysis-stack-hd M ?N D ana›
    using H uL-M uK-M unfolding ana' by auto

  show ?I
    using last-analysis' cach stack' stack-hd' unfolding lit-redundant-inv-def s
    by auto
  have ‹M ⊨as CNot (?f ana)›
    using inv-I' unfolding I'-def s ana analysis ana'
    by (cases ‹L ∈# C›) (auto dest!: multi-member-split)
  then show ?I'
    using inv-I' ‹?I› unfolding I'-def s by (auto simp: analysis ana')

  show ?R
    using next-lit
    unfolding R-def s by (auto simp: ana' analysis dest!: multi-member-split
        intro: minimize-conflict-support.intros)
  have ‹distinct (map (λx. - fst x) ana)›
    using dist distinct-mapI[of ‹atm-of o uminus› ‹map (uminus o fst) (tl analyse)›]
     conflict-min-analysis-stack-neg[OF stack'] by (auto simp: comp-def map-tl
        analysis ana'
     simp flip: distinct-mset-image-mset)
  then show ?J
    using inv-J unfolding lit-redundant-rec-loop-inv-def prod.case s
    apply (subst distinct-subseteq-iff[symmetric])
    using conflict-min-analysis-stack-neg[OF stack'] no-dup-distinct[OF n-d]
    apply (auto simp: comp-def entails-CNot-negate-ann-lits negate-ann-lits-def
     simp flip: distinct-mset-image-mset)
   apply (force simp add: analysis ana ana')
   done

qed
have
  failed-I: ‹lit-redundant-inv M ?N D init-analysis
     (cach', [], False)› (is ?I) and
  failed-I': ‹I' (cach', [], False)› (is ?I') and
  failed-R: ‹((cach', [], False), s) ∈ R› (is ?R) and
  failed-J: ‹lit-redundant-rec-loop-inv M (cach', [], False)› (is ?J)
  if
    inv-I': ‹I' s› and inv-J: ‹lit-redundant-rec-loop-inv M s› and
    cond: ‹case s of (cach, analyse, b) ⇒ analyse ≠ []› and
    s: ‹s = (cach, s')› ‹s' = (analyse, b)› and
    nempty: ‹analyse ≠ []› and
    ‹snd (hd analyse) ≠ {#}› and
    ‹case x of (L, ana) ⇒ L ∈# snd (hd analyse) ∧ tl ana = tl analyse ∧
      ana ≠ [] ∧ hd ana = (fst (hd analyse), remove1-mset L (snd (hd analyse)))› and
    ‹x = (L, ana)› and
    ‹¬ (get-level M L = 0 ∨ cach (atm-of L) = SEEN-REMOVABLE ∨ L ∈# D)› and
    cach-update: ‹∀ L. cach' L = SEEN-REMOVABLE ⟶ cach L = SEEN-REMOVABLE›
  for s cach s' analyse b x L ana E cach'
proof -
  have
    cach: ‹conflict-min-analysis-inv M cach ?N D› and
    ana: ‹conflict-min-analysis-stack M ?N D analyse› and
    stack: ‹conflict-min-analysis-stack M ?N D analyse› and
    last-analysis: ‹analyse ≠ [] ⟶ fst (last analyse) = fst (hd init-analysis)› and
    b: ‹analyse = [] ⟶ b ⟶ cach (atm-of (fst (hd init-analysis))) = SEEN-REMOVABLE›
```

using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def* **by** *auto*
have ‹*conflict-min-analysis-inv M cach' ?N D*›
using *cach cach-update* **by** (*auto simp*: *conflict-min-analysis-inv-def*)
moreover have ‹*conflict-min-analysis-stack M ?N D* []›
by *simp*
ultimately show *?I*
unfolding *lit-redundant-inv-def* **by** *simp*
then show *?I'*
using *M-D* **unfolding** *I'-def* **by** *auto*
show *?R*
using *nempty* **unfolding** *R-def s* **by** *auto*
show *?J*
by (*auto simp*: *lit-redundant-rec-loop-inv-def*)
qed
have *is-propagation-inv*: ‹*lit-redundant-inv M ?N D init-analysis*
(*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*)› (**is** *?I*) **and**
is-propagation-I': ‹*I'* (*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*)› (**is** *?I'*) **and**
is-propagation-R: ‹((*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*), *s*) ∈ *R*› (**is** *?R*) **and**
is-propagation-dist: ‹*distinct-mset E'*› (**is** *?dist*) **and**
is-propagation-tauto: ‹¬*tautology E'*› (**is** *?tauto*) **and**
is-propagation-J': ‹*lit-redundant-rec-loop-inv M* (*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*)› (**is** *?J*)
if
inv-I': ‹*I' s*› **and** *inv-J*: ‹*lit-redundant-rec-loop-inv M s*› **and**
‹*case s of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
s: ‹*s* = (*cach*, *s'*)› ‹*s'* = (*analyse*, *b*)› ‹*x* = (*L*, *ana*)› **and**
nemtpy-stack: ‹*analyse* ≠ []› **and**
‹*snd* (*hd analyse*) ≠ {#}› **and**
next-lit: ‹*case x of* (*L*, *ana*) ⇒
L ∈# *snd* (*hd analyse*) ∧
tl ana = *tl analyse* ∧
ana ≠ [] ∧
hd ana =
(*fst* (*hd analyse*),
remove1-mset L (*snd* (*hd analyse*)))› **and**
‹¬ (*get-level M L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ∨ *L* ∈# *D*)› **and**
E: ‹*E* ≠ *None* ⟶ *Propagated* (− *L*) (*the E*) ∈ *set M*› ‹*E* = *Some E'*› **and**
st: ‹*cach* (*atm-of L*) = *SEEN-UNKNOWN*›
for *s cach s' analyse b x L ana E E'*
proof −
obtain *K C ana'* **where** *analysis*: ‹*analyse* = (*K*, *C*) # *ana'*›
using *nemtpy-stack* **by** (*cases analyse*) *auto*
have *ana'*: ‹*ana* = (*K*, *remove1-mset L C*) # *ana'*›
using *next-lit* **unfolding** *s* **by** (*cases ana*) (*auto simp*: *analysis*)
have
cach: ‹*conflict-min-analysis-inv M cach ?N D*› **and**
ana: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack-hd: ‹*conflict-min-analysis-stack-hd M ?N D analyse*› **and**
last-analysis: ‹*analyse* ≠ [] ⟶ *fst* (*last analyse*) = *fst* (*hd init-analysis*)› **and**
b: ‹*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*› **and**
dist-ana: ‹*distinct* (*map* (*atm-of* ∘ *fst*) *analyse*)›
using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def* **by** *auto*
have
NU-E: ‹*?N* ⊨*pm add-mset* (− *L*) (*remove1-mset* (−*L*) *E'*)› **and**
uL-E: ‹−*L* ∈# *E'*› **and**

M-E': ‹*M* ⊨*as CNot* (*remove1-mset* (− *L*) *E'*)› **and**
tauto: ‹¬ *tautology E'*› **and**
dist: ‹*distinct-mset E'*› **and**
lev-E': ‹*K* ∈# *remove1-mset* (− *L*) *E'* ⟹ *index-in-trail M K* < *index-in-trail M* (− *L*)› **for** *K*
using *Propagated-in-trail-entailed*[*OF invs*, *of* ‹−*L*› *E'*] *E* **by** (*auto simp*: *ac-simps*)
have *uL-M*: ‹− *L* ∈ *lits-of-l M*›
using *next-lit inv-I'* **unfolding** *s analysis I'-def* **by** (*auto dest*!: *multi-member-split*)
obtain *C'* **where**
‹*?N* ⊨*pm add-mset* (− *K*) *C'*› **and**
‹∀ *Ka*∈#*C'*. *index-in-trail M Ka* < *index-in-trail M K*› **and**
‹*C* ⊆# *C'*› **and**
‹∀ *Ka*∈#*C'* − *C*.
?N ⊨*pm add-mset* (− *Ka*) (*filter-to-poslev M K D*) ∨
Ka ∈# *filter-to-poslev M K D*› **and**
uK-M: ‹− *K* ∈ *lits-of-l M*›
using *stack-hd*
unfolding *s ana'*[*symmetric*]
by (*auto simp*: *analysis ana' conflict-min-analysis-stack-change-hd*)

then have *cmas*: ‹*conflict-min-analysis-stack M ?N D* ((*L*, *remove1-mset* (−*L*) *E'*) # *ana*)›
using *stack E next-lit NU-E uL-E uL-M*
filter-to-poslev-mono-entailement-add-mset[*of M - -* ‹*set-mset ?N*› *- D*]
filter-to-poslev-mono[*of M*] *uK-M*
unfolding *s ana'*[*symmetric*] *prod.case*
by (*auto simp*: *analysis ana' conflict-min-analysis-stack-change-hd*)
moreover have ‹*conflict-min-analysis-stack-hd M ?N D* ((*L*, *remove1-mset* (− *L*) *E'*) # *ana*)›
using *NU-E lev-E' uL-M* **by** (*auto intro*!:*exI*[*of -* ‹*remove1-mset* (− *L*) *E'*›])
moreover have ‹*fst* (*hd init-analysis*) = *fst* (*last* ((*L*, *remove1-mset* (− *L*) *E'*) # *ana*))›
using *last-analysis* **unfolding** *analysis ana'* **by** *auto*
ultimately show *?I*
using *cach b* **unfolding** *lit-redundant-inv-def analysis* **by** *auto*
moreover have ‹*L* ≠ *K*›
using *cmas*
unfolding *ana' conflict-min-analysis-stack.simps*(*3*) **by** *blast*
moreover have ‹*L* ≠ −*K*›
using *cmas*
unfolding *ana' conflict-min-analysis-stack.simps*(*3*) **by** *auto*
ultimately show *?I'*
using *M-E' inv-I' conflict-min-analysis-stack-distinct-fst*[*OF cmas*]
unfolding *I'-def s ana analysis ana'*
by (*auto simp*: *true-annot-CNot-diff atm-of-eq-atm-of uminus-lit-swap*)

have ‹*L* ∈# *C*› **and** *in-trail*: ‹*Propagated* (− *L*) (*the E*) ∈ *set M*› **and** *EE'*: ‹*the E* = *E'*›
using *next-lit E* **by** (*auto simp*: *analysis ana' s*)
then obtain *E''* *C'* **where**
E': ‹*E'* = *add-mset* (−*L*) *E''*› **and**
C: ‹*C* = *add-mset L C'*›
using *uL-E* **by** (*blast dest*: *multi-member-split*)

have ‹*minimize-conflict-support M* (*C* + *fold-mset* (+) *D* (*snd* '# *mset ana'*))
(*remove1-mset* (− *L*) *E'* + (*remove1-mset L C* + *fold-mset* (+) *D* (*snd* '# *mset ana'*)))›
using *minimize-conflict-support.resolve-propa*[*OF in-trail*,
of ‹*C'* + *fold-mset* (+) *D* (*snd* '# *mset ana'*)›]
unfolding *C E' EE'*
by (*auto simp*: *ac-simps*)

```
  then show ?R
    using nemtpy-stack unfolding s analysis ana' by (auto simp: R-def
        intro: resolve-propa)
  have ‹distinct (map (λx. − fst x) analyse)›
    using dist-ana distinct-mapI[of ‹atm-of o uminus› ‹map (uminus o fst) analyse›]
     conflict-min-analysis-stack-neg[OF cmas] unfolding analysis ana'
     by (auto simp: comp-def map-tl
        simp flip: distinct-mset-image-mset)
  then show ?J
    using inv-J st unfolding lit-redundant-rec-loop-inv-def prod.case s
    apply (intro conjI)
    apply (subst distinct-subseteq-iff[symmetric])
    using conflict-min-analysis-stack-neg[OF cmas] no-dup-distinct[OF n-d] uL-M
      ‹L ≠ −K› ‹L ≠ K›  conflict-min-analysis-stack-distinct-fst[OF cmas]
    apply (auto simp: comp-def entails-CNot-negate-ann-lits
      negate-ann-lits-def lits-of-def uminus-lit-swap
     simp flip: distinct-mset-image-mset)[2]
  using that by (clarsimp-all simp add: analysis ana')

  show ?tauto
    using tauto .
  show ?dist
    using dist .
qed
have length-aa-le: ‹length aa ≤ length M›
  if
    ‹I' s› and
    ‹case s of (cach, analyse, b) ⇒ analyse ≠ []› and
    ‹s = (a, b)› and
    ‹b = (aa, ba)› and
    ‹aa ≠ []› for s a b aa ba
proof −
  have ‹M ⊨as CNot (fst `# mset aa)› and ‹distinct (map (atm-of o fst) aa)› and
     ‹distinct (map fst aa)› and
    ‹conflict-min-analysis-stack M (N + NE + U + UE) D aa›
    using distinct-mapI[of ‹atm-of› ‹map fst aa›]
    using that by (auto simp: I'-def lit-redundant-inv-def
      dest: conflict-min-analysis-stack-neg)

  then have ‹set (map fst aa) ⊆ uminus ` lits-of-l M›
    by (auto simp: true-annots-true-cls-def-iff-negation-in-model lits-of-def image-image
        uminus-lit-swap
       dest!: multi-member-split)
  from card-mono[OF - this] have ‹length (map fst aa) ≤ length M›
    using ‹distinct (map (fst) aa)› distinct-card[of ‹map fst aa›] n-d
   by (auto simp: card-image[OF lit-of-inj-on-no-dup[OF n-d]] lits-of-def image-image
      distinct-card[OF no-dup-imp-distinct])
  then show ‹?thesis› by auto
qed

show ?thesis
  unfolding lit-redundant-rec-def lit-redundant-rec-spec-def mark-failed-lits-def
    get-literal-and-remove-of-analyse-def get-propagation-reason-def
  apply (refine-vcg WHILEIT-rule-stronger-inv[where R = R and I' = I'])
    — Well-foundness
  subgoal by (rule wf-R)
```

subgoal using *assms* **by** (*auto simp*: *lit-redundant-rec-loop-inv-def lits-of-def*
dest!: *multi-member-split*)
subgoal by (*rule init-I′*)
subgoal by *auto*
subgoal by (*rule length-aa-le*)
— Assertion:
subgoal by (*rule hd-M*)
— We finished one stage:
subgoal by (*rule all-removed-J*)
subgoal by (*rule all-removed-I′*)
subgoal by (*rule all-removed-R*)
— Assertion:
subgoal for *s cach s′ analyse ba*
by (*cases* ‹*analyse*›) (*auto simp*: *I′-def dest*!: *multi-member-split*)

— Cached or level 0:
subgoal by (*rule seen-removable-J*)
subgoal by (*rule seen-removable-I′*)
subgoal by (*rule seen-removable-R*)
— Failed:
subgoal by (*rule failed-J*)
subgoal by (*rule failed-I′*)
subgoal by (*rule failed-R*)
subgoal for *s a b aa ba x ab bb xa* **by** (*cases* ‹*a* (*atm-of ab*)›) *auto*
subgoal by (*rule failed-J*)
subgoal by (*rule failed-I′*)
subgoal by (*rule failed-R*)
— The literal was propagated:
subgoal by (*rule is-propagation-dist*)
subgoal by (*rule is-propagation-tauto*)
subgoal by (*rule is-propagation-J′*)
subgoal by (*rule is-propagation-I′*)
subgoal by (*rule is-propagation-R*)
— End of Loop invariant:
subgoal
using *uL-M* **by** (*auto simp*: *lit-redundant-inv-def conflict-min-analysis-inv-def init-analysis*
I′-def ac-simps)
subgoal by (*auto simp*: *lit-redundant-inv-def conflict-min-analysis-inv-def init-analysis*
I′-def ac-simps)
done
qed

definition *literal-redundant-spec* **where**
‹*literal-redundant-spec M NU D L* =
SPEC(λ(*cach*, *analysis*, *b*). (*b* ⟶ *NU* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M L D*)) ∧
conflict-min-analysis-inv M cach NU D)›

definition *literal-redundant* **where**
‹*literal-redundant M NU D cach L* = *do* {
ASSERT(−*L* ∈ *lits-of-l M*);
if get-level M L = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE*
then RETURN (*cach*, [], *True*)
else if cach (*atm-of L*) = *SEEN-FAILED*
then RETURN (*cach*, [], *False*)
else do {
C ← *get-propagation-reason M* (−*L*);

```
      case C of
        Some C ⇒ do{
    ASSERT(distinct-mset C ∧ ¬tautology C);
    lit-redundant-rec M NU D cach [(L, C − {#−L#})]}
      | None ⇒ do {
          RETURN (cach, [], False)
    }
    }
  }›

lemma true-clss-cls-add-self: ‹NU ⊨p D' + D' ⟷ NU ⊨p D'›
  by (metis subset-mset.sup-idem true-clss-cls-sup-iff-add)

lemma true-clss-cls-add-add-mset-self: ‹NU ⊨p add-mset L (D' + D') ⟷ NU ⊨p add-mset L D'›
  using true-clss-cls-add-self true-clss-cls-mono-r by fastforce


lemma filter-to-poslev-remove1:
  ‹filter-to-poslev M L (remove1-mset K D) =
     (if index-in-trail M K ≤ index-in-trail M L then remove1-mset K (filter-to-poslev M L D)
   else filter-to-poslev M L D)›
  unfolding filter-to-poslev-def
  by (auto simp: multiset-filter-mono2)


lemma filter-to-poslev-add-mset:
  ‹filter-to-poslev M L (add-mset K D) =
     (if index-in-trail M K < index-in-trail M L then add-mset K (filter-to-poslev M L D)
   else filter-to-poslev M L D)›
  unfolding filter-to-poslev-def
  by (auto simp: multiset-filter-mono2)

lemma filter-to-poslev-conflict-min-analysis-inv:
  assumes
    L-D: ‹L ∈# D› and
    NU-uLD: ‹N+U ⊨pm add-mset (−L) (filter-to-poslev M L D)› and
    inv: ‹conflict-min-analysis-inv M cach (N + U) D›
  shows ‹conflict-min-analysis-inv M cach (N + U) (remove1-mset L D)›
  unfolding conflict-min-analysis-inv-alt-def
proof (intro allI impI)
  fix K
  assume ‹−K ∈ lits-of-l M› and ‹cach (atm-of K) = SEEN-REMOVABLE›
  then have K: ‹N + U ⊨pm add-mset (− K) (filter-to-poslev M K D)›
    using inv unfolding conflict-min-analysis-inv-alt-def by blast
  obtain D' where D: ‹D = add-mset L D'›
    using multi-member-split[OF L-D] by blast
  have ‹N + U ⊨pm add-mset (− K) (filter-to-poslev M K D')›
  proof (cases ‹index-in-trail M L < index-in-trail M K›)
    case True
    then have ‹N + U ⊨pm add-mset (− K) (add-mset L (filter-to-poslev M K D'))›
      using K by (auto simp: filter-to-poslev-add-mset D)
    then have 1: ‹N + U ⊨pm add-mset L (add-mset (−K) (filter-to-poslev M K D'))›
      by (simp add: add-mset-commute)
    have H: ‹index-in-trail M L ≤ index-in-trail M K›
      using True by simp
    have 2: ‹N+U ⊨pm add-mset (−L) (filter-to-poslev M K D')›
```

```
    using filter-to-poslev-mono-entailement-add-mset[OF H] NU-uLD
    by (metis (no-types, opaque-lifting) D NU-uLD filter-to-poslev-add-mset
      order-less-irrefl)
  show ?thesis
    using true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or[OF 2 1]
    by (auto simp: true-clss-cls-add-add-mset-self)
 next
  case False
  then show ?thesis using K by (auto simp: filter-to-poslev-add-mset D split: if-splits)
 qed
 then show ‹N + U ⊨pm add-mset (− K) (filter-to-poslev M K (remove1-mset L D))›
  by (simp add: D)
qed

lemma can-filter-to-poslev-can-remove:
 assumes
  L-D: ‹L ∈# D› and
  ‹M ⊨as CNot D› and
  NU-D: ‹NU ⊨pm D› and
  NU-uLD: ‹NU ⊨pm add-mset (−L) (filter-to-poslev M L D)›
 shows ‹NU ⊨pm remove1-mset L D›
proof −
 obtain D′ where
  D: ‹D = add-mset L D′›
  using multi-member-split[OF L-D] by blast
 then have ‹filter-to-poslev M L D ⊆# D′›
  by (auto simp: filter-to-poslev-def)
 then have ‹NU ⊨pm add-mset (−L) D′›
  using NU-uLD true-clss-cls-mono-r[of - ‹ add-mset (− L) (filter-to-poslev M (−L) D)› ]
  by (auto simp: mset-subset-eq-exists-conv)
 from true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or[OF this, of D′]
 show ‹NU ⊨pm remove1-mset L D›
  using NU-D by (auto simp: D true-clss-cls-add-self)
qed

lemma literal-redundant-spec:
 fixes L :: ‹'v literal›
 assumes invs: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (M, N + NE, U + UE, D′)›
 assumes
  inv: ‹conflict-min-analysis-inv M cach (N + NE + U + UE) D› and
  L-D: ‹L ∈# D› and
  M-D: ‹M ⊨as CNot D›
 shows
  ‹literal-redundant M (N + U) D cach L ≤ literal-redundant-spec M (N + U + NE + UE) D L›
proof −
 have lit-redundant-rec: ‹lit-redundant-rec M (N + U) D cach [(L, remove1-mset (− L) E′)]
    ≤ literal-redundant-spec M (N + U + NE + UE) D L›
  if
   E: ‹E ≠ None ⟶ Propagated (− L) (the E) ∈ set M› and
   E′: ‹E = Some E′› and
   failed: ‹¬ (get-level M L = 0 ∨ cach (atm-of L) = SEEN-REMOVABLE)›
    ‹cach (atm-of L) ≠ SEEN-FAILED›
  for E E′
 proof −
  have
   [simp]: ‹−L ∈# E′› and
```

```
    in-trail: ‹Propagated (− L) (add-mset (−L) (remove1-mset (−L) E′)) ∈ set M›
    using Propagated-in-trail-entailed[OF invs, of ‹−L› E′] E E′
    by auto
  have H: ‹lit-redundant-rec-spec M (N + U + NE + UE) D L ≤
    literal-redundant-spec M (N + U + NE + UE) D L›
    by (auto simp: lit-redundant-rec-spec-def literal-redundant-spec-def ac-simps)
  show ?thesis
    apply (rule order.trans)
     apply (rule lit-redundant-rec-spec[OF invs - in-trail])
    subgoal ..
    subgoal by (rule inv)
    subgoal using assms by fast
    subgoal by (rule M-D)
    subgoal using failed by (cases ‹cach (atm-of L)›) auto
    subgoal unfolding literal-redundant-spec-def[symmetric] by (rule H)
    done
qed

have
  L-dist: ‹distinct-mset (C)› and
  L-tauto: ‹¬tautology C›
if
  in-trail: ‹Propagated (− L) C ∈ set M›
  for C
  using that
  Propagated-in-trail-entailed[of M ‹N+NE› ‹U+UE› ‹D′› ‹−L› ‹C›] invs
  by (auto simp: )
have uL-M: ‹−L ∈ lits-of-l M›
  using L-D M-D by (auto dest!: multi-member-split)
show ?thesis
  unfolding literal-redundant-def get-propagation-reason-def literal-redundant-spec-def
  apply (refine-vcg)
  subgoal using uL-M .
  subgoal
    using inv uL-M cdcl_W-restart-mset.literals-of-level0-entailed[OF invs, of ‹−L›]
      true-clss-cls-mono-r′
    by (fastforce simp: mark-failed-lits-def conflict-min-analysis-inv-def
        clauses-def ac-simps)
  subgoal using inv by (auto simp: ac-simps)
  subgoal by auto
  subgoal using inv by (auto simp: ac-simps)
  subgoal using inv by (auto simp: mark-failed-lits-def conflict-min-analysis-inv-def)
  subgoal using inv by (auto simp: mark-failed-lits-def conflict-min-analysis-inv-def ac-simps)
  subgoal using L-dist by simp
  subgoal using L-tauto by simp
  subgoal for E E′
    unfolding literal-redundant-spec-def[symmetric]
    by (rule lit-redundant-rec)
  done
qed

definition set-all-to-list where
  ‹set-all-to-list e ys = do {
    S ← WHILE^λ(i, xs). i ≤ length xs ∧ (∀ x ∈ set (take i xs). x = e) ∧ length xs = length ys
      (λ(i, xs). i < length xs)
      (λ(i, xs). do {
```

```
      ASSERT(i < length xs);
      RETURN(i+1, xs[i := e])
     })
    (0, ys);
  RETURN (snd S)
  }›
```

lemma
‹*set-all-to-list e ys* ≤ *SPEC*(λ*xs*. *length xs* = *length ys* ∧ (∀ *x* ∈ *set xs*. *x* = *e*))›
unfolding *set-all-to-list-def*
apply (*refine-vcg*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *take-Suc-conv-app-nth list-update-append*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

definition *get-literal-and-remove-of-analyse-wl*
:: ‹'*v clause-l* ⇒ (*nat* × *nat* × *nat* × *nat*) *list* ⇒ '*v literal* × (*nat* × *nat* × *nat* × *nat*) *list*› **where**
‹*get-literal-and-remove-of-analyse-wl C analyse* =
(*let* (*i*, *k*, *j*, *ln*) = *last analyse in*
(*C* ! *j*, *analyse*[*length analyse* − *1* := (*i*, *k*, *j* + *1*, *ln*)]))›

definition *mark-failed-lits-wl*
where
‹*mark-failed-lits-wl NU analyse cach* = *SPEC*(λ*cach'*.
(∀ *L*. *cach' L* = *SEEN-REMOVABLE* ⟶ *cach L* = *SEEN-REMOVABLE*))›

definition *lit-redundant-rec-wl-ref* **where**
‹*lit-redundant-rec-wl-ref NU analyse* ⟷
(∀ (*i*, *k*, *j*, *ln*) ∈ *set analyse*. *j* ≤ *ln* ∧ *i* ∈# *dom-m NU* ∧ *i* > *0* ∧
ln ≤ *length* (*NU* ∝ *i*) ∧ *k* < *length* (*NU* ∝ *i*) ∧
distinct (*NU* ∝ *i*) ∧
¬*tautology* (*mset* (*NU* ∝ *i*))) ∧
(∀ (*i*, *k*, *j*, *ln*) ∈ *set* (*butlast analyse*). *j* > *0*)›

definition *lit-redundant-rec-wl-inv* **where**
‹*lit-redundant-rec-wl-inv M NU D* = (λ(*cach*, *analyse*, *b*). *lit-redundant-rec-wl-ref NU analyse*)›

definition *lit-redundant-reason-stack*
:: ‹'*v literal* ⇒ '*v clauses-l* ⇒ *nat* ⇒ (*nat* × *nat* × *nat* × *nat*)› **where**
‹*lit-redundant-reason-stack L NU C'* =
(*if length* (*NU* ∝ *C'*) > *2 then* (*C'*, *0*, *1*, *length* (*NU* ∝ *C'*))
else if NU ∝ *C'* ! *0* = *L then* (*C'*, *0*, *1*, *length* (*NU* ∝ *C'*))
else (*C'*, *1*, *0*, *1*))›

definition *lit-redundant-rec-wl* :: ‹('*v*, *nat*) *ann-lits* ⇒ '*v clauses-l* ⇒ '*v clause* ⇒
- ⇒ - ⇒ - ⇒
(- × - × *bool*) *nres*›

where
‹*lit-redundant-rec-wl M NU D cach analysis - =*
WHILE$_T^{\text{lit-redundant-rec-wl-inv } M\ NU\ D}$
(λ(*cach*, *analyse*, *b*). *analyse* ≠ [])
(λ(*cach*, *analyse*, *b*). *do* {
ASSERT(*analyse* ≠ []);
ASSERT(*length analyse* ≤ *length M*);
let (*C*, *k*, *i*, *ln*) = *last analyse*;
ASSERT(*C* ∈# *dom-m NU*);
ASSERT(*length* (*NU* ∝ *C*) > *k*);
ASSERT(*NU* ∝ *C*!*k* ∈ *lits-of-l M*);
let C = *NU* ∝ *C*;
if i ≥ *ln*
then
RETURN(*cach* (*atm-of* (*C* ! *k*) := *SEEN-REMOVABLE*), *butlast analyse*, *True*)
else do {
let (*L*, *analyse*) = *get-literal-and-remove-of-analyse-wl C analyse*;
ASSERT(*fst*(*snd*(*snd* (*last analyse*))) ≠ *0*);
ASSERT(−*L* ∈ *lits-of-l M*);
b ← *RES* (*UNIV*);
if (*get-level M L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ∨ *L* ∈# *D*)
then RETURN (*cach*, *analyse*, *False*)
else if b ∨ *cach* (*atm-of L*) = *SEEN-FAILED*
then do {
cach ← *mark-failed-lits-wl NU analyse cach*;
RETURN (*cach*, [], *False*)
}
else do {
ASSERT(*cach* (*atm-of L*) = *SEEN-UNKNOWN*);
C′ ← *get-propagation-reason M* (−*L*);
case C′ of
Some C′ ⇒ *do* {
ASSERT(*C′* ∈# *dom-m NU*);
ASSERT(*length* (*NU* ∝ *C′*) ≥ *2*);
ASSERT (*distinct* (*NU* ∝ *C′*) ∧ ¬*tautology* (*mset* (*NU* ∝ *C′*)));
ASSERT(*C′* > *0*);
RETURN (*cach*, *analyse* @ [*lit-redundant-reason-stack* (−*L*) *NU C′*], *False*)
}
| *None* ⇒ *do* {
cach ← *mark-failed-lits-wl NU analyse cach*;
RETURN (*cach*, [], *False*)
}
}
}
})
(*cach*, *analysis*, *False*)›

fun *convert-analysis-l* **where**
‹*convert-analysis-l NU* (*i*, *k*, *j*, *le*) = (−*NU* ∝ *i* ! *k*, *mset* (*Misc.slice j le* (*NU* ∝ *i*)))›

definition *convert-analysis-list* **where**
‹*convert-analysis-list NU analyse* = *map* (*convert-analysis-l NU*) (*rev analyse*)›

lemma *convert-analysis-list-empty*[*simp*]:
‹*convert-analysis-list NU* [] = []›
‹*convert-analysis-list NU a* = [] ⟷ *a* = []›

by (*auto simp*: *convert-analysis-list-def*)

lemma *trail-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
LaC: ‹*Propagated L C* ∈ *set* (*get-trail-l S*)› **and**
C0: ‹*C* > *0*›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
proof −
have *conv*:
‹(*get-trail-l S*, *get-trail T*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-kept-unit-clauses-l S*)›
using *ST* **unfolding** *twl-st-l-def* **by** *auto*

have ‹$cdcl_W$-*restart-mset*.$cdcl_W$-*conflicting* ($state_W$-*of T*)› **and**
lev-inv: ‹$cdcl_W$-*restart-mset*.$cdcl_W$-*M-level-inv* ($state_W$-*of T*)›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv-def*
pcdcl-all-struct-invs-def $state_W$-*of-def*
by *fast*+

have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *ST lev-inv* **unfolding** $cdcl_W$-*restart-mset*.$cdcl_W$-*M-level-inv-def*
by (*auto simp*: *twl-st-l twl-st*)
have
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
using *list-invs C0 LaC* **by** (*auto simp*: *twl-list-invs-def all-conj-distrib*)
have ‹*twl-st-inv T*›
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
then show *le2*: ‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
using *C ST multi-member-split*[*OF C*] **unfolding** *twl-struct-invs-def*
by (*cases S*; *cases T*)
(*auto simp*: *twl-st-inv.simps twl-st-l-def*
image-Un[*symmetric*])
qed

lemma *clauses-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
proof −
have ‹*twl-st-inv T*›
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
then show *le2*: ‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
using *C ST multi-member-split*[*OF C*] **unfolding** *twl-struct-invs-def*
by (*cases S*; *cases T*)
(*auto simp*: *twl-st-inv.simps twl-st-l-def*
image-Un[*symmetric*])
qed

lemma *lit-redundant-rec-wl*:
fixes S :: ‹*nat twl-st-wl*› **and** S' :: ‹*nat twl-st-l*› **and** S'' :: ‹*nat twl-st*› **and** *NU M analyse*
defines
[*simp*]: ‹$S''' \equiv$ *state*$_W$*-of* S''›
defines
‹$M \equiv$ *get-trail-wl* S› **and**
M': ‹$M' \equiv$ *trail* S'''› **and**
NU: ‹*NU* ≡ *get-clauses-wl* S› **and**
NU': ‹$NU' \equiv$ *mset* ‘# *ran-mf NU*› **and**
‹*analyse*′ ≡ *convert-analysis-list NU analyse*›
assumes
S-S′: ‹(S, S') ∈ *state-wl-l None*› **and**
S′*-S*″: ‹(S', S'') ∈ *twl-st-l None*› **and**
bounds-init: ‹*lit-redundant-rec-wl-ref NU analyse*› **and**
struct-invs: ‹*twl-struct-invs* S''› **and**
add-inv: ‹*twl-list-invs* S'›
shows
‹*lit-redundant-rec-wl M NU D cach analyse lbd* ≤ ⇓
(*Id* ×$_r$ {(*analyse*, *analyse*′). *analyse*′ = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse} ×$_r$ *bool-rel*)
(*lit-redundant-rec M*′ *NU*′ *D cach analyse*′)›
(**is** ‹- ≤ ⇓ (- ×$_r$ *?A* ×$_r$ -) -› **is** ‹- ≤ ⇓ *?R* -›)
proof −
obtain *D*′ *NE UE Q W NEk UEk NS US N0 U0* **where**
S: ‹$S = (M, NU, D', NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)$›
using *M-def NU* **by** (*cases S*) *auto*
have *M*′*-def*: ‹(M, M') ∈ *convert-lits-l NU* (*NEk* + *UEk*)›
using *NU S-S*′ *S*′*-S*″ **unfolding** M' **by** (*auto simp*: *S state-wl-l-def twl-st-l-def*)
then have [*simp*]: ‹*lits-of-l* M' = *lits-of-l* M›
by *auto*
have [*simp*]: ‹*fst* (*convert-analysis-l NU x*) = −*NU* ∝ (*fst x*) ! (*fst* (*snd x*))› **for** x
by (*cases x*) *auto*
have [*simp*]: ‹*snd* (*convert-analysis-l NU x*) =
mset (*Misc.slice* (*fst* (*snd* (*snd x*))) (*snd* (*snd* (*snd x*))) (*NU* ∝ *fst x*))› **for** x
by (*cases x*) *auto*

have
no-smaller-propa: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* S'''› **and**
struct-invs: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* S'''›
using *struct-invs* **unfolding** *twl-struct-invs-def* S'''*-def*[*symmetric*]
pcdcl-all-struct-invs-def state$_W$*-of-def*[*symmetric*]
by *fast*+
have *annots*: ‹*set* (*get-all-mark-of-propagated* (*trail* S''')) ⊆
set-mset (*cdcl*$_W$*-restart-mset.clauses* S''')›
using *struct-invs*
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
by *fast*
have ‹*no-dup* (*get-trail-wl* S)›
using *struct-invs S-S*′ *S*′*-S*″ **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by (*auto simp*: *twl-st-wl twl-st-l twl-st*)
then have *n-d*: ‹*no-dup* M›
by (*auto simp*: S)
then have *n-d*′: ‹*no-dup* M'›
using *M*′*-def* **by** (*auto simp*: S)

let *?B* = ‹{(*analyse*, *analyse'*). *analyse'* = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse ∧ *fst* (*snd* (*snd* (*last analyse*))) > *0*}›
have *get-literal-and-remove-of-analyse-wl*:
‹*RETURN* (*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c*)
≤ ⇓ (*Id* ×$_r$ *?B*)
(*get-literal-and-remove-of-analyse x1a*)›
if
xx': ‹(*x*, *x'*) ∈ *?R*› **and**
‹*case x of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*case x' of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*lit-redundant-rec-wl-inv M NU D x*› **and**
s: ‹*x2* = (*x1a*, *x2a*)›
‹*x'* = (*x1*, *x2*)›
‹*x2d* = (*x1f*, *x2e*)›
‹*x2c* = (*x1e*, *x2d*)›
‹(*fst* (*last x1c*), *fst* (*snd* (*last x1c*)), *fst* (*snd* (*snd* (*last x1c*))),
snd (*snd* (*snd* (*last x1c*)))) =
(*x1d*, *x2c*)›
‹*x2b* = (*x1c*, *x2f*)›
‹*x* = (*x1b*, *x2b*)› **and**
‹*x1a* ≠ []› **and**
‹− *fst* (*hd x1a*) ∈ *lits-of-l M'*› **and**
x1c: ‹*x1c* ≠ []› **and**
‹*x1d* ∈# *dom-m NU*› **and**
‹*x1e* < *length* (*NU* ∝ *x1d*)› **and**
‹*NU* ∝ *x1d* ! *x1e* ∈ *lits-of-l M*› **and**
length: ‹¬ *x2e* ≤ *x1f*› **and**
‹*snd* (*hd x1a*) ≠ {#}›
for *x x' x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f*
proof −
have *x1d*: ‹*x1d* = *fst* (*last x1c*)›
using *s* **by** *auto*
have ‹*last x1c* = (*a*, *b*, *c*, *d*) ⟹ *d* ≤ *length* (*NU* ∝ *a*)›
‹*last x1c* = (*a*, *b*, *c*, *d*) ⟹ *c* ≤ *d*› **for** *aa ba list a b c d*
using *xx' x1c length* **unfolding** *s convert-analysis-list-def*
lit-redundant-rec-wl-ref-def
by (*cases x1c rule*: *rev-cases*; *auto*; *fail*)+
then show *?thesis*
supply *convert-analysis-list-def*[*simp*] *hd-rev*[*simp*] *last-map*[*simp*] *rev-map*[*symmetric*, *simp*]
using *x1c xx' length s*
using *Cons-nth-drop-Suc*[*of* ‹*snd* (*snd* (*snd* (*last x1c*)))› ‹*NU* ∝ *fst* (*last x1c*)›, *symmetric*]
unfolding *lit-redundant-rec-wl-ref-def x1d*
by (*cases x1c*; *cases* ‹*last x1c*›)
(*auto simp*: *get-literal-and-remove-of-analyse-wl-def nth-in-sliceI mset-tl*
get-literal-and-remove-of-analyse-def convert-analysis-list-def slice-Suc
slice-head
intro!: *RETURN-SPEC-refine elim*!: *neq-Nil-revE split*: *if-splits*)
qed

have *get-propagation-reason*: ‹*get-propagation-reason M* (− *x1h*)
≤ ⇓ (⟨{(*C'*, *C*). *C* = *mset* (*NU* ∝ *C'*) ∧ *C'* ≠ *0* ∧
Propagated (− *x1g*) (*mset* (*NU*∝*C'*)) ∈ *set M'*
∧ *Propagated* (− *x1g*) *C'* ∈ *set M* ∧ *C'* ∈# *dom-m NU* ∧
length (*NU* ∝ *C'*) ≥ *2*}⟩
option-rel)
(*get-propagation-reason M'* (− *x1g*))›

```
    (is ‹- ≤ ⇓ (⟨?get-propagation-reason⟩option-rel) -›)
  if
    ‹(x, x') ∈ ?R› and
    ‹case x of (cach, analyse, b) ⇒ analyse ≠ []› and
    ‹case x' of (cach, analyse, b) ⇒ analyse ≠ []› and
    ‹lit-redundant-rec-wl-inv M NU D x› and
    st:
‹x2 = (x1a, x2a)›
‹x' = (x1, x2)›
‹x2d = (x1f, x2e)›
‹x2c = (x1e, x2d)›
‹(fst (last x1c), fst (snd (last x1c)), fst (snd (snd (last x1c))),
  snd (snd (snd (last x1c)))) =
 (x1d, x2c)›
‹x2b = (x1c, x2f)›
‹x = (x1b, x2b)›
      ‹x'a = (x1g, x2g)› and
    ‹x1a ≠ []› and
    ‹- fst (hd x1a) ∈ lits-of-l M'› and
    ‹x1c ≠ []› and
    x1d: ‹x1d ∈# dom-m NU› and
    ‹x1e < length (NU ∝ x1d)› and
    ‹NU ∝ x1d ! x1e ∈ lits-of-l M› and
    ‹¬ x2e ≤ x1f› and
    ‹snd (hd x1a) ≠ {#}› and
    H: ‹(get-literal-and-remove-of-analyse-wl (NU ∝ x1d) x1c, x'a)
        ∈ Id ×f ?B›
      ‹get-literal-and-remove-of-analyse-wl (NU ∝ x1d) x1c = (x1h, x2h)› and
    ‹- x1g ∈ lits-of-l M'› and
    ‹- x1h ∈ lits-of-l M› and
    ‹(b, ba) ∈ bool-rel› and
    ‹b ∈ UNIV› and
    ‹ba ∈ UNIV› and
    ‹¬ (get-level M x1h = 0 ∨ x1b (atm-of x1h) = SEEN-REMOVABLE ∨ x1h ∈# D)› and
    cond: ‹¬ (get-level M' x1g = 0 ∨ x1 (atm-of x1g) = SEEN-REMOVABLE ∨ x1g ∈# D)› and
    ‹¬ (b ∨ x1b (atm-of x1h) = SEEN-FAILED)› and
    ‹¬ (ba ∨ x1 (atm-of x1g) = SEEN-FAILED)›
 for x x' x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f x'a x1g x2g x1h
x2h b ba
 proof −
  have [simp]: ‹x1h = x1g›
    using st H by auto
  have le2: ‹length (NU ∝ x1d) ≥ 2›
    using clauses-length-ge2[OF S'-S'' add-inv assms(10), of x1d] x1d st S-S'
    by (auto simp: S)
  have
    ‹Propagated (− x1g) (mset (NU ∝ a)) ∈ set M'› (is ?propa) and
    ‹a ≠ 0› (is ?a) and
    ‹a ∈# dom-m NU› (is ?L) and
    ‹length (NU ∝ a) ≥ 2› (is ?len)
    if x1e-M: ‹Propagated (− x1g) a ∈ set M›
    for a
  proof −
    have [simp]: ‹a ≠ 0›
    proof
      assume [simp]: ‹a = 0›
```

```
      obtain E' where
        x1d-M': ‹Propagated (− x1g) E' ∈ set M'› and
        ‹E' ∈# NEk + UEk›
        using x1e-M M'-def by (auto dest: split-list simp: convert-lits-l-def p2rel-def
            convert-lit.simps
            elim!: list-rel-in-find-correspondanceE split: if-splits)
      moreover have ‹unit-clss S'' = NE + NEk + UE + UEk›
        using S-S' S'-S'' x1d-M' by (auto simp: S)
      moreover have ‹Propagated (− x1g) E' ∈ set (get-trail S'')›
        using S-S' S'-S'' x1d-M' by (auto simp: S state-wl-l-def twl-st-l-def M')
      moreover have ‹0 < count-decided (get-trail S'')›
        using cond S-S' S'-S'' count-decided-ge-get-level[of M ‹x1g›]
        by (auto simp: S M' twl-st)
      ultimately show False
        using clauses-in-unit-clss-have-level0(1)[of S'' E' ‹− x1g›] cond ‹twl-struct-invs S''›
        S-S' S'-S''  M'-def
        by (auto simp: S)
    qed
    show ?propa and ?a
      using that M'-def by (auto simp: convert-lits-l-def p2rel-def convert-lit.simps
          elim!: list-rel-in-find-correspondanceE split: if-splits)
    then show ?L
      using that add-inv S-S' S'-S'' S unfolding twl-list-invs-def
      by (auto 5 5 simp: state-wl-l-def twl-st-l-def)
    show ?len
      using trail-length-ge2[OF S'-S'' add-inv assms(10), of ‹− x1g› a] that S-S'
by (force simp: S)
  qed
  then show ?thesis
    apply (auto simp: get-propagation-reason-def refine-rel-defs intro!: RES-refine)
    apply (case-tac s)
    by auto
qed
have resolve: ‹((x1b, x2h @ [lit-redundant-reason-stack (− x1h) NU xb], False), x1,
(x1g, remove1-mset (− x1g) x'c) # x2g, False)
∈ Id ×f
 ({(analyse, analyse'). analyse' = convert-analysis-list NU analyse ∧
  lit-redundant-rec-wl-ref NU analyse} ×f
  bool-rel)›
  if
    xx': ‹(x, x') ∈ Id ×r ?A ×r bool-rel› and
    ‹case x of (cach, analyse, b) ⇒ analyse ≠ []› and
    ‹case x' of (cach, analyse, b) ⇒ analyse ≠ []› and
    ‹lit-redundant-rec-wl-inv M NU D x› and
    s:
      ‹x2 = (x1a, x2a)›
      ‹x' = (x1, x2)›
      ‹x2d = (x1f, x2e)›
      ‹x2c = (x1e, x2d)›
      ‹(fst (last x1c), fst (snd (last x1c)), fst (snd (snd (last x1c))),
        snd (snd (snd (last x1c)))) =
       (x1d, x2c)›
      ‹x2b = (x1c, x2f)›
      ‹x = (x1b, x2b)›
‹x'a = (x1g, x2g)› and
```

[*simp*]: ‹*x1a* ≠ []› **and**
‹− *fst* (*hd x1a*) ∈ *lits-of-l M′*› **and**
[*simp*]: ‹*x1c* ≠ []› **and**
‹*x1d* ∈# *dom-m NU*› **and**
‹*x1e* < *length* (*NU* ∝ *x1d*)› **and**
‹*NU* ∝ *x1d* ! *x1e* ∈ *lits-of-l M*› **and**
‹¬ *x2e* ≤ *x1f*› **and**
‹*snd* (*hd x1a*) ≠ {#}› **and**
get-literal-and-remove-of-analyse-wl:
‹(*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c*, *x′a*)
∈ *Id* ×*f*
{(*analyse*, *analyse′*).
analyse′ = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse ∧
0 < *fst* (*snd* (*snd* (*last analyse*)))}› **and**
get-lit: ‹*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c* = (*x1h*, *x2h*)› **and**
‹− *x1g* ∈ *lits-of-l M′*› **and**
‹*fst* (*snd* (*snd* (*last x2h*))) ≠ 0› **and**
‹− *x1h* ∈ *lits-of-l M*› **and**
bba: ‹(*b*, *ba*) ∈ *bool-rel*› **and**
‹¬ (*get-level M x1h* = 0 ∨ *x1b* (*atm-of x1h*) = *SEEN-REMOVABLE* ∨ *x1h* ∈# *D*)› **and**
‹¬ (*get-level M′ x1g* = 0 ∨ *x1* (*atm-of x1g*) = *SEEN-REMOVABLE* ∨ *x1g* ∈# *D*)› **and**
‹¬ (*b* ∨ *x1b* (*atm-of x1h*) = *SEEN-FAILED*)› **and**
‹¬ (*ba* ∨ *x1* (*atm-of x1g*) = *SEEN-FAILED*)› **and**
xb-x′c: ‹(*xa*, *x′b*)
∈ ⟨*?get-propagation-reason x1g*⟩*option-rel*› **and**
xa: ‹*xa* = *Some xb*› ‹*x′b* = *Some x′c*› **and**
‹(*xb*, *x′c*)
∈ (*?get-propagation-reason x1g*)› **and**
dist-tauto: ‹*distinct-mset x′c* ∧ ¬ *tautology x′c*› **and**
‹*xb* ∈# *dom-m NU*› **and**
‹2 ≤ *length* (*NU* ∝ *xb*)›
for *x x′ x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f x′a x1g x2g x1h*
x2h b ba xa x′b xb x′c
proof −
have [*simp*]: ‹*mset* (*tl C*) = *remove1-mset* (*C*!0) (*mset C*)› **for** *C*
by (*cases C*) *auto*
have [*simp*]:
‹*x2* = (*x1a*, *x2a*)›
‹*x′* = (*x1*, *x1a*, *x2a*)›
‹*x2d* = (*x1f*, *x2e*)›
‹*x2c* = (*x1e*, *x1f*, *x2e*)›
‹*last x1c* = (*x1d*, *x1e*, *x1f*, *x2e*)›
‹*x2b* = (*x1c*, *x2f*)›
‹*x* = (*x1b*, *x1c*, *x2f*)›
‹*xa* = *Some xb*›
‹*x′b* = *Some x′c*›
‹*x′c* = *mset* (*NU* ∝ *xb*)›
using *s get-literal-and-remove-of-analyse-wl xa xb-x′c*
unfolding *get-lit convert-analysis-list-def*
by *auto*
then have *x1d0*: ‹*length* (*NU* ∝ *xb*) > 2 ⟹ *x1g* = −*NU* ∝ *xb* ! 0› ‹*NU* ∝ *xb* ≠ []› **and**
x1d: ‹−*x1g* ∈ *set* (*watched-l* (*NU* ∝ *xb*))›
using *add-inv xb-x′c S-S′ S′-S″ S* **unfolding** *twl-list-invs-def*
by (*auto 5 5 simp*: *state-wl-l-def twl-st-l-def*)

```
  have le2: ‹length (NU ∝ xb) ≥ 2›
    using clauses-length-ge2[OF S'-S'' add-inv assms(10)] xb-x'c S-S'
    by (auto simp: S)
  have 0: ‹case lit-redundant-reason-stack (−x1g) NU xb of (i, k, j, ln) ⇒
        j ≤ ln ∧ i ∈# dom-m NU ∧ 0 ≤ j ∧ 0 < i ∧ ln ≤ length (NU ∝ i) ∧
     k < length (NU ∝ i) ∧ distinct (NU ∝ i) ∧ ¬ tautology (mset (NU ∝ i))›
    for i j ln k
    using s xx' get-literal-and-remove-of-analyse-wl xb-x'c x1d le2 dist-tauto
    unfolding get-lit convert-analysis-list-def lit-redundant-rec-wl-ref-def
    lit-redundant-reason-stack-def
    by (auto split: if-splits)

  have ‹(x1g, remove1-mset (− x1g) (mset (NU ∝ xb))) =
    convert-analysis-l NU (lit-redundant-reason-stack (− x1g) NU xb)›
    using s xx' get-literal-and-remove-of-analyse-wl xb-x'c x1d le2
    unfolding get-lit convert-analysis-list-def lit-redundant-rec-wl-ref-def
      lit-redundant-reason-stack-def
    by (auto split: simp: Misc.slice-def drop-Suc simp: x1d0(1)
      dest!: list-decomp-2)
  then show ?thesis
    using s xx' get-literal-and-remove-of-analyse-wl xb-x'c x1d 0
    unfolding get-lit convert-analysis-list-def lit-redundant-rec-wl-ref-def
    by (cases x2h rule: rev-cases)
      (auto simp: drop-Suc uminus-lit-swap butlast-append
      dest: list-decomp-2)
qed
have mark-failed-lits-wl: ‹mark-failed-lits-wl NU x2e x1b ≤ ⇓ Id (mark-failed-lits NU' x2d x1)›
  if
    ‹(x, x') ∈ ?R› and
    ‹x' = (x1, x2)› and
    ‹x = (x1b, x2b)›
  for x x' x2e x1b x1 x2 x2b x2d
  using that unfolding mark-failed-lits-wl-def mark-failed-lits-def by auto

have ana: ‹last analyse = (fst (last analyse), fst (snd (last analyse)),
  fst (snd (snd (last analyse))), snd (snd (snd (last analyse))))› for analyse
    by (cases ‹last analyse›) auto

show ?thesis
  supply convert-analysis-list-def[simp] hd-rev[simp] last-map[simp] rev-map[symmetric, simp]
  unfolding lit-redundant-rec-wl-def lit-redundant-rec-def
  apply (rewrite at ‹let - = - ∝ - in -› Let-def)
  apply (rewrite in ‹let - = - in -› ana)
  apply (rewrite at ‹let - = (-, -, -) in -› Let-def)
  apply refine-rcg
  subgoal using bounds-init unfolding analyse'-def by auto
  subgoal for x x'
    by (cases x, cases x')
      (auto simp: lit-redundant-rec-wl-inv-def lit-redundant-rec-wl-ref-def)
  subgoal by auto
  subgoal by auto
  subgoal using M'-def by (auto dest: convert-lits-l-imp-same-length)
  subgoal by (auto simp: lit-redundant-rec-wl-inv-def lit-redundant-rec-wl-ref-def
    elim!: neq-Nil-revE)
  subgoal by (auto simp: lit-redundant-rec-wl-inv-def lit-redundant-rec-wl-ref-def
    elim!: neq-Nil-revE)
```

subgoal by (*auto simp*: *map-butlast rev-butlast-is-tl-rev lit-redundant-rec-wl-ref-def*
dest: *in-set-butlastD*)
subgoal by (*auto simp*: *map-butlast rev-butlast-is-tl-rev lit-redundant-rec-wl-ref-def*
Misc.slice-def
dest: *in-set-butlastD*
elim!: *neq-Nil-revE*)
subgoal by (*auto simp*: *map-butlast rev-butlast-is-tl-rev lit-redundant-rec-wl-ref-def*
Misc.slice-def
dest: *in-set-butlastD*
elim!: *neq-Nil-revE*)
apply (*rule get-literal-and-remove-of-analyse-wl*; *assumption*)
subgoal by *auto*
subgoal by *auto*
subgoal using *M'-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
apply (*rule mark-failed-lits-wl*; *assumption*)
subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
subgoal by *auto*
apply (*rule get-propagation-reason*; *assumption*)
apply *assumption*
apply (*rule mark-failed-lits-wl*; *assumption*)
subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
subgoal for *x x' x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f x'a x1g x2g x1h*
x2h b ba xa x'b xb x'c
by (*rule resolve*)
done
qed

definition *literal-redundant-wl* **where**
‹*literal-redundant-wl M NU D cach L lbd* = *do* {
ASSERT(−*L* ∈ *lits-of-l M*);
if get-level M L = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE*
then RETURN (*cach*, [], *True*)
else if cach (*atm-of L*) = *SEEN-FAILED*
then RETURN (*cach*, [], *False*)
else do {
C ← *get-propagation-reason M* (−*L*);
case C of
Some C ⇒ *do*{
ASSERT(*C* ∈# *dom-m NU*);
ASSERT(*length* (*NU* ∝ *C*) ≥ *2*);
ASSERT(*distinct* (*NU* ∝ *C*) ∧ ¬*tautology* (*mset* (*NU* ∝ *C*)));
lit-redundant-rec-wl M NU D cach [*lit-redundant-reason-stack* (−*L*) *NU C*] *lbd*
}
| *None* ⇒ *do* {
RETURN (*cach*, [], *False*)
}
}
}›

lemma *literal-redundant-wl-literal-redundant*:
fixes S :: ‹*nat twl-st-wl*› **and** S' :: ‹*nat twl-st-l*› **and** S'' :: ‹*nat twl-st*› **and** *NU M*
defines
[*simp*]: ‹$S''' \equiv$ *state*$_W$*-of* S''›
defines
‹$M \equiv$ *get-trail-wl* S› **and**
M': ‹$M' \equiv$ *trail* S'''› **and**
NU: ‹*NU* ≡ *get-clauses-wl* S› **and**
NU': ‹$NU' \equiv$ *mset* ‘# *ran-mf NU*›
assumes
S-S': ‹$(S, S') \in$ *state-wl-l None*› **and**
S'-S'': ‹$(S', S'') \in$ *twl-st-l None*› **and**
‹$M \equiv$ *get-trail-wl* S› **and**
M': ‹$M' \equiv$ *trail* S'''› **and**
NU: ‹*NU* ≡ *get-clauses-wl* S› **and**
NU': ‹$NU' \equiv$ *mset* ‘# *ran-mf NU*›
assumes
struct-invs: ‹*twl-struct-invs* S''› **and**
add-inv: ‹*twl-list-invs* S'› **and**
L-D: ‹$L \in\#$ D› **and**
M-D: ‹$M \models as$ *CNot* D›
shows
‹*literal-redundant-wl M NU D cach L lbd* ≤ ⇓
(*Id* ×$_r$ {(*analyse*, *analyse*′). *analyse*′ = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse} ×$_r$ *bool-rel*)
(*literal-redundant* M' NU' D *cach L*)›
(**is** ‹- ≤ ⇓ (- ×$_r$ *?A* ×$_r$ -) -› **is** ‹- ≤ ⇓ *?R* -›)
proof −
obtain D' *NE UE NEk UEk Q W NS US N0 U0* **where**
S: ‹$S = (M, NU, D', NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)$›
using *M-def NU* **by** (*cases S*) *auto*
have M'*-def*: ‹$(M, M') \in$ *convert-lits-l NU* (*NEk*+*UEk*)›
using *NU S-S' S'-S'' S M'* **by** (*auto simp*: *twl-st-l-def state-wl-l-def*)
have [*simp*]: ‹*lits-of-l* $M' =$ *lits-of-l* M›
using M'*-def* **by** *auto*
have
no-smaller-propa: ‹*cdcl*$_W$*-restart-mset.no-smaller-propa* S'''› **and**
struct-invs′: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* S'''›
using *struct-invs* **unfolding** *twl-struct-invs-def* S'''*-def*[*symmetric*]
pcdcl-all-struct-invs-def state$_W$*-of-def*[*symmetric*]
by *fast*+
have *annots*: ‹*set* (*get-all-mark-of-propagated* (*trail* S''')) ⊆
set-mset (*cdcl*$_W$*-restart-mset.clauses* S''')›
using *struct-invs*′
unfolding *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clause-alt-def*
by *fast*
have *n-d*: ‹*no-dup* (*get-trail-wl* S)›
using *struct-invs*′ *S-S'* *S'-S''* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by (*auto simp*: *twl-st-wl twl-st-l twl-st*)
then have *n-d*: ‹*no-dup* M›
by (*auto simp*: S)
then have *n-d*′: ‹*no-dup* M'›
using M'*-def* **by** (*auto simp*: S)

have *uL-M*: ‹$-L \in$ *lits-of-l* M›
using *L-D M-D* **by** (*auto dest*!: *multi-member-split*)
have *H*: ‹*lit-redundant-rec-wl M NU D cach analyse lbd*
$\leq \Downarrow$ *?R* (*lit-redundant-rec* M' *NU'* *D cach analyse'*)›
if ‹*analyse'* = *convert-analysis-list NU analyse*› **and**
‹*lit-redundant-rec-wl-ref NU analyse*›
for *analyse analyse'*
using *lit-redundant-rec-wl*[*of S S' S'' analyse D cach lbd*, *unfolded S'''-def*[*symmetric*],
unfolded
M-def[*symmetric*] M'[*symmetric*] *NU*[*symmetric*] *NU'*[*symmetric*],
OF S-S' S'-S'' - struct-invs add-inv]
that **by** (*auto simp*:)
have *get-propagation-reason*: ‹*get-propagation-reason* M $(-L)$
$\leq \Downarrow$ ($\langle\{(C', C).\ C = mset\ (NU \propto C') \wedge C' \neq 0 \wedge Propagated\ (-L)\ (mset\ (NU \propto C')) \in set\ M'$
$\wedge\ Propagated\ (-L)\ C' \in set\ M \wedge length\ (NU \propto C') \geq 2\}\rangle$
option-rel)
(*get-propagation-reason* M' $(-L)$)›
(**is** ‹- $\leq \Downarrow$ ($\langle$*?get-propagation-reason*$\rangle$*option-rel*) -› **is** *?G1*) **and**
propagated-L:
‹*Propagated* $(-L)$ $a \in$ *set* $M \Longrightarrow a \neq 0 \wedge$ *Propagated* $(-L)$ (*mset* $(NU \propto a)) \in$ *set* M'›
(**is** ‹*?H2* $\Longrightarrow$ *?G2*›)
if
lev0-rem: ‹$\neg$ (*get-level* M' $L = 0 \vee$ *cach* (*atm-of* L) = *SEEN-REMOVABLE*)› **and**
ux1e-M: ‹$-L \in$ *lits-of-l* M›
for a
proof $-$
have ‹*Propagated* $(-L)$ (*mset* $(NU \propto a)) \in$ *set* M'› (**is** *?propa*) **and**
‹$a \neq 0$› (**is** *?a*) **and**
‹*length* $(NU \propto a) \geq 2$› (**is** *?len*)
if *L-M*: ‹*Propagated* $(-L)$ $a \in$ *set* M›
for a
proof $-$
have [*simp*]: ‹$a \neq 0$›
proof
assume [*simp*]: ‹$a = 0$›
obtain E' **where**
x1d-M': ‹*Propagated* $(-L)$ $E' \in$ *set* M'› **and**
‹$E' \in\#$ *NEk* + *UEk*›
using *L-M M'-def* **by** (*auto dest*: *split-list simp*: *convert-lits-l-def p2rel-def*
convert-lit.simps
elim!: *list-rel-in-find-correspondanceE split*: *if-splits*)
moreover have ‹*unit-clss* $S'' =$ *NE* + *NEk* + *UE* + *UEk*›
using *S-S' S'-S'' x1d-M'* **by** (*auto simp*: *S*)
moreover have ‹*Propagated* $(-L)$ $E' \in$ *set* (*get-trail* S'')›
using *S-S' S'-S'' x1d-M'* **by** (*auto simp*: *S state-wl-l-def twl-st-l-def M'*)
moreover have ‹$0<$ *count-decided* (*get-trail* S'')›
using *lev0-rem S-S' S'-S'' count-decided-ge-get-level*[*of M L*]
by (*auto simp*: *S M' twl-st*)
ultimately show *False*
using *clauses-in-unit-clss-have-level0*(*1*)[*of S'' E'* ‹$-L$›] *lev0-rem* ‹*twl-struct-invs* S''›
S-S' S'-S'' *M'-def*
by (*auto simp*: *S*)
qed

show *?propa* **and** *?a*
using *that M'-def* **by** (*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*

```
        elim!: list-rel-in-find-correspondanceE split: if-splits)
 show ?len
  using trail-length-ge2[OF S'-S'' add-inv struct-invs, of ‹- L› a] that S-S'
  by (force simp: S)
    qed note H = this
   show ‹?H2 ⟹ ?G2›
    using H by auto
   show ?G1
    using H
    apply (auto simp: get-propagation-reason-def refine-rel-defs
       get-propagation-reason-def intro!: RES-refine)
    apply (case-tac s)
    by auto
  qed
 have S''': ‹S''' = (get-trail S'', get-all-init-clss S'', get-all-learned-clss S'',
    get-conflict S'')›
  by (cases S'') (auto simp: S'''-def)
 have [simp]: ‹mset (tl C) = remove1-mset (C!0) (mset C)› for C
  by (cases C) auto
 have S''-M': ‹(get-trail S'') = M'›
  using M' S''' by auto

 have [simp]: ‹length (NU ∝ C) > 2 ⟹ NU ∝ C ! 0 = -L› and
  L-watched: ‹-L ∈ set (watched-l (NU ∝ C))› and
  L-dist: ‹distinct (NU ∝ C)› and
  L-tauto: ‹¬tautology (mset (NU ∝ C))›
 if
  in-trail: ‹Propagated (- L) C ∈ set M› and
  lev: ‹¬ (get-level M' L = 0 ∨ cach (atm-of L) = SEEN-REMOVABLE)›
  for C
  using add-inv that propagated-L[OF lev - in-trail] uL-M S-S' S'-S''
  Propagated-in-trail-entailed[of ‹get-trail S''› ‹get-all-init-clss S''› ‹get-all-learned-clss S''›
    ‹get-conflict S''› ‹-L› ‹mset (NU ∝ C)›] struct-invs' unfolding S'''[symmetric]
  by (auto simp: S twl-list-invs-def S''-M'; fail)+

 have [dest]: ‹C ≠ {#}› if ‹Propagated (- L) C ∈ set M'› for C
 proof -
  have ‹a @ Propagated L mark # b = trail S''' ⟹ b ⊨as CNot (remove1-mset L mark) ∧ L ∈#
mark›
   for L mark a b
   using struct-invs' unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
    cdcl_W-restart-mset.cdcl_W-conflicting-def
   by fast
  then show ?thesis
   using that S-S' S'-S'' M'-def M'
   by (fastforce simp: S state-wl-l-def
      twl-st-l-def convert-lits-l-def convert-lit.simps
      list-rel-append2 list-rel-append1
      elim!: list-relE3 list-relE4
      elim: list-rel-in-find-correspondanceE split: if-splits
      dest!: split-list p2relD)
 qed
 have le2: ‹Propagated (- L) C ∈ set M ⟹ C > 0 ⟹ length (NU ∝ C) ≥ 2› for C
  using trail-length-ge2[OF S'-S'' add-inv struct-invs, of - C] S-S'
  by (auto simp: S)
 have [simp]: ‹Propagated (- L) C ∈ set M ⟹ C > 0 ⟹ C ∈# dom-m NU› for C
```

```
    using add-inv S-S' S'-S'' propagated-L[of C]
    by (auto simp: S twl-list-invs-def state-wl-l-def
        twl-st-l-def)
  show ?thesis
    unfolding literal-redundant-wl-def literal-redundant-def
    apply (refine-rcg H get-propagation-reason)
    subgoal by simp
    subgoal using M'-def by simp
    subgoal using M'-def by (auto simp: lit-redundant-rec-wl-ref-def)
    subgoal by simp
    subgoal by (auto simp: lit-redundant-rec-wl-ref-def)
    apply (assumption)
    subgoal by (auto simp: lit-redundant-rec-wl-ref-def)
    subgoal by simp
    subgoal by simp
    subgoal for x x' C x'a
      using le2[of C] L-watched[of C] L-dist[of C] L-tauto[of C]
      by (auto simp: convert-analysis-list-def drop-Suc slice-0
          lit-redundant-reason-stack-def slice-Suc slice-head slice-end
        dest!: list-decomp-2)
    subgoal for x x' C x'a
      using le2[of C] L-watched[of C] L-dist[of C] L-tauto[of C]
      by (auto simp: convert-analysis-list-def drop-Suc slice-0
          lit-redundant-reason-stack-def slice-Suc slice-head slice-end
        dest!: list-decomp-2)
    subgoal for x x' C x'a
      using le2[of C] L-watched[of C] L-dist[of C] L-tauto[of C]
      by (auto simp: convert-analysis-list-def drop-Suc slice-0
          lit-redundant-reason-stack-def slice-Suc slice-head slice-end
        dest!: list-decomp-2)
    subgoal for x x' C x'a
      using le2[of C] L-watched[of C] L-dist[of C] L-tauto[of C]
      by (auto simp: lit-redundant-reason-stack-def lit-redundant-rec-wl-ref-def)
    done
qed

definition mark-failed-lits-stack-inv where
  ‹mark-failed-lits-stack-inv NU analyse = (λcach.
      (∀(i, k, j, len) ∈ set analyse. j ≤ len ∧ len ≤ length (NU ∝ i) ∧ i ∈# dom-m NU ∧
        k < length (NU ∝ i) ∧ j > 0))›
```

We mark all the literals from the current literal stack as failed, since every minimisation call will find the same minimisation problem.

```
definition mark-failed-lits-stack where
  ‹mark-failed-lits-stack A_in NU analyse cach = do {
    ( -, cach) ← WHILE_T^λ(-, cach). mark-failed-lits-stack-inv NU analyse cach
      (λ(i, cach). i < length analyse)
      (λ(i, cach). do {
        ASSERT(i < length analyse);
        let (cls-idx, -, idx, -) = analyse ! i;
        ASSERT(atm-of (NU ∝ cls-idx ! (idx − 1)) ∈# A_in);
        RETURN (i+1, cach (atm-of (NU ∝ cls-idx ! (idx − 1)) := SEEN-FAILED))
      })
      (0, cach);
    RETURN cach
```

```
  }›

lemma mark-failed-lits-stack-mark-failed-lits-wl:
  shows
    ‹(uncurry2 (mark-failed-lits-stack 𝒜), uncurry2 mark-failed-lits-wl) ∈
      [λ((NU, analyse), cach). literals-are-in-ℒ_in-mm 𝒜 (mset `# ran-mf NU) ∧
         mark-failed-lits-stack-inv NU analyse cach]_f
      Id ×_f Id ×_f Id → ⟨Id⟩nres-rel›
proof −
  have ‹mark-failed-lits-stack 𝒜 NU analyse cach ≤ (mark-failed-lits-wl NU analyse cach)›
    if
      NU-ℒ_in: ‹literals-are-in-ℒ_in-mm 𝒜 (mset `# ran-mf NU)› and
      init: ‹mark-failed-lits-stack-inv NU analyse cach›
    for NU analyse cach
  proof −
    define I where
    ‹I = (λ(i :: nat, cach'). (∀ L. cach' L = SEEN-REMOVABLE ⟶ cach L = SEEN-REMOVABLE))›
    have valid-atm: ‹atm-of (NU ∝ cls-idx ! (idx − 1)) ∈# 𝒜›
      if
        ‹I s› and
        ‹case s of (i, cach) ⇒ i < length analyse› and
        ‹case s of (i, cach) ⇒ mark-failed-lits-stack-inv NU analyse cach› and
        ‹s = (i, cach)› and
        i: ‹i < length analyse› and
        ‹analyse ! i = (cls-idx, k)› ‹k = (k0, k')› ‹k' = (idx, len)›
      for s i cach cls-idx idx k len k' k'' k0
    proof −
      have [iff]: ‹(∀ a b. (a, b) ∉ set analyse) ⟷ False›
        using i by (cases analyse) auto
      show ?thesis
        unfolding in-ℒ_all-atm-of-in-atms-of-iff[symmetric]
        apply (subst atms-of-ℒ_all-𝒜_in[symmetric])
        unfolding atms-of-def
        apply (rule imageI)
        apply (rule literals-are-in-ℒ_in-mm-in-ℒ_all)
        using NU-ℒ_in that nth-mem[of i analyse]
        by (auto simp: mark-failed-lits-stack-inv-def I-def)
    qed
    show ?thesis
      unfolding mark-failed-lits-stack-def mark-failed-lits-wl-def
      apply (refine-vcg WHILEIT-rule-stronger-inv[where R = ‹measure (λ(i, -). length analyse −i)›
        and I' = I])
      subgoal by auto
      subgoal using init by simp
      subgoal unfolding I-def by auto
      subgoal by auto
      subgoal for s i cach cls-idx idx
        by (rule valid-atm)
      subgoal unfolding mark-failed-lits-stack-inv-def by auto
      subgoal unfolding I-def by auto
      subgoal by auto
      subgoal unfolding I-def by auto
      done
  qed
  then show ?thesis
    by (intro frefI nres-relI) auto
```

qed

end