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 $R :: ‹- \Rightarrow - \Rightarrow - \Rightarrow bool›$
shows $‹R^d *_a id\text{-}assn^d = (R \times_a id\text{-}assn)^d›$
⟨*proof*⟩

lemma *Exists-eq-simp*: $‹(\exists x.\ (P\ x \wedge* \uparrow (x = b))\ s) \longleftrightarrow P\ b\ s›$
⟨*proof*⟩

lemma $‹(\uparrow(x = b))\ s \longleftrightarrow x = b \wedge \square\ s›$
⟨*proof*⟩
lemma *split-conj-is-pure*:
assumes ‹*Sepref-Basic.is-pure B*›
shows $‹(B\ b\ bi \wedge* R)\ s = ((bi,\ b) \in the\text{-}pure\ B \wedge R\ s)›$ (**is** *?A*)
⟨*proof*⟩
lemma *split-conj-is-pure2*:
assumes ‹*Sepref-Basic.is-pure B*›
shows
$‹(R1 \wedge* B\ b\ bi \wedge* R)\ s = ((bi,\ b) \in the\text{-}pure\ B \wedge (R1 \wedge* R)\ s)›$ (**is** *?B*)
⟨*proof*⟩

lemma *nempty-list-mset-rel-iff*: $‹M \neq \{\#\} \Longrightarrow$
$(xs,\ M) \in list\text{-}mset\text{-}rel \longleftrightarrow (xs \neq [] \wedge hd\ xs \in\# M \wedge$
$(tl\ xs,\ remove1\text{-}mset\ (hd\ xs)\ M) \in list\text{-}mset\text{-}rel)›$
⟨*proof*⟩

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 ≡ *uncurry2* (*uncurry2 f*)

abbreviation *uncurry6'* **where**
uncurry6' f ≡ *uncurry2* (*uncurry4' f*)

lemma *hrp-comp-Id2*[*simp*]: ‹*hrp-comp A Id* = *A*›
⟨*proof*⟩

lemma *list-rel-update*:
assumes *rel*: ‹(*xs*, *ys*) ∈ ⟨*the-pure R*⟩*list-rel*› **and**
h: ‹*R b bi s*› **and**
p: ‹*is-pure R*›
shows ‹(*list-update xs ba bi*, *list-update ys ba b*) ∈ ⟨*the-pure R*⟩*list-rel*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *sublist-single*: ‹*i* < *length xs* ⟹ *sublist xs i i* = [*xs*!*i*]›
⟨*proof*⟩

lemma *insert-eq*: ‹*insert a b* = *b* ∪ {*a*}›
⟨*proof*⟩

lemma *sublist-nth*: ‹⟦*lo* ≤ *hi*; *hi* < *length xs*; *k*+*lo* ≤ *hi*⟧ ⟹ (*sublist xs lo hi*)!*k* = *xs*!(*lo*+*k*)›
⟨*proof*⟩

lemma *sublist-length*: ‹⟦*i* ≤ *j*; *j* < *length xs*⟧ ⟹ *length* (*sublist xs i j*) = *1* + *j* − *i*›
⟨*proof*⟩

lemma *sublist-not-empty*: ‹⟦*i* ≤ *j*; *j* < *length xs*; *xs* ≠ []⟧ ⟹ *sublist xs i j* ≠ []›
⟨*proof*⟩

lemma *sublist-app*: ‹⟦*i1* ≤ *i2*; *i2* ≤ *i3*⟧ ⟹ *sublist xs i1 i2* @ *sublist xs* (*Suc i2*) *i3* = *sublist xs i1 i3*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *sorted-sublist-wrt-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist-wrt R xs i i*›
⟨*proof*⟩

lemma *sorted-sublist-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist xs i i*›
⟨*proof*⟩

lemma *sorted-sublist-map-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist-map R h xs i i*›
⟨*proof*⟩

lemma *sublist-map*: ‹*sublist* (*map f xs*) *i j* = *map f* (*sublist xs i j*)›
⟨*proof*⟩

lemma *take-set*: ‹*j* ≤ *length xs* ⟹ *x* ∈ *set* (*take j xs*) ≡ (∃ *k*. *k* < *j* ∧ *xs*!*k* = *x*)›
⟨*proof*⟩

lemma *drop-set*: ‹*j* ≤ *length xs* ⟹ *x* ∈ *set* (*drop j xs*) ≡ (∃ *k*. *j*≤*k*∧*k*<*length xs* ∧ *xs*!*k*=*x*)›
⟨*proof*⟩

lemma *sublist-el*: ‹*i* ≤ *j* ⟹ *j* < *length xs* ⟹ *x* ∈ *set* (*sublist xs i j*) ≡ (∃ *k*. *k* < *Suc j*−*i* ∧ *xs*!(*i*+*k*)=*x*)›
⟨*proof*⟩

lemma *sublist-el'*: ‹*i* ≤ *j* ⟹ *j* < *length xs* ⟹ *x* ∈ *set* (*sublist xs i j*) ≡ (∃ *k*. *i*≤*k*∧*k*≤*j* ∧ *xs*!*k*=*x*)›
⟨*proof*⟩

lemma *sublist-lt*: ‹*hi* < *lo* ⟹ *sublist xs lo hi* = []›
⟨*proof*⟩

lemma *nat-le-eq-or-lt*: ‹(*a* :: *nat*) ≤ *b* = (*a* = *b* ∨ *a* < *b*)›
⟨*proof*⟩

lemma *sorted-sublist-wrt-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist-wrt R xs lo hi*›
⟨*proof*⟩

Elements in a sorted sublists are actually sorted

lemma *sorted-sublist-wrt-nth-le*:
assumes ‹*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*⟩

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

lemma *sorted-sublist-wrt-nth-le'*:
assumes *ref*: ‹⋀ *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*⟩

lemma *sorted-sublist-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist xs lo hi*›
⟨*proof*⟩

lemma *sorted-sublist-map-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist-map R h xs lo hi*›
⟨*proof*⟩

lemma *sublist-cons*: ‹*lo* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo hi* = *xs*!*lo* # *sublist xs* (*Suc lo*) *hi*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

lemma *sublist-snoc*: ‹*lo* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo hi* = *sublist xs lo* (*hi−1*) @ [*xs*!*hi*]›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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

lemma *isPartition-wrt-trans*:
‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*) ⟹

isPartition-wrt R xs lo hi p $\Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j))$›
⟨*proof*⟩

lemma *isPartition-map-trans*:
‹$(\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
hi < *length xs* $\Longrightarrow$
isPartition-map R h xs lo hi p $\Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (h\ (xs!i))\ (h\ (xs!j)))$›
⟨*proof*⟩

lemma *merge-sorted-wrt-partitions-between′*:
‹*lo* ≤ *hi* $\Longrightarrow$ *lo* < *p* $\Longrightarrow$ *p* < *hi* $\Longrightarrow$ *hi* < *length xs* $\Longrightarrow$
isPartition-wrt R xs lo hi p $\Longrightarrow$
sorted-sublist-wrt R xs lo (*p*−1) $\Longrightarrow$ *sorted-sublist-wrt R xs* (*p*+1) *hi* $\Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
sorted-sublist-wrt R xs lo hi›
⟨*proof*⟩

lemma *merge-sorted-wrt-partitions-between*:
‹$(\bigwedge\ x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z) \Longrightarrow$
isPartition-wrt R xs lo hi p $\Longrightarrow$
sorted-sublist-wrt R xs lo (*p*−1) $\Longrightarrow$ *sorted-sublist-wrt R xs* (*p*+1) *hi* $\Longrightarrow$
lo ≤ *hi* $\Longrightarrow$ *hi* < *length xs* $\Longrightarrow$ *lo* < *p* $\Longrightarrow$ *p* < *hi* $\Longrightarrow$ *hi* < *length xs* $\Longrightarrow$
sorted-sublist-wrt R xs lo hi›
⟨*proof*⟩

The main theorem to merge sorted lists

lemma *merge-sorted-wrt-partitions*:
‹*isPartition-wrt R xs lo hi p* $\Longrightarrow$
sorted-sublist-wrt R xs lo (*p* − *Suc 0*) $\Longrightarrow$ *sorted-sublist-wrt R xs* (*Suc p*) *hi* $\Longrightarrow$
lo ≤ *hi* $\Longrightarrow$ *lo* ≤ *p* $\Longrightarrow$ *p* ≤ *hi* $\Longrightarrow$ *hi* < *length xs* $\Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
sorted-sublist-wrt R xs lo hi›
⟨*proof*⟩

theorem *merge-sorted-map-partitions*:
‹$(\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
isPartition-map R h xs lo hi p $\Longrightarrow$
sorted-sublist-map R h xs lo (*p* − *Suc 0*) $\Longrightarrow$ *sorted-sublist-map R h xs* (*Suc p*) *hi* $\Longrightarrow$
lo ≤ *hi* $\Longrightarrow$ *lo* ≤ *p* $\Longrightarrow$ *p* ≤ *hi* $\Longrightarrow$ *hi* < *length xs* $\Longrightarrow$
sorted-sublist-map R h xs lo hi›
⟨*proof*⟩

lemma *partition-wrt-extend*:
‹*isPartition-wrt R xs lo′ hi′ p* $\Longrightarrow$
hi < *length xs* $\Longrightarrow$
lo ≤ *lo′* $\Longrightarrow$ *lo′* ≤ *hi* $\Longrightarrow$ *hi′* ≤ *hi* $\Longrightarrow$
lo′ ≤ *p* $\Longrightarrow$ *p* ≤ *hi′* $\Longrightarrow$
$(\bigwedge\ i.\ lo \le i \Longrightarrow i < lo' \Longrightarrow R\ (xs!i)\ (xs!p)) \Longrightarrow$
$(\bigwedge\ j.\ hi' < j \Longrightarrow j \le hi \Longrightarrow R\ (xs!p)\ (xs!j)) \Longrightarrow$
isPartition-wrt R xs lo hi p›
⟨*proof*⟩

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›
⟨*proof*⟩

lemma *isPartition-empty*:
‹(⋀ *j*. ⟦*lo* < *j*; *j* ≤ *hi*⟧ ⟹ *R* (*xs* ! *lo*) (*xs* ! *j*)) ⟹
isPartition-wrt R xs lo hi lo›
⟨*proof*⟩

lemma *take-ext*:
‹(∀ *i*<*k*. *xs′*!*i*=*xs*!*i*) ⟹
k < *length xs* ⟹ *k* < *length xs′* ⟹
take k xs′ = *take k xs*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *drop-ext*:
‹(∀ *i*. *i*≥*k* ∧ *i*<*length xs* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
length xs′=*length xs* ⟹
drop k xs′ = *drop k xs*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *lt-Suc*: ‹(*a* < *b*) = (*Suc a* = *b* ∨ *Suc a* < *b*)›
⟨*proof*⟩

lemma *sublist-until-end-eq-drop*: ‹*Suc hi* = *length xs* ⟹ *sublist xs lo hi* = *drop lo xs*›
⟨*proof*⟩

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›
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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 ≤ p ∧ *p ≤ hi* ∧ — The partition index is in bounds
(∀ *i. i<lo ⟶ xs′!i=xs!i*) ∧ (∀ *i. hi<i∧i<length xs′ ⟶ 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*: ‹⋀ *i. i<lo ⟹ xs′!i = xs!i*› ‹⋀ *j.* ⟦*hi<j*; *j<length xs*⟧ ⟹ *xs′!j = xs!j*›
shows ‹∃ *j. lo≤j∧j≤hi* ∧ *xs!j = x*›
⟨*proof*⟩

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*: ‹$\bigwedge i.\ i<lo \Longrightarrow xs'!i = xs!i$› ‹$\bigwedge j.\ [\![hi<j;\ j<length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
and *bounds*: ‹$lo \le hi$› ‹$hi < length\ xs$›
shows ‹$set\ (sublist\ xs'\ lo\ hi) \subseteq set\ (sublist\ xs\ lo\ hi)$›
⟨*proof*⟩

lemma *mset-sublist-eq*:
assumes ‹$mset\ xs' = mset\ xs$›
and ‹$\bigwedge i.\ i<lo \Longrightarrow xs'!i = xs!i$›
and ‹$\bigwedge j.\ [\![hi<j;\ j<length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
and *bounds*: ‹$lo \le hi$› ‹$hi < length\ xs$›
shows ‹$set\ (sublist\ xs'\ lo\ hi) = set\ (sublist\ xs\ lo\ hi)$›
⟨*proof*⟩

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

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

definition *quicksort-post* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$› **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'*›
⟨*proof*⟩

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

lemma *quicksort-correct-case1*:
assumes *trans*: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs' p*›
and *ifs*: ‹$p-1 \le lo$› ‹$hi \le p+1$›
shows ‹*quicksort-post R h lo hi xs xs'*›

⟨*proof*⟩

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*⟩

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*⟩

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*⟩

In the 4th case, we have to show that the premise holds for (*lo*, *p* − (*1*::'*b*), *xs'*), in case ¬ *p* − (*1*::'*a*) ≤ *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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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-main-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*⟩

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *partition-between-ref-partition-between*:
‹*partition-between-ref R h lo hi xs* ≤ (*partition-between R h lo hi xs*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩
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*⟩

lemma *sublist-entire*:
‹*sublist xs 0* (*length xs* − *1*) = *xs*›
⟨*proof*⟩

lemma *sorted-sublist-wrt-entire*:
assumes ‹*sorted-sublist-wrt R xs 0* (*length xs* − *1*)›
shows ‹*sorted-wrt R xs*›

⟨*proof*⟩

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*⟩

Final correctness lemma

lemma *full-quicksort-correct-sorted*:
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 R h xs* ≤ ⇓ *Id* (*SPEC*(λ*xs*′. *mset xs*′ = *mset xs* ∧ *sorted-wrt* (λ *x y*. *R* (*h x*) (*h y*)) *xs*′))›
⟨*proof*⟩

lemma *full-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*)›
shows ‹*full-quicksort R h xs* ≤ ⇓ *Id* (*SPEC*(λ*xs*′. *mset xs*′ = *mset xs*))›
⟨*proof*⟩

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* ×
'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' ≠ {#L#}› **and**
‹D' ⊆# D› **and**
‹clause '# (N + U) + NE + UE + NS + US + N0 + U0 ⊨pm D'› **and**
‹L ∈# D'›
‹L' ∈# 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'*›
⟨*proof*⟩

lemma [*twl-st*,*simp*]: ‹*pget-conflict* (*pstate$_W$-of S'*) = *get-conflict S'*›
⟨*proof*⟩

lemma [*twl-st*,*simp*]: ‹*cdcl$_W$-restart-mset.clauses* (*state-of* (*pstate$_W$-of S'*)) = *get-all-clss S'*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
  ⟨*proof*⟩

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*›
⟨*proof*⟩

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* ⟶
¬*twl-is-an-exception C* (*literals-to-update S*) (*clauses-to-update S*) ⟶
(*twl-lazy-update* (*get-trail S*) *C*)) ∧
(∀ *C* ∈# *get-clauses S*. *get-conflict S* = *None* ⟶
watched-literals-false-of-max-level (*get-trail S*) *C*)›
⟨*proof*⟩

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

fun *valid-enqueued* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*valid-enqueued* (*M*, *N*, *U*, *C*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟷
(∀(*L*, *C*) ∈# *WS*. *L* ∈# *watched C* ∧ *C* ∈# *N* + *U* ∧ −*L* ∈ *lits-of-l M* ∧
get-level M L = *count-decided M*) ∧
(∀*L* ∈# *Q*. −*L* ∈ *lits-of-l M* ∧ *get-level M L* = *count-decided M*)›

Putting invariants together:

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

definition *twl-stgy-invs* :: ‹*'v twl-st* ⇒ *bool*› **where**
‹*twl-stgy-invs S* ⟷
cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-invariant* (*state*$_W$*-of S*) ∧
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* ⟷
(*twl-is-an-exception C Q WS* ∨ (*L* ∈# *watched C*))›
⟨*proof*⟩

lemma *twl-is-an-exception-add-mset-to-clauses-to-update*:
‹*twl-is-an-exception C Q* (*add-mset* (*L*, *D*) *WS*) ⟷ (*twl-is-an-exception C Q WS* ∨ *C* = *D*)›
⟨*proof*⟩

lemma *twl-is-an-exception-empty*[*simp*]: ‹¬*twl-is-an-exception C* {#} {#}›
⟨*proof*⟩

lemma *twl-inv-empty-trail*:
shows
‹*watched-literals-false-of-max-level* [] *C*› **and**
‹*twl-lazy-update* [] *C*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma
assumes ‹$\bigwedge C.\ C \in\# N + U \Longrightarrow$ *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*)›
⟨*proof*⟩

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*, {#}, {#})›
⟨*proof*⟩

lemma *twl-st-inv-add-mset-clauses-to-update*:
assumes ‹*D* $\in\#$ *N* + *U*›
shows ‹*twl-st-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)
$\longleftrightarrow$ *twl-st-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *add-mset* (*L*, *D*) *WS*, *Q*) $\wedge$
($\neg$ *twl-is-an-exception D Q WS* $\longrightarrow$*twl-lazy-update M D*)›
⟨*proof*⟩

lemma *twl-st-simps*:
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) $\longleftrightarrow$
($\forall$ *C* $\in\#$ *N* + *U*. *struct-wf-twl-cls C* $\wedge$
(*D* = *None* $\longrightarrow$ ($\neg$*twl-is-an-exception C Q WS* $\longrightarrow$ *twl-lazy-update M C*) $\wedge$
watched-literals-false-of-max-level M C))›
⟨*proof*⟩

lemma *propa-cands-enqueued-unit-clause*:
‹*propa-cands-enqueued* (*M*, *N*, *U*, *C*, *add-mset L NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) $\longleftrightarrow$
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*) $\longleftrightarrow$
propa-cands-enqueued (*M*, *N*, *U*, *C*, {#}, {#}, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
⟨*proof*⟩

lemma *past-invs-enqueud*: ‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) $\longleftrightarrow$
past-invs (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, {#})›
⟨*proof*⟩

lemma *confl-cands-enqueued-unit-clause*:
‹*confl-cands-enqueued* (*M*, *N*, *U*, *C*, *add-mset L NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) $\longleftrightarrow$
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*) $\longleftrightarrow$
confl-cands-enqueued (*M*, *N*, *U*, *C*, {#}, {#}, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
⟨*proof*⟩

lemma *twl-inv-decomp*:
assumes
lazy: ‹*twl-lazy-update M C*› **and**
decomp: ‹(*Decided K # M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition M*)› **and**
n-d: ‹*no-dup M*›
shows
‹*twl-lazy-update M1 C*›
⟨*proof*⟩

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*⟩

lemma *is-blit-Cons*:
  ‹*is-blit* (*K* # *M*) *C L* ⟷ (*L* = *lit-of K* ∧ *lit-of K* ∈# *C*) ∨ *is-blit M C L*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

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

lemma *image-Pair-subset-mset*:
  ‹*Pair L* ‘# *A* ⊆# *Pair L* ‘# *B* ⟷ *A* ⊆# *B*›
⟨*proof*⟩

lemma *count-image-mset-Pair2*:
  ‹*count* {#(*L*, *x*). *L* ∈# *M x*#} (*L*, *C*) = (*if x* = *C then count* (*M x*) *L else* 0)›
⟨*proof*⟩

lemma *lit-of-inj-on-no-dup*: ‹*no-dup M* ⟹ *inj-on* (λ*x*. − *lit-of x*) (*set M*)›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

lemma *distinct-mset-Pair*: ‹*distinct-mset* (*Pair L* ‘# *C*) ⟷ *distinct-mset C*›
  ⟨*proof*⟩

lemma *distinct-image-mset-clause*:
  ‹*distinct-mset* (*clause* ‘# *C*) ⟹ *distinct-mset C*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-cp-conflict*:
‹*cdcl-twl-cp S T* ⟹ *get-conflict T* ≠ *None* ⟶
clauses-to-update T = {#} ∧ *literals-to-update T* = {#}›
⟨*proof*⟩

lemma *cdcl-twl-cp-twl-struct-invs*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹ *twl-struct-invs T*›
⟨*proof*⟩

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*⟩

lemma *cdcl-twl-cp-twl-stgy-invs*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹ *twl-stgy-invs S* ⟹ *twl-stgy-invs T*›
⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*⟩

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*⟩

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*)›
  ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

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*⟩

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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-o-twl-stgy-invs*:
‹*cdcl-twl-o S T* ⟹ *twl-struct-invs S* ⟹ *twl-stgy-invs S* ⟹ *twl-stgy-invs T*›
⟨*proof*⟩

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*)}›
⟨*proof*⟩

lemma *wf-cdcl-twl-cp*:
‹*wf* {(*T*, *S*). *twl-struct-invs S* ∧ *cdcl-twl-cp S T*}› (**is** ‹*wf ?TWL*›)

⟨*proof*⟩

lemma *tranclp-wf-cdcl-twl-cp*:
  ‹*wf* {(*T*, *S*). *twl-struct-invs S* ∧ *cdcl-twl-cp*$^{++}$ *S T*}›
⟨*proof*⟩

lemma *wf-cdcl-twl-stgy*:
  ‹*wf* {(*T*, *S*). *twl-struct-invs S* ∧ *cdcl-twl-stgy S T*}› (**is** ‹*wf ?TWL*›)
⟨*proof*⟩

lemma *tranclp-wf-cdcl-twl-stgy*:
  ‹*wf* {(*T*, *S*). *twl-struct-invs S* ∧ *cdcl-twl-stgy*$^{++}$ *S T*}›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-o-stgyD*: ‹*cdcl-twl-o*** *S T* ⟹ *cdcl-twl-stgy*** *S T*›
  ⟨*proof*⟩

lemma *rtranclp-cdcl-twl-cp-stgyD*: ‹*cdcl-twl-cp*** *S T* ⟹ *cdcl-twl-stgy*** *S T*›
  ⟨*proof*⟩

lemma *tranclp-cdcl-twl-o-stgyD*: ‹*cdcl-twl-o*$^{++}$ *S T* ⟹ *cdcl-twl-stgy*$^{++}$ *S T*›
  ⟨*proof*⟩

lemma *tranclp-cdcl-twl-cp-stgyD*: ‹*cdcl-twl-cp*$^{++}$ *S T* ⟹ *cdcl-twl-stgy*$^{++}$ *S T*›
  ⟨*proof*⟩

lemma *wf-cdcl-twl-o*:
  ‹*wf* {(*T*, *S*::'*v twl-st*). *twl-struct-invs S* ∧ *cdcl-twl-o S T*}›
  ⟨*proof*⟩

lemma *tranclp-wf-cdcl-twl-o*:
  ‹*wf* {(*T*, *S*::'*v twl-st*). *twl-struct-invs S* ∧ *cdcl-twl-o*$^{++}$ *S T*}›
  ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

lemma (**in** −)*confl-cands-enqueued-mono*:
  ‹*U'* ⊆# *U* ⟹ *N'* ⊆# *N* ⟹
    confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
     confl-cands-enqueued (*M*, *N'*, *U'*, *D*, *NE'*, *UE'*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
  ⟨*proof*⟩

lemma (**in** −)*twl-st-exception-inv-mono*:
  ‹*U'* ⊆# *U* ⟹ *N'* ⊆# *N* ⟹
    twl-st-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
     twl-st-exception-inv (*M*, *N'*, *U'*, *D*, *NE'*, *UE'*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*)›
  ⟨*proof*⟩

lemma (**in** −)*twl-st-inv-mono*:
  ‹*U'* ⊆# *U* ⟹ *N'* ⊆# *N* ⟹
    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*)›

⟨*proof*⟩

lemma (**in** −)*propa-cands-enqueued-subsumed-mono*:
‹*US'* ⊆# *US* ⟹
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*)›
⟨*proof*⟩

lemma (**in** −)*propa-cands-enqueued-U0-mono*:
‹*U0'* ⊆# *U0* ⟹
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*)›
⟨*proof*⟩

lemma (**in** −)*confl-cands-enqueued-subsumed-mono*:
‹*US'* ⊆# *US* ⟹
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US'*, *N0*, *U0*, *WS*, *Q*)›
⟨*proof*⟩

lemma (**in** −)*confl-cands-enqueued-U0-mono*:
‹*U0'* ⊆# *U0* ⟹
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0'*, *WS*, *Q*)›
⟨*proof*⟩

lemma (**in** −)*twl-st-exception-inv-subsumed-mono*:
‹*US'* ⊆# *US* ⟹
twl-st-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
twl-st-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US'*, *N0*, *U0*, *WS*, *Q*)›
⟨*proof*⟩

lemma (**in** −)*twl-st-exception-inv-U0-mono*:
‹*U0'* ⊆# *U0* ⟹
twl-st-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*) ⟹
twl-st-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *NS*, *US*, *N0*, *U0'*, *WS*, *Q*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›

⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

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*)›
⟨*proof*⟩

lemma *learned-clss-l-update*[*simp*]:
‹*bh* ∈# *dom-m ax* ⟹ *size* (*learned-clss-l* (*ax*(*bh* ↪ *C*))) = *size* (*learned-clss-l ax*)›
⟨*proof*⟩

lemma *Ball-ran-m-dom*:
‹(∀ *x*∈#*ran-m N*. *P* (*fst x*)) ⟷ (∀ *x*∈#*dom-m N*. *P* (*N* ∝ *x*))›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *init-clss-lf-fmdrop-irrelev*[*simp*]:
‹¬*irred N C* ⟹ *init-clss-lf* (*fmdrop C N*) = *init-clss-lf N*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *learned-clss-lf-lf-fmdrop-irrelev*[*simp*]:
‹*irred N C* ⟹ *learned-clss-lf* (*fmdrop C N*) = *learned-clss-lf N*›
⟨*proof*⟩

lemma *ran-mf-lf-fmdrop*[*simp*]:
‹*C* ∈# *dom-m N* ⟹ *ran-mf* (*fmdrop C N*) = *remove1-mset* (*N*∝*C*) (*ran-mf N*)›
⟨*proof*⟩

lemma *ran-mf-lf-fmdrop-notin*[*simp*]:
‹*C* ∉# *dom-m N* ⟹ *ran-mf* (*fmdrop C N*) = *ran-mf N*›

⟨*proof*⟩

lemma *lookup-None-notin-dom-m*[*simp*]:
‹*fmlookup N i* = *None* ⟷ *i* ∉# *dom-m N*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *ran-m-fmdrop-notin*:
‹*C* ∉# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *ran-m N*›
⟨*proof*⟩

lemma *init-clss-l-fmdrop-irrelev*:
‹¬*irred N C* ⟹ *init-clss-l* (*fmdrop C N*) = *init-clss-l N*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *ran-m-lf-fmdrop*:
‹*C* ∈# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*)) (*ran-m N*)›
⟨*proof*⟩
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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-irrel-notin*: ‹*C* ∉# *dom-m N* ⟹

init-clss-l (*fmupd C* (*C′*, *False*) *N*) = *init-clss-l N*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *in-ran-mf-clause-inI*[*intro*]:
‹*C* ∈# *dom-m N* ⟹ *i* = *irred N C* ⟹ (*N* ∝ *C*, *i*) ∈# *ran-m N*›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-notin*:
‹*C* ∉# *dom-m N* ⟹ *init-clss-l* (*fmupd C* (*C′*, *True*) *N*) =
add-mset (*C′*, *True*) (*init-clss-l N*)›
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd-notin-irrelev*: ‹*C* ∉# *dom-m N* ⟹
learned-clss-l (*fmupd C* (*C′*, *True*) *N*) = *learned-clss-l N*›
⟨*proof*⟩

lemma *clause-twl-clause-of*: ‹*clause* (*twl-clause-of C*) = *mset C*› **for** *C*
⟨*proof*⟩

lemma *learned-clss-l-l-fmdrop-irrelev*: ‹*irred N C* ⟹
learned-clss-l (*fmdrop C N*) = *learned-clss-l N*›
⟨*proof*⟩

lemma *init-clss-l-fmdrop-if*:
‹*C* ∈# *dom-m N* ⟹ *init-clss-l* (*fmdrop C N*) = (*if irred N C then remove1-mset* (*the* (*fmlookup N C*)) (*init-clss-l N*)
else init-clss-l N)›
⟨*proof*⟩

lemma *init-clss-l-fmupd-if*:
‹*C′* ∉# *dom-m new* ⟹ *init-clss-l* (*fmupd C′ D new*) = (*if snd D then add-mset D* (*init-clss-l new*) *else init-clss-l new*)›
⟨*proof*⟩

lemma *learned-clss-l-fmdrop-if*:
‹*C* ∈# *dom-m N* ⟹ *learned-clss-l* (*fmdrop C N*) = (*if* ¬*irred N C then remove1-mset* (*the* (*fmlookup N C*)) (*learned-clss-l N*)
else learned-clss-l N)›
⟨*proof*⟩

lemma *learned-clss-l-fmupd-if*:
‹*C′* ∉# *dom-m new* ⟹ *learned-clss-l* (*fmupd C′ D new*) = (*if* ¬*snd D then add-mset D* (*learned-clss-l new*) *else learned-clss-l new*)›
⟨*proof*⟩

definition *op-clauses-at* :: ‹*′v clauses-l* ⇒ *nat* ⇒ *nat* ⇒ *′v literal*› **where**

‹*op-clauses-at N C i* = *N* ∝ *C* ! *i*›

definition *mop-clauses-at* :: ‹'*v clauses-l* ⇒ *nat* ⇒ *nat* ⇒ '*v literal nres*› **where**
‹*mop-clauses-at N C i* = *do* {
  ASSERT(*C* ∈# *dom-m N*);
  ASSERT(*i* < *length* (*N* ∝ *C*));
  RETURN (*N* ∝ *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*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

lemma *mop-clauses-at-itself*:
  ‹(*uncurry2 mop-clauses-at*, *uncurry2 mop-clauses-at*) ∈ *Id* ×$_f$ *Id* ×$_f$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
  ⟨*proof*⟩

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'*)›
  ⟨*proof*⟩

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'*)›
  ⟨*proof*⟩

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'*))›
  ⟨*proof*⟩

lemma *mop-clauses-swap-itself*:

‹(*uncurry3 mop-clauses-swap*, *uncurry3 mop-clauses-swap*) ∈ *Id* ×$_f$ *Id* ×$_f$ *Id* ×$_f$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
⟨*proof*⟩

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'*)›
⟨*proof*⟩

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'*)›
⟨*proof*⟩

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* {#} = {#}›
⟨*proof*⟩
lemma *in-all-lits-of-mm-ain-atms-of-iff*:
‹*L* ∈# *all-lits-of-mm N* ⟷ *atm-of L* ∈ *atms-of-mm N*›
⟨*proof*⟩

lemma *all-lits-of-mm-union*:
‹*all-lits-of-mm* (*M* + *N*) = *all-lits-of-mm M* + *all-lits-of-mm N*›
⟨*proof*⟩

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* {#} = {#}›
⟨*proof*⟩

lemma *all-lits-of-m-empty-iff*[*iff*]: ‹*all-lits-of-m A* = {#} ⟷ *A* = {#}›
⟨*proof*⟩

lemma *in-all-lits-of-m-ain-atms-of-iff*: ‹*L* ∈# *all-lits-of-m N* ⟷ *atm-of L* ∈ *atms-of N*›
⟨*proof*⟩

lemma *in-clause-in-all-lits-of-m*: ‹*x* ∈# *C* ⟹ *x* ∈# *all-lits-of-m C*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*))›

⟨*proof*⟩

lemma *all-lits-of-m-union*:
‹*all-lits-of-m* (*A* + *B*) = *all-lits-of-m A* + *all-lits-of-m B*›
⟨*proof*⟩

lemma *all-lits-of-m-mono*:
‹*D* ⊆# *D'* ⟹ *all-lits-of-m D* ⊆# *all-lits-of-m D'*›
⟨*proof*⟩

lemma *all-lits-of-m-diffD*: ‹*x* ∈# *all-lits-of-m* (*D* − *D'*) ⟹ *x* ∈# *all-lits-of-m D*›
⟨*proof*⟩

lemma *in-all-lits-of-mm-uminusD*: ‹*x2* ∈# *all-lits-of-mm N* ⟹ −*x2* ∈# *all-lits-of-mm N*›
⟨*proof*⟩

lemma *in-all-lits-of-mm-uminus-iff*: ‹−*x2* ∈# *all-lits-of-mm N* ⟷ *x2* ∈# *all-lits-of-mm N*›
⟨*proof*⟩

lemma *all-lits-of-mm-diffD*:
‹*L* ∈# *all-lits-of-mm* (*A* − *B*) ⟹ *L* ∈# *all-lits-of-mm A*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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* $\mathcal{A}_{in}$›
⟨*proof*⟩

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-*$\mathcal{L}_{in}$ {#}›
⟨*proof*⟩

lemma *in-*$\mathcal{L}_{all}$*-atm-of-in-atms-of-iff*: ‹*x* ∈# $\mathcal{L}_{all}$ ⟷ *atm-of x* ∈ *atms-of* $\mathcal{L}_{all}$›
⟨*proof*⟩

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* ∈# $\mathcal{L}_{all}$›
⟨*proof*⟩

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*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-sub*:
‹*literals-are-in-$\mathcal{L}_{in}$ y* ⟹ *literals-are-in-$\mathcal{L}_{in}$* (*y* − *z*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *all-lits-of-m-remdups-mset*:
‹*set-mset* (*all-lits-of-m* (*remdups-mset N*)) = *set-mset* (*all-lits-of-m N*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *uminus-$\mathcal{A}_{in}$-iff*: ‹− *L* ∈# $\mathcal{L}_{all}$ ⟷ *L* ∈# $\mathcal{L}_{all}$›
⟨*proof*⟩

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* ∈# $\mathcal{L}_{all}$›
⟨*proof*⟩

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*›
⟨*proof*⟩

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* ⟹ *a* ∈ *lits-of-l M* ⟹ *a* ∈# $\mathcal{L}_{all}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-uminus-in-lits-of-l*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* ⟹ −*a* ∈ *lits-of-l M* ⟹ *a* ∈# $\mathcal{L}_{all}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-uminus-in-lits-of-l-atms*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail M* ⟹ −*a* ∈ *lits-of-l M* ⟹ *atm-of a* ∈# $\mathcal{A}_{in}$›
⟨*proof*⟩
end

lemma *isasat-input-ops-$\mathcal{L}_{all}$-empty*[*simp*]:
‹$\mathcal{L}_{all}$ {#} = {#}›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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* ∈# *dom-m xs*› **and** *j-xs*: ‹*j* < *length* (*xs* ∝ *i*)›
shows ‹*xs* ∝ *i* ! *j* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l-atms*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}_{in}$ *M* ⟹ *a* ∈ *lits-of-l M* ⟹ *atm-of a* ∈# $\mathcal{A}_{in}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-Cons*:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}_{in}$ (*L* # *M*) ⟷
literals-are-in-$\mathcal{L}_{in}$-trail $\mathcal{A}_{in}$ *M* ∧ *lit-of L* ∈# $\mathcal{L}_{all}$ $\mathcal{A}_{in}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-empty*[*simp*]:
‹*literals-are-in-$\mathcal{L}_{in}$-trail* $\mathcal{A}$ []›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-in-mset-$\mathcal{L}_{all}$*:
‹*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}$ *C* ⟹ *L* ∈# *C* ⟹ *L* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
⟨*proof*⟩

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 ! $i \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}$›
⟨*proof*⟩

lemma *is-*$\mathcal{L}_{all}$*-*$\mathcal{L}_{all}$*-rewrite*[*simp*]:
‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* $\mathcal{A}'$) ⟹
set-mset ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits-of-mm* $\mathcal{A}'$)) = *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$)›
⟨*proof*⟩

lemma *is-*$\mathcal{L}_{all}$*-alt-def*: ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* A) ⟷ *atms-of* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *atms-of-mm* A›
⟨*proof*⟩

lemma *in-*$\mathcal{L}_{all}$*-atm-of-*$\mathcal{A}_{in}$: ‹$L \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}_{in}$ ⟷ *atm-of* $L \in\#$ $\mathcal{A}_{in}$›
⟨*proof*⟩

lemma *literals-are-in-*$\mathcal{L}_{in}$*-alt-def*:
‹*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}$ S ⟷ *atms-of* $S \subseteq$ *atms-of* ($\mathcal{L}_{all}$ $\mathcal{A}$)›
⟨*proof*⟩

lemma *literals-are-in-*$\mathcal{L}_{in}$*-trail-atm-of*:
‹*literals-are-in-*$\mathcal{L}_{in}$*-trail* $\mathcal{A}_{in}$ M ⟷ *atm-of* ‘ *lits-of-l* $M \subseteq$ *set-mset* $\mathcal{A}_{in}$›
⟨*proof*⟩

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))) ⟷ *literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ C›
⟨*proof*⟩

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))) ⟷ *literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ C›
⟨*proof*⟩

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›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma $\mathcal{L}_{all}$*-union*:
‹*set-mset* ($\mathcal{L}_{all}$ (A + B)) = *set-mset* ($\mathcal{L}_{all}$ A) ∪ *set-mset* ($\mathcal{L}_{all}$ B)›
⟨*proof*⟩

lemma $\mathcal{L}_{all}$*-cong*:
‹*set-mset* A = *set-mset* B ⟹ *set-mset* ($\mathcal{L}_{all}$ A) = *set-mset* ($\mathcal{L}_{all}$ B)›
⟨*proof*⟩

lemma *atms-of-*$\mathcal{L}_{all}$*-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *atms-of* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *atms-of* ($\mathcal{L}_{all}$ $\mathcal{B}$)›
⟨*proof*⟩

lemma *is-*$\mathcal{L}_{all}$*-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *is-*$\mathcal{L}_{all}$ $\mathcal{A}$ = *is-*$\mathcal{L}_{all}$ $\mathcal{B}$›
⟨*proof*⟩

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)
      }
    })
    (S_0, 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⟩

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' = {#})›
 ⟨proof⟩

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 (λL. 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*⟩

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*⟩

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 S₀ =*
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*(λ*T*. *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*›)
⟨*proof*⟩

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(λ*S'*. ∃ *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' ⊆# the (get-conflict S) ∧ clause `# (get-clauses S) + unit-clss S + subsumed-clauses S +
get-all-clauses0 S ⊨pm D' ∧
−lit-of (hd (get-trail S)) ∈# D')}›
⟨*proof*⟩

definition *reduce-trail-bt* :: ‹*'v literal ⇒ 'v twl-st ⇒ 'v twl-st nres*› **where**
‹*reduce-trail-bt = (λL (M, N, U, D', NE, UE, WS, Q). do {*
M1 ← SPEC(λM1. ∃ 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 = (λL 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 = (λL 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* = (λ*L 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(λ*L*. *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* = (λ*S L*. *do* {
ASSERT(*distinct-mset* (*the* (*get-conflict S*)) ∧ −*L* ∈# *the* (*get-conflict S*));
SPEC(λ*L'*. *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 (λT. 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⟩

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)›
  ⟨proof⟩
```

lemma *skip-and-resolve-conflict-before*:
assumes ‹*cdcl-twl-o S T*› ‹*get-conflict T ≠ None*›
shows ‹*get-conflict S ≠ None*›
⟨*proof*⟩

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)›
⟨*proof*⟩

lemma *empty-conflict-lvl0*:
‹*twl-stgy-invs T ⟹ get-conflict T = Some {#} ⟹ count-decided (get-trail T) = 0*›
⟨*proof*⟩

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*⟩

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 S$_0$ ≡ λ(brk, T). twl-struct-invs T ∧ twl-stgy-invs T ∧*
*(brk ⟶ final-twl-state T) ∧ cdcl-twl-stgy** S$_0$ 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 S$_0$ =*
do {
do {
(*brk, T*) ← *WHILE$_T$*$^{cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}inv\ S_0}$
(*λ(brk, -). ¬brk*)
(*λ(brk, S).*
do {
T ← unit-propagation-outer-loop S;

```
        cdcl-twl-o-prog T
      })
      (False, S_0);
    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*⟩

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*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-break* :: ‹*'v twl-st* ⇒ *'v twl-st nres*› **where**
‹*cdcl-twl-stgy-prog-break* 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);
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*⟩

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*›
⟨*proof*⟩

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 $S \leq$ partial-conclusive-TWL-norestart-run S⟩
⟨*proof*⟩

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)⟩
⟨*proof*⟩

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⟩
⟨*proof*⟩

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⟩
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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:
‹*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*)›
if
‹*clause C* = *add-mset L D*›
‹*clause C'* = *add-mset* (−*L*) *D'*›
‹*count-decided M* = *0*› ‹*D* ⊆# *D'*›
‹*clause E* = *remdups-mset D'*› ‹¬*tautology* (*D* + *D'*)› ‹*size* (*watched E*) = *2*› ‹∀ *L* ∈# *clause E*. *undefined-lit M L*› |
twl-subresolution-LL-unit:

‹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)›
if
‹clause C = add-mset L D›
‹clause C' = add-mset (−L) D'›
‹count-decided M = 0› ‹D ⊆# D'›
‹remdups-mset D' = {#K#}› ‹¬tautology (D + D')› ‹undefined-lit M K›|

twl-subresolution-LI-nonunit:
‹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)›
if
‹clause C = add-mset L D›
‹clause C' = add-mset (−L) D'›
‹count-decided M = 0› ‹D ⊆# D'›
‹clause E = remdups-mset D'› ‹¬tautology (D + D')› ‹size (watched E) = 2› ‹∀ L ∈# clause E. undefined-lit M L›|
twl-subresolution-LI-unit:
‹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)›
if
‹clause C = add-mset L D›
‹clause C' = add-mset (−L) D'›
‹count-decided M = 0› ‹D ⊆# D'›
‹remdups-mset D' = {#K#}› ‹¬tautology (D + D')› ‹undefined-lit M K›|

twl-subresolution-IL-nonunit:
‹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)›
if
‹clause C = add-mset L D›
‹clause C' = add-mset (−L) D'›
‹count-decided M = 0› ‹D ⊆# D'›
‹clause E = remdups-mset D'› ‹¬tautology (D + D')› ‹size (watched E) = 2› ‹∀ L ∈# clause E. undefined-lit M L› |
twl-subresolution-IL-unit:
‹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)›
if
‹clause C = add-mset L D›
‹clause C' = add-mset (−L) D'›
‹count-decided M = 0› ‹D ⊆# D'›
‹remdups-mset D' = {#K#}› ‹¬tautology (D + D')› ‹undefined-lit M K›

lemma *past-invs-count-decided0*: ‹count-decided (get-trail S) = 0 ⟹ past-invs S›
⟨*proof*⟩

lemma *cdcl-twl-subresolution-past-invs*: ‹cdcl-twl-subresolution S T ⟹ past-invs T›
⟨*proof*⟩

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) $C \Longrightarrow$
twl-lazy-update $(Propagated\ K\ \{\#K\#\}\ \#\ M)$ $C\rangle$
⟨*proof*⟩

lemma *watched-literals-false-of-max-level-prop-lvl0*: ‹*count-decided* $M = 0 \Longrightarrow$ *watched-literals-false-of-max-level* $(Propagated\ K\ \{\#K\#\}\ \#\ M)$ $C\rangle$
⟨*proof*⟩

lemma *watched-literals-false-of-max-level-lvl0*: ‹*count-decided* $M = 0 \Longrightarrow$ *watched-literals-false-of-max-level* (M) $C\rangle$
⟨*proof*⟩

lemma *twl-lazy-update-undefined*: ‹$\forall L \in\#$ *clause* E. *undefined-lit* $M\ L \Longrightarrow$ *twl-lazy-update* $M\ E\rangle$
⟨*proof*⟩

lemma *struct-wf-twl-cls-remdupsI*:
‹*clause* $E =$ *remdups-mset* $D' \Longrightarrow$ *size* $(watched\ E) = 2 \Longrightarrow$ *struct-wf-twl-cls* $E\rangle$
⟨*proof*⟩

lemma *cdcl-twl-subresolution-twl-st-inv*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$ *twl-st-inv* $S \Longrightarrow$ *twl-st-inv* $T\rangle$
⟨*proof*⟩

lemma *cdcl-twl-subresolution-valid-enqueued*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$ *valid-enqueued* $S \Longrightarrow$ *valid-enqueued* $T\rangle$
⟨*proof*⟩

lemma *cdcl-twl-subresolution-decompE*:
assumes
‹*cdcl-twl-subresolution* $S\ T\rangle$ **and** ‹*Multiset.Ball* $(get\text{-}clauses\ S)$ $(distinct\text{-}mset\ o\ clause)\rangle$ **and**
subres: ‹*cdcl-subresolution* $(pstate_W\text{-}of\ S)$ $(pstate_W\text{-}of\ T) \Longrightarrow thesis\rangle$ **and**
unit: ‹$\bigwedge S'\ T'$. *cdcl-subresolution* $(pstate_W\text{-}of\ S)\ S' \Longrightarrow$
cdcl-propagate $S'\ (T') \Longrightarrow$ *cdcl-flush-unit* (T') $(pstate_W\text{-}of\ T) \Longrightarrow thesis\rangle$
shows *thesis*
⟨*proof*⟩

lemma *cdcl-twl-subresolution-pcdcl-all-struct-invs*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$
pcdcl-all-struct-invs $(pstate_W\text{-}of\ S) \Longrightarrow$ *pcdcl-all-struct-invs* $(pstate_W\text{-}of\ T)\rangle$
⟨*proof*⟩

lemma *cdcl-twl-subresolution-smaller-propa*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$ $cdcl_W$*-restart-mset.no-smaller-propa* $(state_W\text{-}of\ T)\rangle$
⟨*proof*⟩
lemma *cdcl-twl-subresolution-twl-st-exception-inv*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$ *no-dup* $(get\text{-}trail\ S) \Longrightarrow$
twl-st-exception-inv $S \Longrightarrow$ *twl-st-exception-inv* $T\rangle$
⟨*proof*⟩

lemma *cdcl-twl-subresolution-dup-enqueued*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$ *no-duplicate-queued* $S \Longrightarrow$ *no-duplicate-queued* $T\rangle$
⟨*proof*⟩

lemma *cdcl-twl-subresolution-distinct-enqueued*:
‹*cdcl-twl-subresolution* $S\ T \Longrightarrow$ *distinct-queued* $S \Longrightarrow$ *no-duplicate-queued* $S \Longrightarrow$ *distinct-queued* $T\rangle$

⟨*proof*⟩

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*
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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')
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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′*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *cdcl-twl-subresolution-conflict*:
‹*cdcl-twl-subresolution S T* ⟹ *get-conflict T* = *None*›
⟨*proof*⟩

lemma *clause-alt-def*:
‹*clause C* = *watched C* + *unwatched C*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *cdcl-twl-subresolution-twl-struct-invs*:
assumes ‹*cdcl-twl-subresolution S T*›
‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
⟨*proof*⟩

lemma *Propagated-eq-DecidedD*:
‹*Propagated L C* # *M1* = *M* @ *Decided K* # *M′* ⟷
M ≠ [] ∧ *hd M* = *Propagated L C* ∧ *M1* = *tl M* @ *Decided K* # *M′*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-unitres-true-twl-struct-invs*:
‹*cdcl-twl-unitres-true S T* ⟹ *twl-struct-invs S* ⟹ *twl-struct-invs T*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*) $\models$*psm mset-set* (*CNot* C')›
‹$\neg$*tautology* C› ‹*distinct-mset* C›
‹*struct-wf-twl-cls* E›
‹*clause* $E = C$›
‹*Multiset.Ball* (*clause* E) (*undefined-lit* M)›
‹*atms-of* $C \subseteq$ *atms-of-ms* (*clause* ‘ *set-mset* N) $\cup$ *atms-of-mm NE* $\cup$ *atms-of-mm NS* $\cup$ *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* $\models$*psm mset-set* (*CNot* C')›
‹$\neg$*tautology* C› ‹*distinct-mset* C›
‹$C = \{\#K\#\}$›
‹*undefined-lit* M K›
‹*atms-of* $C \subseteq$ *atms-of-ms* (*clause* ‘ *set-mset* N) $\cup$ *atms-of-mm NE* $\cup$ *atms-of-mm NS* $\cup$ *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*) $\models$*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*) $\models$*psm mset-set* (*CNot* (*clause* D))›

lemma *cdcl-twl-unitres-cdcl-unitres*:
‹*cdcl-twl-unitres* S $T \Longrightarrow$ *cdcl-unitres* ($pstate_W$*-of* S) ($pstate_W$*-of* T)›
⟨*proof*⟩

lemma *cdcl-twl-unitres-invs*:
‹*cdcl-twl-unitres* S $T \Longrightarrow$ *twl-st-inv* $S \Longrightarrow$ *twl-st-inv* T›
⟨*proof*⟩

lemma *struct-wf-twl-cls-alt-def*:
‹*struct-wf-twl-cls* $C \longleftrightarrow$ *distinct-mset* (*clause* C) $\wedge$ *size* (*watched* C) = 2›
⟨*proof*⟩

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›
⟨*proof*⟩

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›

⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-subsumed-RI-all-struct-invs*:
‹*cdcl-subsumed-RI S T* ⟹ *pcdcl-all-struct-invs S* ⟹ *pcdcl-all-struct-invs T*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-subsumed-struct-invs*:
assumes ‹*cdcl-twl-subsumed S T*› **and** ‹*twl-struct-invs S*›
shows ‹*twl-struct-invs T*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *cdcl-twl-subsumed-twl-stgy-invs*:
assumes ‹*cdcl-twl-subsumed S T*›
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*›
  ⟨*proof*⟩

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''*›

$\langle UE'' \subseteq\# \ UE + clauses\ UE' \rangle$ |
restart-clauses:
$\langle cdcl\text{-}twl\text{-}restart\ (M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$
$(M,\ N',\ U',\ None,\ NE + clauses\ NE',\ UE'',\ NS,\ US',\ N0,\ \{\#\},\ \{\#\},\ Q)\rangle$
if
$\langle U' + UE' \subseteq\# \ U\rangle$ **and**
$\langle N = N' + NE'\rangle$ **and**
$\langle \forall\, E{\in}\#NE'{+}UE'.\ \exists\, L{\in}\#clause\ E.\ L \in lits\text{-}of\text{-}l\ M \wedge get\text{-}level\ M\ L = 0\rangle$
$\langle \forall\, L\ E.\ Propagated\ L\ E \in set\ M \longrightarrow E \in\# \ clause\ `\#\ (N + U') + (NE{+}\ clauses\ NE') + UE''\rangle$
$\langle US' = \{\#\}\rangle$
$\langle UE'' \subseteq\# \ UE + clauses\ UE'\rangle$

inductive-cases *cdcl-twl-restartE*: $\langle cdcl\text{-}twl\text{-}restart\ S\ T\rangle$

inductive *cdcl-twl-restart-only* :: $\langle 'v\ twl\text{-}st \Rightarrow 'v\ twl\text{-}st \Rightarrow bool\rangle$ **where**
restart-trail:
$\langle cdcl\text{-}twl\text{-}restart\text{-}only\ (M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$
$(M',\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ \{\#\})\rangle$
if
$\langle (Decided\ K\ \#\ M',\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)\rangle$ |
norestart-trail:
$\langle cdcl\text{-}twl\text{-}restart\text{-}only\ (M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$
$(M,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)\rangle$

lemma *past-invs-simps-emptyU0*:
$\langle past\text{-}invs\ (M,\ N,\ U,\ D,\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ WS,\ Q) \longleftrightarrow$
$(\forall\, M1\ M2\ K.\ M = M2\ @\ Decided\ K\ \#\ M1 \longrightarrow ($
$(\forall\, C \in\# \ N + U.\ twl\text{-}lazy\text{-}update\ M1\ C \wedge$
$watched\text{-}literals\text{-}false\text{-}of\text{-}max\text{-}level\ M1\ C \wedge$
$twl\text{-}exception\text{-}inv\ (M1,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ \{\#\},\ \{\#\},\ \{\#\})\ C) \wedge$
$conft\text{-}cands\text{-}enqueued\ (M1,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ \{\#\},\ \{\#\},\ \{\#\}) \wedge$
$propa\text{-}cands\text{-}enqueued\ (M1,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ \{\#\},\ \{\#\},\ \{\#\}) \wedge$
$clauses\text{-}to\text{-}update\text{-}inv\ (M1,\ N,\ U,\ None,\ NE,\ UE,\ NS,\ US,\ N0,\ \{\#\},\ \{\#\},\ \{\#\})))\rangle$
⟨*proof*⟩

lemma *cdcl-twl-restart-pcdcl*: $\langle cdcl\text{-}twl\text{-}restart\ S\ T \Longrightarrow pcdcl\text{-}restart\ (pstate_W\text{-}of\ S)\ (pstate_W\text{-}of\ T)\rangle$
⟨*proof*⟩

lemma *cdcl-twl-restart-twl-struct-invs*:
assumes
$\langle cdcl\text{-}twl\text{-}restart\ S\ T\rangle$ **and**
$\langle twl\text{-}struct\text{-}invs\ S\rangle$
shows $\langle twl\text{-}struct\text{-}invs\ T\rangle$
⟨*proof*⟩

lemma *in-multiset-subsetD*:
$\langle A \in\# \ B \Longrightarrow B \subseteq\# \ C + D \Longrightarrow A \in\# \ C \vee A \in\# \ D\rangle$
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-restart-twl-struct-invs*:
assumes
$\langle cdcl\text{-}twl\text{-}restart^{**}\ S\ T\rangle$ **and**
$\langle twl\text{-}struct\text{-}invs\ S\rangle$
shows $\langle twl\text{-}struct\text{-}invs\ T\rangle$

⟨*proof*⟩

lemma *cdcl-twl-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-restart S T*› **and** ‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-restart*** *S T*› **and**
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
⟨*proof*⟩

lemma *cdcl-twl-restart-only-cdcl*: ‹*cdcl-twl-restart-only T U* ⟹
pcdcl-restart-only (*pstate*$_W$ *-of T*) (*pstate*$_W$ *-of U*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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* ⟹ *A* ⊨*ps set-mset UE* ⟹ *A* ⊨*ps set-mset UE'* ⟹ *A* ⊨*ps set-mset UE''*›
‹*UE* + *UE'* = *ca* + *UE''* ⟹ *A* ⊨*ps set-mset UE* ⟹ *A* ⊨*ps set-mset UE'* ⟹ *A* ⊨*ps set-mset UE''*›
for *UE* :: ‹*'v clauses*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *rtranclp-pcdcl-restart-inprocessing*: ‹*pcdcl*** *S T* ⟹ *pcdcl-inprocessing*** *S T*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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*)›

⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *pcdcl-twl-final-state-pcdcl*:
‹*pcdcl-twl-final-state S* ⟹
twl-struct-invs S ⟹ *pcdcl-final-state* (*pstate$_W$-of S*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*))))›
⟨*proof*⟩

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*))))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›

⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-twl-restart-inv*:
‹*cdcl-twl-stgy-restart S T* ⟹ *twl-restart-inv S* ⟹ *twl-restart-inv T*›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-stgy-restart-twl-restart-inv*:
‹*cdcl-twl-stgy-restart*** *S T* ⟹ *twl-restart-inv S* ⟹ *twl-restart-inv T*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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›
⟨*proof*⟩

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›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cdcl$_W$-init-state*:
‹*cdcl$_W$-restart-mset.cdcl$_W$*** (*init-state* {#}) *S* ⟷ *S* = *init-state* {#}›
⟨*proof*⟩

lemma *rtranclp-pcdcl-core-is-cdcl*:
‹*pcdcl-core*** *S T* ⟹ *cdcl$_W$-restart-mset.cdcl$_W$*** (*state-of S*) (*state-of T*)›
⟨*proof*⟩

lemma *pcdcl-tcore-is-cdclD*:
‹*pcdcl-tcore T T′* ⟹
*cdcl$_W$-restart-mset.cdcl$_W$-restart*** (*state-of T*) (*state-of T′*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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›
⟨*proof*⟩

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›
⟨*proof*⟩

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)›
⟨*proof*⟩

lemma *pcdcl-core-entailed-iff*:
‹*pcdcl-core* S T ⟹ M ⊨*m* *pget-all-init-clss* T ⟷ M ⊨*m* *pget-all-init-clss* S›
⟨*proof*⟩

lemma *pcdcl-entailed-iff*:
‹*pcdcl* S T ⟹ *consistent-interp* M ⟹
total-over-set M (*atms-of-mm* (*pget-all-init-clss* T)) ⟹
$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init (state-of S) ⟹
M ⊨*m* *pget-all-init-clss* T ⟹ M ⊨*m* *pget-all-init-clss* S›
⟨*proof*⟩

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))›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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*›)
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *pcdcl-restart-entailed-iff*:
‹*pcdcl-restart S T* ⟹ *M* ⊨*m pget-all-init-clss T* ⟷ *M* ⊨*m pget-all-init-clss S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*simp*]: ‹*pget-all-init-clss* ($pstate_W$*-of S*) = (*get-all-init-clss S*)›
⟨*proof*⟩

lemma *empty-entails-novars-iff*: ‹*atms-of-mm S* = {} ⟹ {} ⊨*ps set-mset S* ⟷ *S* = {#}›
⟨*proof*⟩
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*›
⟨*proof*⟩

lemma *mset-take-mset-drop-mset'*: ‹*mset* (*take* 2 *x*) + *mset* (*drop* 2 *x*) = *mset x*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

definition *set-conflict'* :: ‹'v clause option ⇒ 'v twl-st ⇒ 'v twl-st› **where**
‹set-conflict' = (λC (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›
⟨proof⟩

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 = []›
⟨proof⟩

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›
⟨proof⟩

lemma *take-convert-lits-lD*:
‹(M, M') ∈ convert-lits-l N E ⟹
(take n M, take n M') ∈ convert-lits-l N E›
⟨proof⟩

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›
⟨proof⟩

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 ›
⟨proof⟩

lemma *convert-lits-l-map-lit-of*: ‹(ay, bq) ∈ convert-lits-l N e ⟹ map lit-of ay = map lit-of bq›
⟨proof⟩

lemma *convert-lits-l-tlD*:
‹(M, M') ∈ convert-lits-l N E ⟹
(tl M, tl M') ∈ convert-lits-l N E›
⟨proof⟩

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›
⟨proof⟩

lemma *get-clauses-l-set-clauses-to-update-l*[*simp*]:
‹*get-clauses-l* (*set-clauses-to-update-l WC S*) = *get-clauses-l S*›
⟨*proof*⟩

lemma *get-trail-l-set-clauses-to-update-l*[*simp*]:
‹*get-trail-l* (*set-clauses-to-update-l WC S*) = *get-trail-l S*›
⟨*proof*⟩

lemma *get-trail-set-clauses-to-update*[*simp*]:
‹*get-trail* (*set-clauses-to-update WC S*) = *get-trail S*›
⟨*proof*⟩

lemma *convert-lits-l-add-mset*:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
(*M*, *M'*) ∈ *convert-lits-l N* (*add-mset C E*)›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *resolve-cls-l-nil-iff*:
‹*resolve-cls-l L D' E* = [] ⟷ $cdcl_W$*-restart-mset.resolve-cls L* (*mset D'*) (*mset E*) = {#}›
⟨*proof*⟩

lemma *lit-of-convert-lit*[*simp*]:
‹*convert-lit N E L L'* ⟹ *lit-of L'* = *lit-of L*›
⟨*proof*⟩

lemma *is-decided-convert-lit*[*simp*]:
‹*convert-lit N E L L'* ⟹ *is-decided L'* ⟷ *is-decided L*›
⟨*proof*⟩

lemma *defined-lit-convert-lits-l*[*simp*]: ‹$(M,\ M') \in$ *convert-lits-l N E* $\Longrightarrow$
defined-lit M′ = *defined-lit M*›
⟨*proof*⟩

lemma *no-dup-convert-lits-l*[*simp*]: ‹$(M,\ M') \in$ *convert-lits-l N E* $\Longrightarrow$
no-dup M′ $\longleftrightarrow$ *no-dup M*›
⟨*proof*⟩

lemma
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*›
shows
count-decided-convert-lits-l[*simp*]:
‹*count-decided M′* = *count-decided M*›
⟨*proof*⟩

lemma
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*›
shows
get-level-convert-lits-l[*simp*]:
‹*get-level M′* = *get-level M*›
⟨*proof*⟩

lemma
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*›
shows
get-maximum-level-convert-lits-l[*simp*]:
‹*get-maximum-level M′* = *get-maximum-level M*›
⟨*proof*⟩

lemma *list-of-l-convert-map-lit-of*:
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*›
shows
‹*map lit-of M′* = *map lit-of M*›
⟨*proof*⟩

lemma *list-of-l-convert-lits-l*[*simp*]:
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*›
shows
‹*lits-of-l M′* = *lits-of-l M*›
⟨*proof*⟩

lemma *is-proped-hd-convert-lits-l*[*simp*]:
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*› **and** ‹$M \neq []$›
shows ‹*is-proped* (*hd M′*) $\longleftrightarrow$ *is-proped* (*hd M*)›
⟨*proof*⟩

lemma *is-decided-hd-convert-lits-l*[*simp*]:
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*› **and** ‹$M \neq []$›
shows
‹*is-decided* (*hd M′*) $\longleftrightarrow$ *is-decided* (*hd M*)›
⟨*proof*⟩

lemma *lit-of-hd-convert-lits-l*[*simp*]:
assumes ‹$(M,\ M') \in$ *convert-lits-l N E*› **and** ‹$M \neq []$›
shows
‹*lit-of* (*hd M′*) = *lit-of* (*hd M*)›

⟨*proof*⟩

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*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

lemma *convert-lits-l-nil-iff*[*simp*]:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows
‹*M'* = [] ⟷ *M* = []›
⟨*proof*⟩

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'*›
⟨*proof*⟩

lemma *convert-lits-l-true-clss-clss*[*simp*]:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹ *M'* ⊨*as C* ⟷ *M* ⊨*as C*›
⟨*proof*⟩

lemma *convert-lit-propagated-decided*[*iff*]:
‹*convert-lit b d* (*Propagated x21 x22*) (*Decided x1*) ⟷ *False*›
⟨*proof*⟩

lemma *convert-lit-decided*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Decided x2*) ⟷ *x1* = *x2*›
⟨*proof*⟩

lemma *convert-lit-decided-propagated*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Propagated x21 x22*) ⟷ *False*›
⟨*proof*⟩

lemma *convert-lits-l-lit-of-mset*[*simp*]:
‹(*a*, *af*) ∈ *convert-lits-l N E* ⟹ *lit-of* ‘# *mset af* = *lit-of* ‘# *mset a*›
⟨*proof*⟩

lemma *convert-lits-l-imp-same-length*:
‹(*a*, *b*) ∈ *convert-lits-l N E* ⟹ *length a* = *length b*›
⟨*proof*⟩

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*⟩

lemma *in-convert-lits-lD*:
‹*K* ∈ *set TM* ⟹
(*M*, *TM*) ∈ *convert-lits-l N NE* ⟹
∃ *K'*. *K'* ∈ *set M* ∧ *convert-lit N NE K' K*›
⟨*proof*⟩

lemma *in-convert-lits-lD2*:
‹*K* ∈ *set M* ⟹
(*M*, *TM*) ∈ *convert-lits-l N NE* ⟹
∃ *K'*. *K'* ∈ *set TM* ∧ *convert-lit N NE K K'*›
⟨*proof*⟩

lemma *convert-lits-l-filter-decided*: ‹(*S*, *S'*) ∈ *convert-lits-l M N* ⟹
map lit-of (*filter is-decided S'*) = *map lit-of* (*filter is-decided S*)›
⟨*proof*⟩

lemma *convert-lits-lI*:
‹*length M* = *length M'* ⟹ (⋀*i*. *i* < *length M* ⟹ *convert-lit N NE* (*M*!*i*) (*M'*!*i*)) ⟹
(*M*, *M'*) ∈ *convert-lits-l N NE*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*) ∈ *convert-lits-l N* (*NEk*+*UEk*) ∧
N' = *twl-clause-of* ‘# *init-clss-lf N* ∧
U' = *twl-clause-of* ‘# *learned-clss-lf N* ∧
C' = *C* ∧
NE' = *NE*+*NEk* ∧
UE' = *UE* + *UEk* ∧
NS' = *NS* ∧
US' = *US* ∧
N0' = *N0* ∧
U0' = *U0* ∧
WS' = (*case L of None* ⇒ {#} | *Some L* ⇒ *image-mset* (λ*j*. (*L*, *twl-clause-of* (*N* ∝ *j*))) *WS*) ∧
Q' = *Q*
}›

lemma *clss-state*$_W$*-of*[*twl-st*]:

assumes ‹$(S,\ R) \in$ *twl-st-l* L›
shows
‹*init-clss* ($state_W$*-of* R) = *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›
‹*learned-clss* ($state_W$*-of* R) = *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›
⟨*proof*⟩

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

lemma [*twl-st-l*]:
assumes ‹$(S,\ T) \in$ *twl-st-l* L›
shows
‹(*get-trail-l* S, *get-trail* T) ∈ *convert-lits-l* (*get-clauses-l* S) (*get-kept-unit-clauses-l* S)› **and**
‹*get-clauses* T = *twl-clause-of* ‘# *fst* ‘# *ran-m* (*get-clauses-l* S)› **and**
‹*get-conflict* T = *get-conflict-l* S› **and**
‹L = *None* ⟹ *clauses-to-update* T = {#}›
‹$L \neq$ *None* ⟹ *clauses-to-update* T =
(λ*j*. (*the* L, *twl-clause-of* (*get-clauses-l* S ∝ *j*))) ‘# *clauses-to-update-l* S› **and**
‹*literals-to-update* T = *literals-to-update-l* S›
‹*backtrack-lvl* ($state_W$*-of* T) = *count-decided* (*get-trail-l* S)›
‹*unit-clss* T = *get-unit-clauses-l* S›
‹$cdcl_W$*-restart-mset.clauses* ($state_W$*-of* T) =
mset ‘# *ran-mf* (*get-clauses-l* S) + *get-unit-clauses-l* S + *get-subsumed-clauses-l* S +
get-clauses0-l S› **and**
‹*no-dup* (*get-trail* T) ⟷ *no-dup* (*get-trail-l* S)› **and**
‹*lits-of-l* (*get-trail* T) = *lits-of-l* (*get-trail-l* S)› **and**
‹*count-decided* (*get-trail* T) = *count-decided* (*get-trail-l* S)› **and**
‹*get-trail* T = [] ⟷ *get-trail-l* S = []› **and**
‹*get-trail* $T \neq$ [] ⟷ *get-trail-l* $S \neq$ []› **and**
‹*get-trail* $T \neq$ [] ⟹ *is-proped* (*hd* (*get-trail* T)) ⟷ *is-proped* (*hd* (*get-trail-l* S))›
‹*get-trail* $T \neq$ [] ⟹ *is-decided* (*hd* (*get-trail* T)) ⟷ *is-decided* (*hd* (*get-trail-l* S))›
‹*get-trail* $T \neq$ [] ⟹ *lit-of* (*hd* (*get-trail* T)) = *lit-of* (*hd* (*get-trail-l* S))›
‹*get-level* (*get-trail* T) = *get-level* (*get-trail-l* S)›
‹*get-maximum-level* (*get-trail* T) = *get-maximum-level* (*get-trail-l* S)›
‹*get-trail* T ⊨*as* D ⟷ *get-trail-l* S ⊨*as* D›
‹*subsumed-init-clauses* T = *get-subsumed-init-clauses-l* S›
‹*subsumed-learned-clauses* T = *get-subsumed-learned-clauses-l* S›
‹*subsumed-clauses* T = *get-subsumed-clauses-l* S›
‹*get-all-clss* T = *get-all-clss-l* S›
‹*all-lits-of-st* T = *all-lits-of-st-l* S›
⟨*proof*⟩

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›
⟨*proof*⟩

lemma [*twl-st-l*]:
assumes ‹$(S,\ T) \in$ *twl-st-l* L›
shows ‹*lit-of* ‘ *set* (*get-trail* T) = *lit-of* ‘ *set* (*get-trail-l* S)›

⟨*proof*⟩

lemma [*twl-st-l*]:
‹*get-trail-l* (*set-literals-to-update-l D S*) = *get-trail-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]: ‹(*S*, *T*) ∈ *twl-st-l L* ⟹ *defined-lit* (*get-trail T*) = *defined-lit* (*get-trail-l S*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*twl-st-l*]: ‹*get-conflict-l* (*remove-one-lit-from-wq L S*) = *get-conflict-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]: ‹*literals-to-update-l* (*set-clauses-to-update-l Cs S*) = *literals-to-update-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]: ‹*get-unit-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-clauses-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]: ‹*get-unit-clauses-l* (*remove-one-lit-from-wq L S*) = *get-unit-clauses-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹*get-unit-init-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-init-clauses-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹*get-unit-init-clauses-l* (*remove-one-lit-from-wq L S*) = *get-unit-init-clauses-l S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹*get-unit-learned-clss-l* (*set-clauses-to-update-l Cs S*) = *get-unit-learned-clss-l S*›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹*get-unit-learned-clss-l* (*remove-one-lit-from-wq L S*) = *get-unit-learned-clss-l S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'* ∧ *N* = *N'* ∧ *NE* = *NE'* ∧ *UE* = *UE'* ∧ *NEk* = *NEk'* ∧ *UEk* = *UEk'* ∧ *NS* = *NS'* ∧ *US* = *US'* ∧ *WS* = *WS'* ∧ *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*›
⟨*proof*⟩

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*›

⟨*proof*⟩

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›
⟨*proof*⟩

lemma *all-lits-of-st-simps*[*simp*]:
‹*all-lits-of-st* (*set-clauses-to-update C S*)= *all-lits-of-st S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *polarity-spec*:
assumes ‹*no-dup M*›
shows
‹*RETURN* (*polarity M L*) ≤ *SPEC*(λ*v*. (*v* = *None* ⟷ *undefined-lit M L*) ∧
(*v* = *Some True* ⟷ *L* ∈ *lits-of-l M*) ∧ (*v* = *Some False* ⟷ −*L* ∈ *lits-of-l M*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)) ∧ *distinct* (*get-clauses-l S* ∝ *C*) ∧
get-trail-l S ⊨*as CNot* (*mset* (*get-clauses-l S* ∝ *C*)) ∧ *twl-list-invs S* ∧
(∃ *S'* *b*. (*set-clauses-to-update-l* (*clauses-to-update-l S* + {#*C*#}) *S*, *S'*) ∈ *twl-st-l b* ∧ *twl-struct-invs S'* ∧ *twl-stgy-invs S'*)›

definition *set-conflict-l* :: ‹*nat* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*set-conflict-l* = (λ*C* (*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* ∝ *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* ⇒ *nat* ⇒ *nat* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*propagate-lit-l* = (λ*L'* *C* *i* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*). *do* {
ASSERT(*C* ∈# *dom-m N*);
ASSERT(*L'* ∈# *all-lits-of-st-l* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *WS*, *Q*));
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*, *WS*, *add-mset* (−*L'*) *Q*)})›

definition *update-clause-l-pre* :: ‹*nat* ⇒ *nat* ⇒ *nat* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
‹*update-clause-l-pre C i f S* ⟷
(∃ *S'* *L*. (*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *C* ∈# *dom-m* (*get-clauses-l S*) ∧
i < *length* (*get-clauses-l S* ∝ *C*) ∧ *f* < *length* (*get-clauses-l S* ∝ *C*) ∧

update-clauseS-pre (*get-clauses-l S* ∝ *C* ! *i*) (*twl-clause-of* (*get-clauses-l S* ∝ *C*)) *S'*⟩

definition *update-clause-l* :: ⟨*nat* ⇒ *nat* ⇒ *nat* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*⟩ **where**
⟨*update-clause-l* = (λ*C 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' ← *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* ⇒ *nat* ⇒ *'v twl-st-l* ⇒ *bool*⟩
where
⟨*unit-propagation-inner-loop-body-l-inv L C S* ⟷
(∃ *S'*. (*set-clauses-to-update-l* (*clauses-to-update-l S* + {#*C*#}) *S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧
twl-struct-invs S' ∧
twl-stgy-invs S' ∧
C ∈# *dom-m* (*get-clauses-l S*) ∧
C > *0* ∧
0 < *length* (*get-clauses-l S* ∝ *C*) ∧
no-dup (*get-trail-l S*) ∧
(*if* (*get-clauses-l S* ∝ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* ∝ *C*) ∧
1 − (*if* (*get-clauses-l S* ∝ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* ∝ *C*) ∧
L ∈ *set* (*watched-l* (*get-clauses-l S* ∝ *C*)) ∧
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*(λ*K*. *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*(λ*S'*. *Q S'*)› **and**
‹*y* ≤ ⇓ {(*S*, *S'*). *P S S'*} (*f S*)›
shows ‹*y* ≤ ⇓ {(*S*, *S'*). *P S S'* ∧ *Q S'*} (*f S*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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 (λK. K ∈# clause C);
    val-bL' ← mop-lit-is-pos bL' S;
    if val-bL'
    then RETURN S
    else do {
      i ← RETURN ();
      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
      }
    }
  }›
⟨proof⟩

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)#}›
  ⟨proof⟩

lemma in-set-takeI:
  ‹i < j ⟹ i < length xs ⟹ xs ! i ∈ set (take j xs)›
  ⟨proof⟩

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⟩

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*›
⟨*proof*⟩

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*
⟨*proof*⟩

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*›
⟨*proof*⟩

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 ‹$(S,\ S') \in twl\text{-}st\text{-}l\ (Some\ L)$›
shows
‹$RES\ (set\text{-}mset\ (clauses\text{-}to\text{-}update\text{-}l\ S)) \leq \Downarrow \{(C,\ (L',\ C')).\ L' = L \wedge$
$C' = twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C)\}$
$(RES\ (set\text{-}mset\ (clauses\text{-}to\text{-}update\ S')))$›
⟨*proof*⟩

lemma *clauses-to-update-l-empty-tw-st-of-Some-None*[*simp*]:
‹$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\} \Longrightarrow (S,\ S') \in twl\text{-}st\text{-}l\ (Some\ L) \longleftrightarrow (S,\ S') \in twl\text{-}st\text{-}l\ None$›
⟨*proof*⟩

lemma *cdcl-twl-cp-in-trail-stays-in*:
‹$cdcl\text{-}twl\text{-}cp^{**}\ S'\ aa \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\ S') \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\ aa)$›
⟨*proof*⟩

lemma *cdcl-twl-cp-in-trail-stays-in-l*:
‹$(x2,\ S') \in twl\text{-}st\text{-}l\ (Some\ x1) \Longrightarrow cdcl\text{-}twl\text{-}cp^{**}\ S'\ aa \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ x2) \Longrightarrow$
$(a,\ aa) \in twl\text{-}st\text{-}l\ (Some\ x1) \Longrightarrow\ -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ a)$›
⟨*proof*⟩

lemma *unit-propagation-inner-loop-l*:
‹$(uncurry\ unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l,\ unit\text{-}propagation\text{-}inner\text{-}loop) \in$
$\{((L,\ S),\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}list\text{-}invs\ S \wedge -L \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ S) \wedge$
$L\ \in\#\ all\text{-}lits\text{-}of\text{-}st\text{-}l\ S\} \rightarrow_f$
$\langle\{(T,\ T').\ (T,\ T') \in twl\text{-}st\text{-}l\ None \wedge clauses\text{-}to\text{-}update\text{-}l\ T = \{\#\} \wedge$
$twl\text{-}list\text{-}invs\ T\}\rangle\ nres\text{-}rel$›
(**is** ‹$?unit\text{-}prop\text{-}inner \in ?A \rightarrow_f \langle ?B\rangle nres\text{-}rel$›)
⟨*proof*⟩

definition *clause-to-update* :: ‹$'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}l \Rightarrow 'v\ clauses\text{-}to\text{-}update\text{-}l$›**where**
‹$clause\text{-}to\text{-}update\ L\ S =$
$filter\text{-}mset$
$(\lambda C{::}nat.\ L \in set\ (watched\text{-}l\ (get\text{-}clauses\text{-}l\ S \propto C)))$
$(dom\text{-}m\ (get\text{-}clauses\text{-}l\ S))$›

lemma *distinct-mset-clause-to-update*: ‹$distinct\text{-}mset\ (clause\text{-}to\text{-}update\ L\ C)$›
⟨*proof*⟩

lemma *in-clause-to-updateD*: ‹$b\ \in\#\ clause\text{-}to\text{-}update\ L'\ T \Longrightarrow b\ \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}l\ T)$›
⟨*proof*⟩

lemma *in-clause-to-update-iff*:
‹$C\ \in\#\ clause\text{-}to\text{-}update\ L\ S \longleftrightarrow$
$C\ \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}l\ S) \wedge L \in set\ (watched\text{-}l\ (get\text{-}clauses\text{-}l\ S \propto C))$›
⟨*proof*⟩

definition *select-and-remove-from-literals-to-update* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow$
$('v\ twl\text{-}st\text{-}l \times 'v\ literal)\ nres$› **where**
‹$select\text{-}and\text{-}remove\text{-}from\text{-}literals\text{-}to\text{-}update\ S = SPEC(\lambda(S',\ L).\ L \in\#\ literals\text{-}to\text{-}update\text{-}l\ S \wedge$
$S' = set\text{-}clauses\text{-}to\text{-}update\text{-}l\ (clause\text{-}to\text{-}update\ L\ S)$
$(set\text{-}literals\text{-}to\text{-}update\text{-}l\ (literals\text{-}to\text{-}update\text{-}l\ S - \{\#L\#\})\ S))$›

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *get-conflict-l-get-conflict-state-spec*:
assumes ‹(*S*, *S'*) ∈ *twl-st-l None*› **and** ‹*twl-list-invs S*› **and** ‹*clauses-to-update-l S* = {#}›

shows ‹((*False*, *S*), (*False*, *S′*))
∈ {((*brk*, *S*), (*brk′*, *S′*)). *brk* = *brk′* ∧ (*S*, *S′*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}}›
⟨*proof*⟩

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* = (λ(*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*) ∪# *mset* (*get-clauses-l S* ∝ *C*)))›

definition *update-confl-tl-l* :: ‹*'v literal* ⇒ *nat* ⇒ *'v twl-st-l* ⇒ *bool* × *'v twl-st-l*› **where**
‹*update-confl-tl-l* = (λ*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* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *twl-st-l None* ∧ *update-confl-tl-pre L* (*mset* (*get-clauses-l S* ∝ *C*)) *S′* ∧
twl-list-invs S ∧ *C* ∈# *dom-m* (*get-clauses-l S*) ∧ *get-trail-l S* ≠ [] ∧ *hd* (*get-trail-l S*) = *Propagated L C* ∧ *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* = (λ*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* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *twl-st-l None* ∧ *mop-hd-trail-pre S′* ∧
twl-list-invs S ∧ *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(λ(*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* ≠ *None* ∧ −*L* ∉# *the* (*get-conflict-l S*) ∧ *L* ∈# *all-lits-of-st-l S*);
RETURN (*L* ∉# *the* (*get-conflict-l S*))
}›

definition *mop-tl-state-l-pre* :: ‹*'v twl-st-l* ⇒ *bool*› **where**
‹*mop-tl-state-l-pre S* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *twl-st-l None* ∧ *mop-tl-state-pre S′* ∧
twl-list-invs S)›

definition *mop-tl-state-l* :: ‹*'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres*› **where**
‹*mop-tl-state-l* = (λ*S*. *do* {
ASSERT(*mop-tl-state-l-pre S*);
RETURN(*False*, *tl-state-l S*)})›

definition *mop-maximum-level-removed-l-pre* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*mop-maximum-level-removed-l-pre L S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *mop-maximum-level-removed-pre L S'* ∧
twl-list-invs S)›

definition *mop-maximum-level-removed-l* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *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* ⟷
(∃ *S'* *S*$_0$'. (*S*, *S'*) ∈ *twl-st-l None* ∧ (*S*$_0$, *S*$_0$') ∈ *twl-st-l None* ∧
skip-and-resolve-loop-inv S$_0$' (*brk*, *S'*) ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#} ∧
(¬*is-decided* (*hd* (*get-trail-l S*)) ⟶ *mark-of* (*hd*(*get-trail-l S*)) > *0*))›

definition *skip-and-resolve-loop-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*skip-and-resolve-loop-l S*$_0$ =
do {
ASSERT(*get-conflict-l S*$_0$ ≠ *None*);
(-, *S*) ←
WHILE$_T$$^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\text{-}l\ S_0\ brk\ S}$
(λ(*brk*, *S*). ¬*brk* ∧ ¬*is-decided* (*hd* (*get-trail-l S*)))
(λ(-, *S*).
do {
(*L*, *C*) ← *mop-hd-trail-l S*;
b ← *mop-lit-notin-conflict-l* (−*L*) *S*;
if b then
mop-tl-state-l S
else do {
b ← *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*⟩

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*⟩

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*›
⟨*proof*⟩

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*⟩

definition *find-decomp* :: ‹′*v literal* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l nres*› **where**
‹*find-decomp* = (λ*L* (*M*, *N*, *D*, *NE*, *NEk*, *UEk*, *UE*, *NS*, *US*, *WS*, *Q*).
SPEC(λ*S*. ∃ *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(λ*T*. ∃ *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)›
⟨*proof*⟩

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*#})))›
⟨*proof*⟩

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'*) }›
⟨*proof*⟩

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*(λ*i*. *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* = (λ*L 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* = (λ*L* (*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(λ*L*. *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⟩

definition find-unassigned-lit-l :: ‹'v twl-st-l ⇒ 'v literal option nres› where
  ‹find-unassigned-lit-l = (λS.
    SPEC (λL.
        (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 = (λL' (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*⟩

lemma *refinement-trans-eq*:
‹*A* = *A'* ⟹ *B* = *B'* ⟹ *R'* = *R* ⟹ *A* ≤ ⇓ *R B* ⟹ *A'* ≤ ⇓ *R' B'*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *weaken-⇓'*: ‹*f* ≤ ⇓ *R' g* ⟹ *R'* ⊆ *R* ⟹ *f* ≤ ⇓ *R g*›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

definition *cdcl-twl-stgy-prog-l-pre* **where**
  ‹*cdcl-twl-stgy-prog-l-pre S S'* ⟷

$((S,\ S') \in \textit{twl-st-l None} \wedge \textit{twl-struct-invs } S' \wedge \textit{twl-stgy-invs } S' \wedge$
$\textit{clauses-to-update-l } S = \{\#\} \wedge \textit{get-conflict-l } S = \textit{None} \wedge \textit{twl-list-invs } S \wedge$
$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-learned-clauses-entailed-by-init } (\textit{state}_W\textit{-of } S'))\rangle$

lemma *cdcl-twl-stgy-prog-l-spec-final*:
assumes
$\langle \textit{cdcl-twl-stgy-prog-l-pre } S\ S' \rangle$
shows
$\langle \textit{cdcl-twl-stgy-prog-l } S \leq \Downarrow (\textit{twl-st-l None})\ (\textit{conclusive-TWL-norestart-run } S') \rangle$
⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-l-spec-final'*:
assumes
$\langle \textit{cdcl-twl-stgy-prog-l-pre } S\ S' \rangle$
shows
$\langle \textit{cdcl-twl-stgy-prog-l } S \leq \Downarrow \{(S,\ T).\ (S,\ T) \in \textit{twl-st-l None} \wedge \textit{twl-list-invs } S \wedge$
$\textit{twl-struct-invs } S' \wedge \textit{twl-stgy-invs } S'\}\ (\textit{conclusive-TWL-norestart-run } S') \rangle$
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-break-l* :: $\langle 'v\ \textit{twl-st-l} \Rightarrow 'v\ \textit{twl-st-l nres} \rangle$ **where**
$\langle \textit{cdcl-twl-stgy-prog-break-l } S_0 =$
do {
$b \leftarrow \textit{SPEC}(\lambda\text{-}.\ \textit{True})$;
$(b,\ \textit{brk},\ T) \leftarrow \textit{WHILE}_T \lambda(b,\ S).\ \textit{cdcl-twl-stgy-prog-l-inv } S_0\ S$
$(\lambda(b,\ \textit{brk},\ \text{-}).\ b \wedge \neg \textit{brk})$
$(\lambda(\text{-},\ \textit{brk},\ S).\ \textit{do}$ {
$T \leftarrow \textit{unit-propagation-outer-loop-l } S$;
$T \leftarrow \textit{cdcl-twl-o-prog-l } T$;
$b \leftarrow \textit{SPEC}(\lambda\text{-}.\ \textit{True})$;
$\textit{RETURN } (b,\ T)$
})
$(b,\ \textit{False},\ S_0)$;
if brk then RETURN T
else cdcl-twl-stgy-prog-l T
}$\rangle$

lemma *cdcl-twl-stgy-prog-break-l-spec*:
$\langle (\textit{cdcl-twl-stgy-prog-break-l},\ \textit{cdcl-twl-stgy-prog-break}) \in$
$\{(S,\ S').\ (S,\ S') \in \textit{twl-st-l None} \wedge \textit{twl-list-invs } S \wedge \textit{clauses-to-update-l } S = \{\#\}\} \rightarrow_f$
$\langle \{(T,\ T').\ (T,\ T') \in \{(T,\ T').\ (T,\ T') \in \textit{twl-st-l None} \wedge \textit{twl-list-invs } T\} \wedge \textit{True}\} \rangle \textit{nres-rel} \rangle$
(**is** $\langle$ - $\in$ *?R* $\rightarrow_f$ *?I* $\rangle$ **is** $\langle$ - $\in$ *?R* $\rightarrow_f$ $\langle$*?J*$\rangle$*nres-rel*$\rangle$)
⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-break-l-spec-final*:
assumes
$\langle \textit{cdcl-twl-stgy-prog-l-pre } S\ S' \rangle$
shows
$\langle \textit{cdcl-twl-stgy-prog-break-l } S \leq \Downarrow (\textit{twl-st-l None})\ (\textit{conclusive-TWL-norestart-run } S') \rangle$
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-early-l* :: $\langle 'v\ \textit{twl-st-l} \Rightarrow (\textit{bool} \times 'v\ \textit{twl-st-l})\ \textit{nres} \rangle$ **where**
$\langle \textit{cdcl-twl-stgy-prog-early-l } S_0 =$
do {
$b \leftarrow \textit{SPEC}(\lambda\text{-}.\ \textit{True})$;

```
    (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⟩

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⟩

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')›
  ⟨proof⟩

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⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma (**in** −) *wf-trancl-iff*: ‹*wf* (*r*+) ⟷ *wf r*›
⟨*proof*⟩

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*⟩

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 m*0 *n*0.
*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 ($\lambda b.\ b \longrightarrow$ *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 ($\lambda b.\ b \longrightarrow$ *size* (*get-all-learned-clss S*) > *last-Restart-learnt-clss*)
}›

definition *inprocessing-required* :: *'v twl-st* ⇒ *bool nres* **where**
‹*inprocessing-required S* = *do* {
SPEC ($\lambda b.$ *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*(λ*T*. *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*(λ*T*. *cdcl-twl-restart S T*);
RETURN (*T*, *T*, *T*, *m*, *Suc n*)
}
else do {
T ← *SPEC*(λ*T*. *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* (S_0, T_0, U_0, m_0, n_0) (*brk*, *R*, *S*, *T*, *m*, *n*) ⟷
twl-restart-inv (S_0, T_0, U_0, m_0, n_0, *True*) ∧
twl-stgy-restart-inv (S_0, T_0, U_0, m_0, n_0, *True*) ∧
(*brk* ⟶ *final-twl-state T*) ∧
*cdcl-twl-stgy-restart*** (S_0, T_0, U_0, m_0, n_0, *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* $(S_0,\ T_0,\ U_0,\ m_0,\ n_0)$ $(brk,\ R,\ S,\ T,\ m,\ n) \longleftrightarrow$
twl-restart-inv $(S_0,\ T_0,\ U_0,\ m_0,\ n_0,\ True) \wedge$
twl-stgy-restart-inv $(S_0,\ T_0,\ U_0,\ m_0,\ n_0,\ True) \wedge$
twl-stgy-restart-inv $(R,\ S,\ T,\ m,\ n,\ \neg brk) \wedge$
twl-restart-inv $(R,\ S,\ T,\ m,\ n,\ \neg brk) \wedge$
$(brk \longrightarrow$ *final-twl-state* $T) \wedge$
cdcl-twl-stgy-restart** $(S_0,\ T_0,\ U_0,\ m_0,\ n_0,\ True)$ $(R,\ S,\ T,\ m,\ n,\ \neg brk) \wedge$
clauses-to-update $T = \{\#\} \wedge (\neg brk \longrightarrow$ *get-conflict* $T = None)$›
⟨*proof*⟩

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* $(S_0,\ T_0,\ U_0,\ m_0,\ n_0)$
$(\lambda(brk,$ -$).\ \neg brk)$
$(\lambda(brk,\ S,\ S',\ S'',\ m,\ \ n).$
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 $(brk \vee$ *get-conflict* $T \neq None,\ S,\ S',\ T,\ m',\ n')$
})
$(False,\ S_0,\ T_0,\ U_0,\ m_0,\ n_0)$;
RETURN T
}›

abbreviation *cdcl-twl-stgy-restart-prog-int* **where**
‹*cdcl-twl-stgy-restart-prog-int* $S \equiv$ *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* :: *'v twl-st*, *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 $(R,\ S,\ T,\ m,\ n,\ \neg brkS) \wedge \neg brkS \wedge$
cdcl-twl-stgy-restart^{++} $(R,\ S,\ T,\ m,\ n,\ \neg brkS)$ $(R',\ S',\ T',\ m',\ n',\ \neg brkT)\} \cup$
{$(($*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*)).
$\neg brkS \wedge brkT\}$›

end

context *twl-restart*
begin

lemma *cdcl-twl-stgy-restart-tranclpI*:
‹*cdcl-twl-stgy*$^{++}$ *T U* $\Longrightarrow$ *cdcl-twl-stgy-restart*$^{++}$ (*R*, *S*, *T*, *m*, *n*, *True*) (*R*, *S*, *U*, *m*, *n*, *True*)›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-rtranclpI*:
‹*cdcl-twl-stgy*** *T U* $\Longrightarrow$ *cdcl-twl-stgy-restart*** (*R*, *S*, *T*, *m*, *n*, *True*) (*R*, *S*, *U*, *m*, *n*, *True*)›
⟨*proof*⟩

lemma *wf-cdcl-algo-termination-early-rel*: ‹*wf cdcl-algo-termination-early-rel*› (**is** ‹*wf* (*?C* ∪ *?D*)›)
⟨*proof*⟩

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\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}int\text{-}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* $\Longrightarrow$
size (*pget-all-learned-clss T*) ≤ *size* (*pget-all-learned-clss U*)›
⟨*proof*⟩

lemma *cdcl-twl-cp-get-all-learned-clss*:
‹*cdcl-twl-cp T U* $\Longrightarrow$
size (*get-all-learned-clss T*) = *size* (*get-all-learned-clss U*)›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-cp-get-all-learned-clss*:
‹*cdcl-twl-cp*** *T U* $\Longrightarrow$
size (*get-all-learned-clss T*) = *size* (*get-all-learned-clss U*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-inp-clauses-to-update*:
‹*cdcl-twl-inp*** *S T* ⟹ *clauses-to-update S* = {#} ⟹ *clauses-to-update T* = {#}›
⟨*proof*⟩

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*
(λ*s*′. (*cdcl-twl-stgy-restart-prog-int-inv* (*S*$_0$, *T*$_0$, *U*$_0$, *m*$_0$, *n*$_0$) ∘ *snd*) *s*′ ∧
(*s*′, *ebrk*, *brk*, *S*, *T*, *U*, *m*, *n*)
∈ *cdcl-algo-termination-early-rel*))›
(**is** ‹- ≤ *SPEC* (λ*x*. - ≤ *SPEC*(λ*s*. *?I s* ∧ - ∈ *?term*))›)
⟨*proof*⟩

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*›
⟨*proof*⟩

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⟩

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')).

$S = S' \wedge$ *last-GC* $=$ *size* (*get-all-learned-clss last-GC'*) $\wedge$
last-Restart $=$ *size* (*get-all-learned-clss last-Restart'*) $\wedge$
$n = n'\}\rangle$*nres-rel*›
⟨*proof*⟩

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* (S_0, n_0)(*brk*, *T*, *last-GC*, *last-Restart*, *n*) ⟷
(∃ *last-GC'* *last-Restart'* *m* m_0 T_0 U_0. *last-GC* $=$ *size* (*get-all-learned-clss last-GC'*) $\wedge$
last-Restart $=$ *size* (*get-all-learned-clss last-Restart'*) $\wedge$
cdcl-twl-stgy-restart-prog-int-inv (T_0, U_0, S_0, m_0, n_0) (*brk*, *last-GC'*, *last-Restart'*, *T*, *m*, *n*))›

lemma *cdcl-twl-stgy-restart-prog-inv-def*:
‹*cdcl-twl-stgy-restart-prog-inv*= (λ(S_0, n_0) (*brk*, *T*, *last-GC*, *last-Restart*, *n*).
(∃ *last-GC'* *last-Restart'* *m* m_0 T_0 U_0. *last-GC* $=$ *size* (*get-all-learned-clss last-GC'*) $\wedge$
last-Restart $=$ *size* (*get-all-learned-clss last-Restart'*) $\wedge$
cdcl-twl-stgy-restart-prog-int-inv (T_0, U_0, S_0, m_0, n_0) (*brk*, *last-GC'*, *last-Restart'*, *T*, *m*, *n*)))›
⟨*proof*⟩

definition *cdcl-twl-stgy-restart-progg*
:: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'v twl-st* ⇒ *'v twl-st nres*
where
‹*cdcl-twl-stgy-restart-progg* n_0 *last-GC last-Restart* S_0 =
do {
(*brk*, *T*, -, -, -) ← $WHILE_T$*cdcl-twl-stgy-restart-prog-inv* (S_0, n_0)
(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*, S_0, *last-GC*, *last-Restart*, n_0);
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'*))›
⟨*proof*⟩

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*⟩

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*⟩

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)) $S_0 \leq$ *conclusive-TWL-run2* m_0 n_0 T_0 U_0 S_0›
⟨*proof*⟩

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* $S \leq$ *conclusive-TWL-run* S›
⟨*proof*⟩

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* $T \neq$ *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* $S \equiv$ *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* n_0 (*size* (*get-all-learned-clss* T_0))
(*size* (*get-all-learned-clss* U_0)) $S \leq \Downarrow Id$ (*cdcl-twl-stgy-restart-prog-bounded-intg* m_0 n_0 T_0 U_0 S)›
⟨*proof*⟩

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*⟩
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*›
⟨*proof*⟩

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
}›
 ⟨proof⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

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*›

⟨*proof*⟩

lemma *convert-lits-l-update-sel*:
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow C' = N \propto C \Longrightarrow$
$(M, M') \in \mathit{convert\text{-}lits\text{-}l}\ (\mathit{fmupd}\ C\ (C', b)\ N)\ E \longleftrightarrow (M, M') \in \mathit{convert\text{-}lits\text{-}l}\ N\ E$›
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd-in-irrelev*: ‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \neg \mathit{irred}\ N\ C \Longrightarrow$
$\mathit{learned\text{-}clss\text{-}l}\ (\mathit{fmupd}\ C\ (C', \mathit{True})\ N) = \mathit{remove1\text{-}mset}\ (N \propto C, \mathit{irred}\ N\ C)\ (\mathit{learned\text{-}clss\text{-}l}\ N)$›
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-irrelev*:
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \neg \mathit{irred}\ N\ C \Longrightarrow$
$\mathit{init\text{-}clss\text{-}l}\ (\mathit{fmupd}\ C\ (C', \mathit{True})\ N) = \mathit{add\text{-}mset}\ (C', \mathit{True})\ ((\mathit{init\text{-}clss\text{-}l}\ N))$›
⟨*proof*⟩

lemma *cdcl-twl-subsumed-l-cdcl-twl-subsumed*:
assumes ‹*cdcl-twl-subsumed-l S T*› **and**
SS': ‹$(S, S') \in \mathit{twl\text{-}st\text{-}l}\ \mathit{None}$›
shows ‹$\exists T'.\ (T, T') \in \mathit{twl\text{-}st\text{-}l}\ \mathit{None} \wedge \mathit{cdcl\text{-}twl\text{-}subsumed}\ S'\ T'$›
⟨*proof*⟩

lemma *cdcl-twl-subsumed-l-list-invs*:
‹*cdcl-twl-subsumed-l S T* $\Longrightarrow$ *twl-list-invs S* $\Longrightarrow$ *twl-list-invs T*›
⟨*proof*⟩

inductive *cdcl-twl-subresolution-l* :: ‹*'v twl-st-l* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *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* ($N \propto C'$)) *NS*, *US*, *N0*, *U0*, {#}, 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'›
‹*mset* E = *remdups-mset* D'› ‹*length* $E \geq 2$› ‹*distinct* E› ‹$\forall L \in\#$ *mset* E. *undefined-lit* M L›
‹$C' \notin$ *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* ($N \propto C'$)) *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'› |
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* ($N \propto C'$)) *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
‹$C \in\# \mathit{dom\text{-}m}\ N$›
‹$C' \in\# \mathit{dom\text{-}m}\ N$›
‹$\mathit{mset}\ (N \propto C) = \mathit{add\text{-}mset}\ L\ D$›
‹$\mathit{mset}\ (N \propto C') = \mathit{add\text{-}mset}\ (-L)\ D'$›
‹$\mathit{count\text{-}decided}\ M = 0$› ‹$D \subseteq\# D'$›

‹*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-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* ∝ *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'*›

lemma *convert-lits-l-update-sel2*:
‹*C* ∈# *dom-m N* ⟹ *C* ∉ *set* (*get-all-mark-of-propagated M*) ⟹
(*M*, *M'*) ∈ *convert-lits-l* (*fmupd C* (*C'*, *b*) *N*) *E* ⟷ (*M*, *M'*) ∈ *convert-lits-l N E*›
⟨*proof*⟩

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* ⊆# *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'* ∈# *N*›
⟨*proof*⟩

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* ⊆# *D'*› ‹¬*tautology* (*D* + *D'*)›
‹*remdups-mset D'* = {#*K*#}›
‹*undefined-lit M K*›
‹*C* ∈# *N*›
‹*C'* ∈# *N*›
⟨*proof*⟩

lemma *twl-subresolution-LL-nonunit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹*S* = (*M*, *N*, *U*, *None*, *NE*, *UE*, *NS*, *US*, *N0*, *U0*, {#}, *Q*)›

‹$T = (M,\ N,\ \textit{add-mset}\ E\ (\textit{remove1-mset}\ C'\ U),\ \textit{None},\ NE,\ UE,\ NS,\ \textit{add-mset}\ (\textit{clause}\ C')\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$\textit{clause}\ C = \textit{add-mset}\ L\ D$›
‹$\textit{clause}\ C' = \textit{add-mset}\ (-L)\ D'$›
‹$\textit{count-decided}\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg\textit{tautology}\ (D + D')$›
‹$\textit{clause}\ E = \textit{remdups-mset}\ D'$› ‹$\textit{size}\ (\textit{watched}\ E) = 2$› ‹$\textit{struct-wf-twl-cls}\ E$›
‹$\forall L \in\#\ \textit{clause}\ E.\ \textit{undefined-lit}\ M\ L$›
‹$C \in\#\ U$›
‹$C' \in\#\ U$›
⟨*proof*⟩

lemma *twl-subresolution-LL-unit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹$S = (M,\ N,\ U,\ \textit{None},\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$T = (\textit{Propagated}\ K\ \{\#K\#\}\ \#\ M,\ (N),\ \textit{remove1-mset}\ C'\ U,\ \textit{None},\ NE,\ \textit{add-mset}\ \{\#K\#\}\ UE,\ NS,\ \textit{add-mset}\ (\textit{clause}\ C')\ US,\ N0,\ U0,\ \{\#\},\ \textit{add-mset}\ (-K)\ Q)$›
‹$\textit{clause}\ C = \textit{add-mset}\ L\ D$›
‹$\textit{clause}\ C' = \textit{add-mset}\ (-L)\ D'$›
‹$\textit{count-decided}\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg\textit{tautology}\ (D + D')$›
‹$\textit{remdups-mset}\ D' = \{\#K\#\}$›
‹$\textit{undefined-lit}\ M\ K$›
‹$C \in\#\ U$›
‹$C' \in\#\ U$›
⟨*proof*⟩

lemma *twl-subresolution-LI-nonunit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹$S = (M,\ N,\ U,\ \textit{None},\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$T = (M,\ N,\ \textit{add-mset}\ E\ (\textit{remove1-mset}\ C'\ U),\ \textit{None},\ NE,\ UE,\ NS,\ \textit{add-mset}\ (\textit{clause}\ C')\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$\textit{clause}\ C = \textit{add-mset}\ L\ D$›
‹$\textit{clause}\ C' = \textit{add-mset}\ (-L)\ D'$›
‹$\textit{count-decided}\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg\textit{tautology}\ (D + D')$›
‹$\textit{clause}\ E = \textit{remdups-mset}\ D'$› ‹$\textit{size}\ (\textit{watched}\ E) = 2$› ‹$\textit{struct-wf-twl-cls}\ E$›
‹$\forall L \in\#\ \textit{clause}\ E.\ \textit{undefined-lit}\ M\ L$›
‹$C \in\#\ N$›
‹$C' \in\#\ U$›
⟨*proof*⟩

lemma *twl-subresolution-LI-unit-simps*:
‹*cdcl-twl-subresolution S T*›
if
‹$S = (M,\ N,\ U,\ \textit{None},\ NE,\ UE,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
‹$T = (\textit{Propagated}\ K\ \{\#K\#\}\ \#\ M,\ (N),\ \textit{remove1-mset}\ C'\ U,\ \textit{None},\ NE,\ \textit{add-mset}\ \{\#K\#\}\ UE,\ NS,\ \textit{add-mset}\ (\textit{clause}\ C')\ US,\ N0,\ U0,\ \{\#\},\ \textit{add-mset}\ (-K)\ Q)$›
‹$\textit{clause}\ C = \textit{add-mset}\ L\ D$›
‹$\textit{clause}\ C' = \textit{add-mset}\ (-L)\ D'$›
‹$\textit{count-decided}\ M = 0$› ‹$D \subseteq\# D'$› ‹$\neg\textit{tautology}\ (D + D')$›
‹$\textit{remdups-mset}\ D' = \{\#K\#\}$›
‹$\textit{undefined-lit}\ M\ K$›
‹$C \in\#\ N$›
‹$C' \in\#\ U$›
⟨*proof*⟩

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-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*›
‹∀ *L* ∈# *clause E*. *undefined-lit M L*›
‹*C'* ∈# *N*›
‹*C* ∈# *U*›
⟨*proof*⟩

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-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'* ∈# *N*›
‹*C* ∈# *U*›
⟨*proof*⟩

lemma *cdcl-twl-subresolution-l-cdcl-twl-subresolution*:
assumes ‹*cdcl-twl-subresolution-l S T*› **and**
SS': ‹(*S*, *S'*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*›
shows ‹∃ *T'*. (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-subresolution S' T'*›
⟨*proof*⟩

inductive *cdcl-twl-unitres-true-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l* ⇒ *bool*› **where**
‹$cdcl\text{-}twl\text{-}unitres\text{-}true\text{-}l\ (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, Q)$
$(M, fmdrop\ C\ N, None, add\text{-}mset\ (mset\ (N \propto C))\ NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, Q)$›
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*)› |
‹$cdcl\text{-}twl\text{-}unitres\text{-}true\text{-}l\ (M, N, None, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, Q)$
$(M, fmdrop\ C\ N, None, NE, add\text{-}mset\ (mset\ (N \propto C))\ UE, NEk, UEk, NS, US, N0, U0, \{\#\}, Q)$›
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 ‹$S = (M, N, U, None, NE, UE, NS, US, N0, U0, \{\#\}, Q)$›
‹$T = (M, remove1\text{-}mset\ C\ N, U, None, add\text{-}mset\ (clause\ C)\ NE, UE, NS, US, N0, U0, \{\#\}, Q)$›
‹*L* ∈# *clause C*› ‹*get-level M L = 0*› ‹*L* ∈ *lits-of-l M*› ‹*C* ∈# *N*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

lemma *cdcl-twl-subresolution-l-list-invs*:
‹*cdcl-twl-subresolution-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
⟨*proof*⟩

lemma *cdcl-twl-unitres-True-l-list-invs*:
‹*cdcl-twl-unitres-true-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
⟨*proof*⟩

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* ∉ *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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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 $(N \propto C) = 1$
then do {
let $L = ((N \propto C) ! 0)$;
let $T = (Propagated\ L\ 0\ \#\ M,\ fmdrop\ C\ N,\ D,\ NE,\ UE,\ (if\ irr\ then\ add\text{-}mset\ \{\#L\#\}\ else\ id)\ NEk,$
$(if\ \neg irr\ then\ add\text{-}mset\ \{\#L\#\}\ else\ id)\ UEk,\ (if\ irr\ then\ add\text{-}mset\ E\ else\ id)\ NS,$
$(if\ \neg irr\ then\ add\text{-}mset\ E\ else\ id)\ US,\ N0,\ U0,\ WS,\ add\text{-}mset\ (-L)\ Q)$;
ASSERT $(set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ T) = set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ S))$;
ASSERT $(set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ T) = set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ S))$;
ASSERT $(undefined\text{-}lit\ M\ L \wedge L \in\# all\text{-}init\text{-}lits\text{-}of\text{-}l\ S)$;
RETURN T}
else if size $(N \propto C) = 0$
then do {
let $T = (M,\ fmdrop\ C\ N,\ Some\ \{\#\},\ NE,\ UE,\ NEk,\ UEk,\ (if\ irr\ then\ add\text{-}mset\ E\ else\ id)$
$NS,\ (if\ \neg irr\ then\ add\text{-}mset\ E\ else\ id)\ US,\ (if\ irr\ then\ add\text{-}mset\ \{\#\}\ else\ id)\ N0,\ (if\ \neg irr\ then\ add\text{-}mset$
$\{\#\}\ else\ id)\ U0,\ WS,\ \{\#\})$;
ASSERT $(set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ T) = set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ S))$;
ASSERT $(set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ T) = set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ S))$;
RETURN T
}
else do {
let $T = (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ (if\ irr\ then\ add\text{-}mset\ E\ else\ id)\ NS,\ (if\ \neg irr\ then$
$add\text{-}mset\ E\ else\ id)\ US,\ N0,\ U0,\ WS,\ Q)$;
ASSERT $(set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ T) = set\text{-}mset\ (all\text{-}init\text{-}lits\text{-}of\text{-}l\ S))$;
ASSERT $(set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ T) = set\text{-}mset\ (all\text{-}learned\text{-}lits\text{-}of\text{-}l\ S))$;
RETURN T
}
}
})›

lemma *true-clss-clss-def-more-atms*:
‹$N \models ps\ N' \longleftrightarrow (\forall I.\ total\text{-}over\text{-}m\ I\ (N \cup N' \cup N'') \longrightarrow consistent\text{-}interp\ I \longrightarrow I \models s\ N \longrightarrow I \models s\ N'$)›
(**is** ‹$?A \longleftrightarrow ?B$›)
⟨*proof*⟩

lemma *true-clss-clss-def-iff-negation-in-model*:
‹$A \models ps\ CNot\ C' \longleftrightarrow (\forall L \in\#\ C'.\ A \models ps\ \{\{\#-L\#\}\})$›
⟨*proof*⟩

lemma
fixes M :: ‹$('v,\ nat)\ ann\text{-}lit\ list$› **and** $N\ NE\ UE\ NS\ US\ N0\ U0\ Q\ NEk\ UEk$
defines ‹$S \equiv (M,\ N,\ None,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ Q)$›
assumes
‹$D \in\#\ dom\text{-}m\ N$›
‹$count\text{-}decided\ M = 0$› **and**
‹$D \notin set\ (get\text{-}all\text{-}mark\text{-}of\text{-}propagated\ M)$› **and**
ST: ‹$(S,\ T) \in twl\text{-}st\text{-}l\ None$› **and**
st-invs: ‹$twl\text{-}struct\text{-}invs\ T$› **and**
dec: ‹$count\text{-}decided\ (get\text{-}trail\text{-}l\ S) = 0$› **and**
false: ‹$\forall L \in\#\ C'.\ \neg undefined\text{-}lit\ (get\text{-}trail\text{-}l\ S)\ L$›
‹$\forall L \in\#\ C'.\ L \notin lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ S)$› **and**
ent: ‹$cdcl_W\text{-}restart\text{-}mset.cdcl_W\text{-}learned\text{-}clauses\text{-}entailed\text{-}by\text{-}init\ (state_W\text{-}of\ T)$›
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*⟩

lemma *fmdrop-eq-update-eq*: ‹*fmdrop C aa* = *fmdrop C bh* ⟹ *C* ∈# *dom-m bh* ⟹
bh = *fmupd C* (*bh* ∝ *C*, *irred bh C*) *aa*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *list-length-2-isabelle-come-on*:
‹*length C* ≠ *Suc 0* ⟹ *C* ≠ [] ⟹ *length C* ≥ *2*›
⟨*proof*⟩

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*#}›
⟨*proof*⟩

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*›
⟨*proof*⟩
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*›
⟨*proof*⟩

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*(λ*T*.
(*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*⟩

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* $\neg$*irred* N C' *then add-mset* (*mset* $(N \propto C')$) *else id*) o (*if* $\neg$*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* $\neg$*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. $\exists S'$. *length* (*get-clauses-l* $S \propto C$) $\geq$ 2 $\wedge$ $C \in\#$ *dom-m* (*get-clauses-l* S) $\wedge$
count-decided (*get-trail-l* S) = 0 $\wedge$ *distinct* (*get-clauses-l* $S \propto C$) $\wedge$ ($\forall L \in$*set* (*get-clauses-l* $S \propto C$). *undefined-lit* (*get-trail-l* S) L) $\wedge$ *get-conflict-l* S = *None* $\wedge$
$C \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S)) $\wedge$ *clauses-to-update-l* $S = \{\#\} \wedge$
twl-list-invs $S \wedge$
(S,S')∈*twl-st-l None* $\wedge$ $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of* S') $\wedge$
twl-struct-invs $S' \wedge$
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of* S') $\wedge$
(*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)) $\wedge$ *distinct* ((*get-clauses-l* S) $\propto C'$) $\wedge$ $C \neq C' \wedge \neg$*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) $\propto C$) $\wedge$ $\neg$*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) $\wedge$ *distinct* ((*get-clauses-l* S) $\propto C'$) $\wedge$
$C' \notin$ *set* (*get-all-mark-of-propagated* (*get-trail-l* S))$\wedge$ 2 $\leq$ *length* ((*get-clauses-l* S) $\propto C'$) $\wedge$
$\neg$ *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* $\neg$*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* $\neg$*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'*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

lemma *cdcl-twl-pure-remove-l-twl-list-invs*:
‹*cdcl-twl-pure-remove-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*)) ⟹
¬ *tautology* (*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, False*) *N, None, NE, UE, NEk, UEk, NS, add-mset* (*mset* (*N* ∝ *C'*)) *US, N0, U0,*
{#}, *Q*)›

$\langle C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow C' \in\# \mathit{dom\text{-}m}\ N \Longrightarrow -L \in\# \mathit{mset}\ (N \propto C) \Longrightarrow L \in\# \mathit{mset}\ (N \propto C') \Longrightarrow \mathit{count\text{-}decided}\ M = 0 \Longrightarrow$
$\mathit{remove1\text{-}mset}\ (-L)\ (\mathit{mset}\ (N \propto C)) \subseteq\# \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C')) \Longrightarrow$
$\mathit{distinct}\ (N \propto C') \Longrightarrow 3 \leq \mathit{length}\ (N \propto C') \Longrightarrow$
$\forall L \in \mathit{set}\ (N \propto C').\ \mathit{undefined\text{-}lit}\ M\ L \Longrightarrow$
$C \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow$
$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow \mathit{mset}\ E = \mathit{mset}\ (\mathit{remove1}\ (L)\ (N \propto C')) \Longrightarrow$
$\mathit{irred}\ N\ C' \Longrightarrow \mathit{irred}\ N\ C \Longrightarrow \mathit{length}\ (N \propto C') = \mathit{length}\ (N \propto C) \Longrightarrow$
$\mathit{cdcl\text{-}twl\text{-}inprocessing\text{-}l}^{**}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
$(M,\ \mathit{fmdrop}\ C\ (\mathit{fmupd}\ C'\ (E,\ \mathit{True})\ N),\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},$
$\mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C))\ \ (\mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{NS}),$
$\mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)\rangle$
$\langle C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow C' \in\# \mathit{dom\text{-}m}\ N \Longrightarrow -L \in\# \mathit{mset}\ (N \propto C) \Longrightarrow L \in\# \mathit{mset}\ (N \propto C') \Longrightarrow \mathit{count\text{-}decided}\ M = 0 \Longrightarrow$
$\mathit{remove1\text{-}mset}\ (-L)\ (\mathit{mset}\ (N \propto C)) \subseteq\# \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C')) \Longrightarrow$
$\mathit{distinct}\ (N \propto C') \Longrightarrow 3 \leq \mathit{length}\ (N \propto C') \Longrightarrow$
$\forall L \in \mathit{set}\ (N \propto C').\ \mathit{undefined\text{-}lit}\ M\ L \Longrightarrow$
$C \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow$
$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow \mathit{mset}\ E = \mathit{mset}\ (\mathit{remove1}\ (L)\ (N \propto C')) \Longrightarrow$
$\neg\ \mathit{tautology}\ (\mathit{remove1\text{-}mset}\ (-\ L)\ (\mathit{mset}\ (N \propto C)) + \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C'))) \Longrightarrow$
$\neg \mathit{irred}\ N\ C' \Longrightarrow \neg \mathit{irred}\ N\ C \Longrightarrow \mathit{length}\ (N \propto C') = \mathit{length}\ (N \propto C) \Longrightarrow$
$\mathit{cdcl\text{-}twl\text{-}inprocessing\text{-}l}^{**}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
$(M,\ \mathit{fmdrop}\ C\ (\mathit{fmupd}\ C'\ (E,\ \mathit{False})\ N),\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},$
$\mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C))\ \ (\mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{US}),\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)\rangle$
$\langle C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow C' \in\# \mathit{dom\text{-}m}\ N \Longrightarrow -L \in\# \mathit{mset}\ (N \propto C) \Longrightarrow L \in\# \mathit{mset}\ (N \propto C') \Longrightarrow \mathit{count\text{-}decided}\ M = 0 \Longrightarrow$
$\mathit{remove1\text{-}mset}\ (-L)\ (\mathit{mset}\ (N \propto C)) \subseteq\# \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C')) \Longrightarrow$
$\mathit{distinct}\ (N \propto C') \Longrightarrow 3 \leq \mathit{length}\ (N \propto C') \Longrightarrow$
$\forall L \in \mathit{set}\ (N \propto C').\ \mathit{undefined\text{-}lit}\ M\ L \Longrightarrow$
$C \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow$
$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow$
$\neg\ \mathit{tautology}\ (\mathit{remove1\text{-}mset}\ (-\ L)\ (\mathit{mset}\ (N \propto C)) + \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C'))) \Longrightarrow$
$\neg \mathit{irred}\ N\ C' \Longrightarrow \mathit{irred}\ N\ C \Longrightarrow \mathit{length}\ (N \propto C') = \mathit{length}\ (N \propto C) \Longrightarrow \mathit{mset}\ E = \mathit{mset}\ (\mathit{remove1}\ (L)\ (N \propto C')) \Longrightarrow$
$\mathit{cdcl\text{-}twl\text{-}inprocessing\text{-}l}^{**}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
$(M,\ \mathit{fmdrop}\ C\ (\mathit{fmupd}\ C'\ (E,\ \mathit{True})\ N),\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C))\ \mathit{NS},$
$(\mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{US}),\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)\rangle$
$\langle C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow C' \in\# \mathit{dom\text{-}m}\ N \Longrightarrow -L \in\# \mathit{mset}\ (N \propto C) \Longrightarrow L \in\# \mathit{mset}\ (N \propto C') \Longrightarrow \mathit{count\text{-}decided}\ M = 0 \Longrightarrow$
$\mathit{remove1\text{-}mset}\ (-L)\ (\mathit{mset}\ (N \propto C)) \subseteq\# \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C')) \Longrightarrow$
$\mathit{distinct}\ (N \propto C') \Longrightarrow 3 \leq \mathit{length}\ (N \propto C') \Longrightarrow$
$\forall L \in \mathit{set}\ (N \propto C').\ \mathit{undefined\text{-}lit}\ M\ L \Longrightarrow$
$C \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow$
$C' \notin \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ M) \Longrightarrow \mathit{mset}\ E = \mathit{mset}\ (\mathit{remove1}\ (L)\ (N \propto C')) \Longrightarrow$
$\neg\ \mathit{tautology}\ (\mathit{remove1\text{-}mset}\ (-\ L)\ (\mathit{mset}\ (N \propto C)) + \mathit{remove1\text{-}mset}\ L\ (\mathit{mset}\ (N \propto C'))) \Longrightarrow$
$\mathit{irred}\ N\ C' \Longrightarrow \neg \mathit{irred}\ N\ C \Longrightarrow \mathit{length}\ (N \propto C') = \mathit{length}\ (N \propto C) \Longrightarrow$
$\mathit{cdcl\text{-}twl\text{-}inprocessing\text{-}l}^{**}\ (M,\ N,\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{NS},\ \mathit{US},\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)$
$(M,\ \mathit{fmdrop}\ C\ (\mathit{fmupd}\ C'\ (E,\ \mathit{True})\ N),\ \mathit{None},\ \mathit{NE},\ \mathit{UE},\ \mathit{NEk},\ \mathit{UEk},\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C'))\ \mathit{NS},$
$(\mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C))\ \mathit{US}),\ \mathit{N0},\ \mathit{U0},\ \{\#\},\ Q)\rangle$
and

subresolution-strengtheningI-binary:
$\langle C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow C' \in\# \mathit{dom\text{-}m}\ N \Longrightarrow -L \in\# \mathit{mset}\ (N \propto C) \Longrightarrow L \in\# \mathit{mset}\ (N \propto C') \Longrightarrow \mathit{count\text{-}decided}\ M = 0 \Longrightarrow$
$\mathit{length}\ (N \propto C) = 2 \Longrightarrow \mathit{remove1\text{-}mset}\ (L)\ (\mathit{mset}\ (N \propto C')) \subseteq\# \mathit{remove1\text{-}mset}\ (-L)\ (\mathit{mset}\ (N \propto C)) \Longrightarrow$

distinct $(N \propto C') \Longrightarrow$ *length* $(N \propto C') \geq 2 \Longrightarrow$
$\forall L \in$ *set* $(N \propto C)$. *undefined-lit* $M\ L \Longrightarrow$
irred $N\ C \Longrightarrow C' \neq 0 \Longrightarrow$ *mset* $E =$ *mset* (*remove1* $(-L)$ $(N \propto C)) \Longrightarrow$
irred $N\ C' \Longrightarrow C \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow C' \notin$ *set* (*get-all-mark-of-propagated* M) $\Longrightarrow$
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* $(N \propto C'))$ *NE*, *UE*, *add-mset* {#*hd E*#} *NEk*, *UEk*,
(*add-mset* (*mset* $(N \propto C)$) *NS*), *US*, *N0*, *U0*, {#}, *add-mset* $(-$*hd E*$)$ 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* $M = 0 \Longrightarrow$
length $(N \propto C) = 2 \Longrightarrow$ *remove1-mset* (L) (*mset* $(N \propto C')) \subseteq\#$ *remove1-mset* $(-L)$ (*mset* $(N \propto C)) \Longrightarrow$
distinct $(N \propto C') \Longrightarrow$ *length* $(N \propto C') \geq 2 \Longrightarrow$
$\forall L \in$ *set* $(N \propto C)$. *undefined-lit* $M\ L \Longrightarrow$
$\neg$*irred* $N\ C \Longrightarrow \neg$*irred* $N\ C' \Longrightarrow C' \neq 0 \Longrightarrow$ *mset* $E =$ *mset* (*remove1* $(-L)$ $(N \propto C)) \Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow C' \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow$
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* $(N \propto C'))$ *UE*, *NEk*,
add-mset {#*hd E*#} *UEk*, *NS*, *add-mset* (*mset* $(N \propto C))$ *US*, *N0*, *U0*,
{#}, *add-mset* $(-$ *hd E*$)$ 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* $M = 0 \Longrightarrow$
length $(N \propto C) = 2 \Longrightarrow$ *remove1-mset* (L) (*mset* $(N \propto C')) \subseteq\#$ *remove1-mset* $(-L)$ (*mset* $(N \propto C)) \Longrightarrow$
distinct $(N \propto C') \Longrightarrow$ *length* $(N \propto C') \geq 2 \Longrightarrow$
$\forall L \in$ *set* $(N \propto C)$. *undefined-lit* $M\ L \Longrightarrow$
$\neg$*irred* $N\ C \Longrightarrow$ *irred* $N\ C' \Longrightarrow C' \neq 0 \Longrightarrow$ *mset* $E =$ *mset* (*remove1* $(-L)$ $(N \propto C)) \Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow C' \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow$
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* $(N \propto C'))$ *NE*, *UE*, *NEk*,
add-mset {#*hd E*#} *UEk*, *NS*, *add-mset* (*mset* $(N \propto C))$ *US*, *N0*, *U0*,
{#}, *add-mset* $(-$ *hd E*$)$ 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* $M = 0 \Longrightarrow$
length $(N \propto C) = 2 \Longrightarrow$ *remove1-mset* (L) (*mset* $(N \propto C')) \subseteq\#$ *remove1-mset* $(-L)$ (*mset* $(N \propto C)) \Longrightarrow$
distinct $(N \propto C') \Longrightarrow$ *length* $(N \propto C') \geq 2 \Longrightarrow$
$\forall L \in$ *set* $(N \propto C)$. *undefined-lit* $M\ L \Longrightarrow$
irred $N\ C \Longrightarrow \neg$*irred* $N\ C' \Longrightarrow C' \neq 0 \Longrightarrow$ *mset* $E =$ *mset* (*remove1* $(-L)$ $(N \propto C)) \Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow C' \notin$ *set* (*get-all-mark-of-propagated* $M) \Longrightarrow$
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* $(N \propto C'))$ *UE*, *add-mset* {#*hd E*#} *NEk*, *UEk*,
add-mset (*mset* $(N \propto C))$ *NS*, *US*, *N0*, *U0*, {#}, *add-mset* $(-$ *hd E*$)$ Q)⟩
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

lemma *cdcl-twl-unitres-l-list-invs*:
‹*cdcl-twl-unitres-l S T* ⟹ *twl-list-invs S* ⟹ *twl-list-invs T*›
⟨*proof*⟩

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*›

⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*) ∈ *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*)›
⟨*proof*⟩

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* ⟷ (∃ *C′ S′*.
(*S*,*S′*) ∈ *twl-st-l None* ∧
mset ((*get-clauses-l S*) ∝ *C*) = *mset* ((*get-clauses-l S*) ∝ *C′*) ∧
(¬*irred* (*get-clauses-l S*) *C′* ⟶ ¬*irred* (*get-clauses-l S*) *C*) ∧
C′ ∉ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∧
C ∈# *dom-m* (*get-clauses-l S*) ∧
clauses-to-update-l S = {#} ∧
C′ ∈# *dom-m* (*get-clauses-l S*) ∧
C ∉ *set* (*get-all-mark-of-propagated* (*get-trail-l S*)) ∧
C ≠ *C′* ∧
twl-list-invs S ∧ *twl-struct-invs S′* ∧
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* = (λ(*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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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-binary-clauses-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* (λ*L'*. *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*›
⟨*proof*⟩

lemma *fmdrop-eq-dom-m-iff*: ‹*C* ∈# *dom-m N* ⟹ *fmdrop C N* = *fmdrop C' N* ⟷ *C* = *C'*›
⟨*proof*⟩

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*)›

    ⟨*proof*⟩

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*)›

    ⟨*proof*⟩

lemma *deduplicate-binary-clauses*:
  assumes ‹*deduplicate-binary-clauses-pre L S*›
  shows ‹*deduplicate-binary-clauses L S* ≤ *SPEC*(*cdcl-twl-inprocessing-l*** *S*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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* (λ*L*. *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*⟩

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*)›
⟨*proof*⟩

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*⟩

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 ∉# *xs* ∧ *C* ∈# *dom-m* (*get-clauses-l S*) ∧ *count-decided* (*get-trail-l S*) = *0* ∧
clauses-to-update-l S = {#} ∧ *twl-list-invs S* ∧ *twl-struct-invs S'* ∧
set (*get-all-mark-of-propagated* (*get-trail-l S*)) ⊆ {*0*} ∧
(∀ *C'*∈#*xs*. *C'* ∈# *dom-m* (*get-clauses-l S*) ⟶ *length* (*get-clauses-l S* ∝ *C'*) ≤ *length* (*get-clauses-l S* ∝ *C*))∧
cdcl$_W$*-restart-mset.cdcl*$_W$*-learned-clauses-entailed-by-init* (*state*$_W$*-of S'*) ∧
subsume-or-strengthen-pre C s S ∧ *xs* ⊆# *zs* ∧
(*is-subsumed s* ⟶ *subsumed-by s* ∈# *zs*) ∧
(*is-strengthened s* ⟶ *strengthened-by s* ∈# *zs*)))›

definition *forward-subsumption-one-select* **where**
‹*forward-subsumption-one-select C ys S* = (λ*xs*. *C* ∉# *xs* ∧ *ys* ∩# *xs* = {#} ∧
(∀ *D*∈#*xs*. *D* ∈# *dom-m* (*get-clauses-l S*) ⟶
(∀ *L*∈ *set* (*get-clauses-l S* ∝ *D*). *undefined-lit* (*get-trail-l S*) *L*) ∧
(*length* (*get-clauses-l S* ∝ *D*) ≤ *length* (*get-clauses-l S* ∝ *C*))))›

definition *forward-subsumption-one* :: ‹*nat* ⇒ *nat multiset* ⇒ *'v twl-st-l* ⇒ (*'v twl-st-l* × *bool*) *nres*›
where
‹*forward-subsumption-one* = (λ*C cands S*$_0$. *do* {
ASSERT(*forward-subsumption-one-pre C cands S*$_0$);
xs$_0$ ← *SPEC* (*forward-subsumption-one-select C cands S*$_0$);
(*xs*, *s*) ←
WHILE$_T$ *forward-subsumption-one-inv C S*$_0$ *xs*$_0$ (λ(*xs*, *s*). *xs* ≠ {#} ∧ *s* = *NONE*)
(λ(*xs*, *s*). *do* {
C' ← *SPEC*(λ*C'*. *C'* ∈# *xs*);
if C' ∉# *dom-m* (*get-clauses-l S*$_0$)
then RETURN (*remove1-mset C' xs*, *s*)
else do {
s ← *SPEC*(*try-to-subsume C C'* (*get-clauses-l S*$_0$));
RETURN (*remove1-mset C' xs*, *s*)
}
})
(*xs*$_0$, *NONE*);
ASSERT (*forward-subsumption-one-inv C S*$_0$ *xs*$_0$ (*xs*, *s*));
S ← *subsume-or-strengthen C s S*$_0$;
ASSERT (*set-mset* (*all-learned-lits-of-l S*) ⊆ *set-mset* (*all-learned-lits-of-l S*$_0$) ∧ *set-mset* (*all-init-lits-of-l S*) = *set-mset* (*all-init-lits-of-l S*$_0$));
RETURN (*S*, *s* ≠ *NONE*)
}
)›

lemma *forward-subsumption-one*:
assumes ‹*forward-subsumption-one-pre C cands S*›
shows ‹*forward-subsumption-one C cands S* ≤ ⇓{((*st*, *SR*), *st'*). *st* = *st'* ∧ (¬*SR* ⟶ *st* = *S*)}
(*SPEC*(λ*T*. *cdcl-twl-inprocessing-l*** *S T* ∧ *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*⟩

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*(λ*xs*. *finite xs*);
T ← *FOREACHci*(λ*it T*. *simplify-clauses-with-unit-st-inv S it T*)
(*xs*)
(λ*S*. *get-conflict-l S* = *None*)
(λ*i 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*⟩

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*⟩

definition *simplify-clauses-with-units-st-pre* **where**
‹*simplify-clauses-with-units-st-pre S* ⟷

$(\exists\, T.\ (S,\ T) \in \textit{twl-st-l None} \wedge \textit{twl-struct-invs } T \wedge \textit{twl-list-invs } S \wedge$
$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-learned-clauses-entailed-by-init } ((\textit{state}_W\textit{-of } T)) \wedge$
$\textit{set } (\textit{get-all-mark-of-propagated } (\textit{get-trail-l } S)) \subseteq \{0\} \wedge$
$\textit{clauses-to-update-l } S = \{\#\} \wedge$
$\textit{count-decided } (\textit{get-trail-l } S) = 0)\rangle$

definition *simplify-clauses-with-units-st* **where**
$\langle \textit{simplify-clauses-with-units-st } S = \textit{do } \{$
$\textit{ASSERT}(\textit{simplify-clauses-with-units-st-pre } S);$
$\textit{new-units} \leftarrow \textit{SPEC } (\lambda b.\ b \longrightarrow \textit{get-conflict-l } S = \textit{None});$
$\textit{if new-units}$
$\textit{then simplify-clauses-with-unit-st } S$
$\textit{else RETURN } S\}\rangle$

lemma *simplify-clauses-with-units-st-spec*:
assumes $\langle \textit{count-decided } (\textit{get-trail-l } S) = 0\rangle$
$\langle \textit{clauses-to-update-l } S = \{\#\}\rangle$ **and**
ST: $\langle (S,\ T) \in \textit{twl-st-l None}\rangle$ **and**
st-invs: $\langle \textit{twl-struct-invs } T\rangle$ **and**
list-invs: $\langle \textit{twl-list-invs } S\rangle$ **and**
$\langle \textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-learned-clauses-entailed-by-init } ((\textit{state}_W\textit{-of } T))\rangle$ **and**
$\langle \textit{set } (\textit{get-all-mark-of-propagated } (\textit{get-trail-l } S)) \subseteq \{0\}\rangle$
shows $\langle \textit{simplify-clauses-with-units-st } S \leq \Downarrow \textit{Id } (\textit{SPEC}(\lambda T.\ \textit{cdcl-twl-inprocessing-l}^{**}\ S\ T \wedge$
$\textit{simplify-clauses-with-unit-st-inv } S\ \{\}\ T))\rangle$
⟨*proof*⟩

definition *try-to-forward-subsume-inv* :: $\langle 'v\ \textit{twl-st-l} \Rightarrow \textit{nat multiset} \Rightarrow \textit{nat} \Rightarrow \textit{nat} \times \textit{bool} \times 'v\ \textit{twl-st-l}$
$\Rightarrow \textit{bool}\rangle$ **where**
$\langle \textit{try-to-forward-subsume-inv } S0 = (\lambda \textit{cands } C\ (i,\textit{brk},S).$
$(\textit{cdcl-twl-inprocessing-l}^{**}\ S0\ S \wedge$
$\textit{clauses-to-update-l } S = \{\#\} \wedge$
$\textit{count-decided } (\textit{get-trail-l } S) = 0 \wedge$
$\textit{set } (\textit{get-all-mark-of-propagated } (\textit{get-trail-l } S)) \subseteq \{0\} \wedge$
$(\neg \textit{brk} \longrightarrow (\textit{get-conflict-l } S = \textit{None} \wedge C \in\# \textit{dom-m } (\textit{get-clauses-l } S) \wedge$
$(\forall L \in\# \textit{mset } (\textit{get-clauses-l } S \propto C).\ \textit{undefined-lit } (\textit{get-trail-l } S)\ L) \wedge$
$\textit{length } (\textit{get-clauses-l } S \propto C) > 2)) \wedge$
$(\textit{get-trail-l } S = \textit{get-trail-l } S0) \wedge$
$(\forall D \in\# \textit{remove1-mset } C\ (\textit{dom-m } (\textit{get-clauses-l } S)).$
$D \in\# \textit{dom-m } (\textit{get-clauses-l } S0) \wedge \textit{get-clauses-l } S \propto D = \textit{get-clauses-l } S0 \propto D) \wedge$
$(\forall D \in\# \textit{cands}.\ D \neq C \longrightarrow (D \in\# \textit{dom-m } (\textit{get-clauses-l } S0)) = (D \in\# \textit{dom-m } (\textit{get-clauses-l } S)))))\rangle$

definition *try-to-forward-subsume-pre* :: $\langle \textit{nat} \Rightarrow \textit{nat multiset} \Rightarrow 'v\ \textit{twl-st-l} \Rightarrow \textit{bool}\rangle$ **where**
$\langle \textit{try-to-forward-subsume-pre} = (\lambda C\ \textit{xs}\ S.$
$\exists\, T.\ C \neq 0 \wedge$
$(S,\ T) \in \textit{twl-st-l None} \wedge$
$\textit{twl-struct-invs } T \wedge$
$\textit{twl-list-invs } S \wedge$
$\textit{clauses-to-update-l } S = \{\#\} \wedge$
$\textit{get-conflict-l } S = \textit{None} \wedge$
$\textit{cdcl}_W\textit{-restart-mset.cdcl}_W\textit{-learned-clauses-entailed-by-init } (\textit{state}_W\textit{-of } T) \wedge$
$\textit{count-decided } (\textit{get-trail-l } S) = 0 \wedge$
$\textit{set } (\textit{get-all-mark-of-propagated } (\textit{get-trail-l } S)) \subseteq \{0\} \wedge$
$C \in\# \textit{dom-m } (\textit{get-clauses-l } S) \wedge$
$(\forall L \in\# \textit{mset } (\textit{get-clauses-l } S \propto C).\ \textit{undefined-lit } (\textit{get-trail-l } S)\ L) \wedge$
$\textit{length } (\textit{get-clauses-l } S \propto C) > 2)\rangle$

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*) ← $WHILE_T$ *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*›
⟨*proof*⟩

lemma *in-remove1-mset-dom-m-iff*[*simp*]: ‹*a* ∈# *remove1-mset C* (*dom-m N*) ⟷ *a* ≠ *C* ∧ *a* ∈# *dom-m N*›
⟨*proof*⟩

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*⟩

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*⟩

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))*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *in-all-lits-of-mm-direct-inI*: ‹−*L* ∈ ⋃(*set-mset* ‘ *set-mset A*) ⟹ *L* ∈# *all-lits-of-mm A*›
⟨*proof*⟩
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*⟩

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*›
⟨*proof*⟩

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 (λLs. 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 (λL. 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⟩

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* (λ*occs2*.
(*occs2* = (λ*L*. *if L* ∈ *set* (*get-clauses-l S* ∝ *C*) *then True else occs L*)))›
⟨*proof*⟩

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* = (λ*L*. *L*∈⋃(*set-mset* ‘ *set-mset* (*mset* ‘# (λ*C*. *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* (λ*xs*. *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* (λ*C*. *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*

⟨*proof*⟩

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*⟩

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*$_T^{\textit{pure-literal-deletion-l-inv S xs}}$ (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-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*⟩

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*⟩

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* (λ*T*. *cdcl-twl-inprocessing-l*** *S T*)›
⟨*proof*⟩

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* (λ*T*. *cdcl-twl-inprocessing-l*** *S T*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*)›
⟨*proof*⟩

lemma *valid-trail-reduction-refl*[*simp*]:
‹*valid-trail-reduction M M*›
⟨*proof*⟩

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*⟩

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*⟩

lemma *valid-trail-reduction-Propagated-inD*:
‹*valid-trail-reduction M M' ⟹ Propagated L C ∈ set M' ⟹ ∃ C'. Propagated L C' ∈ set M*›
⟨*proof*⟩

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'›
⟨*proof*⟩

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'›
⟨*proof*⟩

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*⟩

lemma *valid-trail-reduction-length-leD*: ‹*valid-trail-reduction M M' ⟹ length M' ≤ length M*›
⟨*proof*⟩

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*⟩

lemma *map-lit-of-eq-defined-litD*: ‹*map lit-of M = map lit-of M' ⟹ defined-lit M = defined-lit M'*›
⟨*proof*⟩

lemma *map-lit-of-eq-no-dupD*: ‹*map lit-of M = map lit-of M' ⟹ no-dup M = no-dup M'*›
⟨*proof*⟩

Remarks about the predicate:

- The cases $\forall\, L\ E\ E'.\ \mathit{Propagated}\ L\ E \in \mathit{set}\ M' \longrightarrow \mathit{Propagated}\ L\ E' \in \mathit{set}\ M \longrightarrow E = (0::'b) \longrightarrow E' \neq (0::'c) \longrightarrow P$ 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›
⟨*proof*⟩

lemma *cdcl-twl-restart-l-list-invs*:
assumes
‹*cdcl-twl-restart-l S T*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs T*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
  ⟨*proof*⟩

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'*›
  ⟨*proof*⟩

lemma *cdcl-twl-restart-only-l-cdcl-twl-restart-only-spec*:
  assumes *ST*: ‹(*S*, *T*) ∈ *twl-st-l None*› ‹*twl-list-invs S*›
  shows ‹*SPEC* (*cdcl-twl-restart-only-l S*) ≤ ⇓ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
      clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None*}
    (*SPEC* (*cdcl-twl-restart-only T*))›
  ⟨*proof*⟩

lemma *cdcl-twl-restart-l-cdcl-twl-restart*:
  assumes *ST*: ‹(*S*, *T*) ∈ *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*) ≤ ⇓ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
      clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None*}
    (*SPEC* (*cdcl-twl-restart T*))›
⟨*proof*⟩

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*:
‹$i \in set\ A \Longrightarrow mset\ A = B \Longrightarrow i \in\# B$›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *reduce-dom-clauses-refl*[*simp*]: ‹*reduce-dom-clauses N N*›
⟨*proof*⟩

lemma *reduce-dom-clauses-trans*:
‹*reduce-dom-clauses N N'* ⟹ *reduce-dom-clauses N' N'a* ⟹ *reduce-dom-clauses N N'a*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

lemma *valid-trail-reduction-change-annot*:
‹*valid-trail-reduction* (*M* @ *Propagated L C* # *M'*)

(*M* @ *Propagated L 0* # *M'*)›
⟨*proof*⟩

lemma *valid-trail-reduction-eq-change-annot*:
‹*valid-trail-reduction-eq* (*M* @ *Propagated L C* # *M'*)
(*M* @ *Propagated L 0* # *M'*)›
⟨*proof*⟩

lemma *valid-trail-reduction-eq-refl*: ‹*valid-trail-reduction-eq M M*›
⟨*proof*⟩

lemma *valid-trail-reduction-eq-get-level*:
‹*valid-trail-reduction-eq M M'* ⟹ *get-level M* = *get-level M'*›
⟨*proof*⟩

lemma *valid-trail-reduction-eq-lits-of-l*:
‹*valid-trail-reduction-eq M M'* ⟹ *lits-of-l M* = *lits-of-l M'*›
⟨*proof*⟩

lemma *valid-trail-reduction-eq-trans*:
‹*valid-trail-reduction-eq M M'* ⟹ *valid-trail-reduction-eq M' M''* ⟹
valid-trail-reduction-eq M M''›
⟨*proof*⟩

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* ≠ *C'*› **and** ‹*C'* = *0*› **and**
‹*C* ∈# *dom-m N* ⟹ *mset* (*N*∝*C*) ∈# *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* ⟷ *is-proped z* ∨ *is-decided z*›
⟨*proof*⟩

abbreviation *different-annots-all-killed* **where**
‹*different-annots-all-killed N NUE* ≡ *list-all2* (*different-annot-all-killed N NUE*)›

lemma *different-annots-all-killed-refl*:
‹*different-annots-all-killed N NUE M M*›
⟨*proof*⟩

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* ⇒ *'v twl-st-l* ⇒ *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*∝*C*)) *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)›
if
‹*get-level* (*M* @ *Propagated L C* # *M'*) *L* = *0*› **and**
‹*C* > *0*› **and**
‹*C* ∈# *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*∝*C*)) *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)›
if
‹*get-level* (*M* @ *Propagated L C* # *M'*) *L* = *0*› **and**
‹*C* > *0*› **and**
‹*C* ∈# *dom-m N*› **and**
‹¬*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*∝*C*))*NE*, {#}, *NEk*, *UEk*, *NS*, {#}, *N0*, {#}, *W*, *Q*)›
if
‹*L* ∈ *lits-of-l M*› **and**
‹*get-level M L* = *0*› **and**
‹*C* ∈# *dom-m N*› **and**
‹*L* ∈ *set* (*N*∝*C*)› **and**
‹*irred N C*› **and**
‹∀ *L*. *Propagated L C* ∉ *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* ∈# *dom-m N*› **and**
‹¬*irred N C*› **and**
‹∀ *L*. *Propagated L C* ∉ *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. ∀ *L*. *Propagated L C* ∉ *set M* is overkill. However, I am currently unsure how I want to handle it (either as *Propagated* (*N* ∝ *C* ! *0*) *C* ∉ *set M* or as "the trail contains only zero anyway").

lemma *Ex-ex-eq-Ex*: ‹(∃ *NE'*. (∃ *b*. *NE'* = {#*b*#} ∧ *P b NE'*) ∧ *Q NE'*) ⟷
(∃ *b*. *P b* {#*b*#} ∧ *Q* {#*b*#})›
⟨*proof*⟩

lemma *in-set-definedD*:

‹*Propagated L′ C ∈ set M′ ⟹ defined-lit M′ L′*›
‹*Decided L′ ∈ set M′ ⟹ defined-lit M′ L′*›
⟨*proof*⟩

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 ∈ set* (*trail S*)› **and**
LC′: ‹*Propagated L′ C ∈ set* (*trail S*)›
shows *L* = *L′*
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *remove-one-annot-true-clause-get-conflict-l*:
‹*remove-one-annot-true-clause S T* ⟹ *get-conflict-l T* = *get-conflict-l S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*}›
⟨*proof*⟩

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-propagat* (*get-trail-l S*)) ∪ {*0*}›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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'›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*(λ*U'*. *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*⟩

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*)

$\vee$ (*count-decided* (*get-trail-l S*) = 0 $\wedge$ *k* = *length* (*get-trail-l S*)));
start ← *SPEC* (λ*i*. *i* $\leq$ *k* $\wedge$ ($\forall$ *j* < *i*. *is-proped* (*rev* (*get-trail-l S*) ! *j*) $\wedge$ *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* $\Longrightarrow$ *length* (*get-trail-l S*) = *length* (*get-trail-l T*)›
⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-imp-same-length*:
‹*remove-one-annot-true-clause*** *S T* $\Longrightarrow$ *length* (*get-trail-l S*) = *length* (*get-trail-l T*)›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-map-is-decided-trail*:
‹*remove-one-annot-true-clause S U* $\Longrightarrow$
map is-decided (*get-trail-l S*) = *map is-decided* (*get-trail-l U*)›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-map-mark-of-same-or-0*:
‹*remove-one-annot-true-clause S U* $\Longrightarrow$
mark-of (*get-trail-l S* ! *i*) = *mark-of* (*get-trail-l U* ! *i*) $\vee$ *mark-of* (*get-trail-l U* ! *i*) = 0›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-imp-inv-trans*:
‹*remove-one-annot-true-clause-imp-inv S* (*i*, *T*) $\Longrightarrow$ *remove-one-annot-true-clause-imp-inv T U* $\Longrightarrow$
remove-one-annot-true-clause-imp-inv S U›
⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-map-is-decided-trail*:
‹*remove-one-annot-true-clause*** *S U* $\Longrightarrow$
map is-decided (*get-trail-l S*) = *map is-decided* (*get-trail-l U*)›
⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-map-mark-of-same-or-0*:
‹*remove-one-annot-true-clause*** *S U* $\Longrightarrow$
mark-of (*get-trail-l S* ! *i*) = *mark-of* (*get-trail-l U* ! *i*) $\vee$ *mark-of* (*get-trail-l U* ! *i*) = 0›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-map-lit-of-trail*:
‹*remove-one-annot-true-clause S U* $\Longrightarrow$
map lit-of (*get-trail-l S*) = *map lit-of* (*get-trail-l U*)›
⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-map-lit-of-trail*:
‹*remove-one-annot-true-clause*** *S U* $\Longrightarrow$
map lit-of (*get-trail-l S*) = *map lit-of* (*get-trail-l U*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *RETURN-le-RES-no-return*:
‹*f* ≤ *SPEC* (λ*S*. *g S* ∈ Φ) ⟹ *do* {*S* ← *f*; *RETURN* (*g S*)} ≤ *RES* Φ›
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *valid-trail-reduction-count-dec-ge*:
‹*valid-trail-reduction M M'* ⟹
count-decided M' ≤ *count-decided M*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*(λ*T*. *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*⟩

definition *collect-valid-indices* :: ‹- ⇒ *nat list nres*› **where**
‹*collect-valid-indices S* = *SPEC* (λ*N*. *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* = (λ*S 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* = (λ*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*))›

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* = (λ*S*. *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*(λ*T*. *remove-one-annot-true-clause*$^{++}$ *S T* ∧
    get-trail-l S = *get-trail-l T*))›
⟨*proof*⟩

definition *GC-clauses* :: ‹*nat clauses-l* ⇒ *nat clauses-l* ⇒ (*nat clauses-l* × (*nat* ⇒ *nat option*)) *nres*›
where
‹*GC-clauses N N'* = *do* {
  xs ← *SPEC*(λ*xs*. *set-mset* (*dom-m N*) ⊆ *set xs*);
  (*N*, *N'*, *m*) ← *nfoldli*
    xs
    (λ(*N*, *N'*, *m*). *True*)
    (λ*C* (*N*, *N'*, *m*).
       if C ∈# *dom-m N*
       then do {
         C' ← *SPEC*(λ*i*. *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'*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *GC-remap-ran-m-no-rewrite-map*:
‹*GC-remap* (*old*, *m*, *new*) (*old'*, *m'*, *new'*) ⟹ *C* ∉# *dom-m old* ⟹ *m' C* = *m C*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*))›
⟨*proof*⟩

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'*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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›
⟨proof⟩

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›
⟨proof⟩

lemma *GC-remap-ran-m-no-lost*:
‹GC-remap S S' ⟹ C ∈# dom-m (fst S') ⟹ C ∈# dom-m (fst S)›
⟨proof⟩

lemma *rtranclp-GC-remap-ran-m-no-lost*:
‹GC-remap** S S' ⟹ C ∈# dom-m (fst S') ⟹ C ∈# dom-m (fst S)›
⟨proof⟩

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))›
⟨proof⟩

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))›
⟨proof⟩

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'))›
⟨proof⟩

lemma *rtranclp-GC-remap-ran-m-no-new-map*:
‹GC-remap** S S' ⟹ C ∈# dom-m (fst S') ⟹ C ∈# dom-m (fst S)›
⟨proof⟩

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'›
⟨proof⟩

lemma *rtranclp-GC-remap-learned-clss-l*:
‹GC-remap** (x1a, Map.empty, fmempty) (fmempty, m, x1ad) ⟹ learned-clss-l x1ad = learned-clss-l x1a›
⟨proof⟩

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*⟩

inductive-cases *GC-remapE*: ‹$GC\text{-}remap \ S \ T$›

lemma *rtranclp-GC-remap-finite-map*:
‹$GC\text{-}remap^{**} \ \ S \ S' \Longrightarrow finite \ (dom \ (fst \ (snd \ S))) \Longrightarrow finite \ (dom \ (fst \ (snd \ S')))$›
⟨*proof*⟩

lemma *rtranclp-GC-remap-old-dom-map*:
‹$GC\text{-}remap^{**} \ \ R \ S \Longrightarrow (\bigwedge x. \ x \in\# \ dom\text{-}m \ (fst \ R) \Longrightarrow x \notin dom \ (fst \ (snd \ R))) \Longrightarrow$
$(\bigwedge x. \ x \in\# \ dom\text{-}m \ (fst \ S) \Longrightarrow x \notin dom \ (fst \ (snd \ S)))$›
⟨*proof*⟩

lemma *remap-cons2-rtranclp*:
assumes
‹$(the \circ\circ fst) \ (snd \ R) \ `\# \ mset\text{-}set \ (dom \ (fst \ (snd \ R))) = dom\text{-}m \ (snd \ (snd \ R))$› **and**
‹$\bigwedge x. \ x \in\# \ dom\text{-}m \ (fst \ R) \Longrightarrow x \notin dom \ (fst \ (snd \ R))$› **and**
‹$finite \ (dom \ (fst \ (snd \ R)))$› **and**
st: ‹$GC\text{-}remap^{**} \ R \ S$› **and**
C': ‹$C' \notin\# \ dom\text{-}m \ (snd \ (snd \ S))$› **and**
C: ‹$C \in\# \ dom\text{-}m \ (fst \ S)$›
shows
‹$GC\text{-}remap^{**} \ R \ (fmdrop \ C \ (fst \ S), \ (fst \ (snd \ S))(C \mapsto C'), \ fmupd \ C' \ (the \ (fmlookup \ (fst \ S) \ C)) \ (snd \ (snd \ S)))$›
⟨*proof*⟩

lemma (**in** −) *fmdom-fmrestrict-set*: ‹$fmdrop \ xa \ (fmrestrict\text{-}set \ s \ N) = fmrestrict\text{-}set \ (s - \{xa\}) \ N$›
⟨*proof*⟩

lemma (**in** −) *GC-clauses-GC-remap*:
‹$GC\text{-}clauses \ N \ fmempty \leq SPEC(\lambda(N'', \ m). \ GC\text{-}remap^{**} \ (N, \ Map.empty, \ fmempty) \ (fmempty, \ m, \ N'') \wedge$
$0 \notin\# \ dom\text{-}m \ N'')$›
⟨*proof*⟩

definition *cdcl-GC-clauses-pre* :: ‹$'v \ twl\text{-}st\text{-}l \Rightarrow bool$› **where**
‹$cdcl\text{-}GC\text{-}clauses\text{-}pre \ S \longleftrightarrow ($
$\exists \ T. \ (S, \ T) \in twl\text{-}st\text{-}l \ None \wedge$
$twl\text{-}list\text{-}invs \ S \wedge twl\text{-}struct\text{-}invs \ T \wedge$
$get\text{-}conflict\text{-}l \ S = None \wedge clauses\text{-}to\text{-}update\text{-}l \ S = \{\#\} \wedge$
$count\text{-}decided \ (get\text{-}trail\text{-}l \ S) = 0 \wedge (\forall \ L \in set \ (get\text{-}trail\text{-}l \ S). \ mark\text{-}of \ L = 0)$
$)$›

definition *cdcl-GC-clauses* :: ‹$'v \ twl\text{-}st\text{-}l \Rightarrow \ 'v \ twl\text{-}st\text{-}l \ nres$› **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* (λ*T*. *cdcl-twl-restart-l S T* ∧
get-trail-l S = *get-trail-l T*)›
⟨*proof*⟩

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*(λ*T*. *cdcl-twl-restart-l S T*)›
⟨*proof*⟩

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*⟩

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *all-init-lits-of-wl-all-lits-st*:
‹*set-mset (all-init-lits-of-wl S) ⊆ set-mset (all-lits-st S)*›
⟨*proof*⟩

lemma *in-all-lits-uminus-iff*[*simp*]: ‹*- L ∈# all-lits-st S ⟷ L ∈# all-lits-st S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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 < 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* ≤ *length* (*W K*));
  RETURN (*W* (*K* := *take i* (*W K*)))
}›

lemma shows
op-watch-list:
‹*i* < *length* (*W K*) ⟹ *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* ≤ *length* (*W K*) ⟹ *mop-watch-list-take W K i* ≤ *SPEC*(λ*c*. (*c*, *op-watch-list-take W K i*) ∈ *Id*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)) ∧
(∀ (*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* :: ‹*'v twl-st-wl* ⇒ *bool*› **where**
‹*correct-watching* (*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*).
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 $(\lambda i.\ i \in\#\ dom\text{-}m\ N)$ $(fst\ `\#\ mset\ (W\ L)) = clause\text{-}to\text{-}update\ L\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, \{\#\}))\rangle$

declare *correct-watching.simps*[*simp del*]

lemma *all-lits-st-simps*[*simp*]:
‹*all-lits-st* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(K := WK)) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
‹*all-lits-st* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0,$ *add-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
‹$x1 \in\#\ dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow n' < length\ (x1aa \propto x1) \Longrightarrow$
all-lits-st $(x1b, x1aa(x1 \hookrightarrow$ *WB-More-Refinement-List.swap* $(x1aa \propto x1)\ n\ n'), D, x1c, x1d, NEk, UEk, NS, US, N0, U0, x1e,$
$x2e) =$
all-lits-st
$(x1b, x1aa, D, x1c, x1d, NEk, UEk, NS, US, N0, U0, x1e,$
$x2e)\rangle$
‹*all-lits-st* $(L\ \#\ M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
‹*NO-MATCH* $\{\#\}\ Q \Longrightarrow$ *all-lits-st* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, \{\#\}, W)\rangle$
‹*NO-MATCH* $[]\ M \Longrightarrow$ *all-lits-st* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $([], N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
‹*NO-MATCH None* $D \Longrightarrow$ *all-lits-st* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W) =$
all-lits-st $(M, N,$ *None*, $NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
‹*all-lits-st* $(set\text{-}literals\text{-}to\text{-}update\text{-}wl\ WS\ S) =$ *all-lits-st* $S\rangle$
⟨*proof*⟩

lemma *in-clause-in-all-lits-of-mm*[*simp*]:
‹$x1 \in\#\ dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow$
$x1aa \propto x1\ !\ n \in\#$ *all-lits-st* $(x1b, x1aa, D, x1c, x1d, NS, US, N0, U0, x1e,$
$x2e)\rangle$
⟨*proof*⟩

lemma *correct-watching-except-correct-watching*:
assumes
j: ‹$j \geq length\ (W\ K)\rangle$ **and**
corr: ‹*correct-watching-except* $i\ j\ K\ (M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W)\rangle$
shows ‹*correct-watching* $(M, N, D, NE, UE, NEk, UEk, NS, US, N0, U0, Q, W(K := take\ i\ (W\ K)))\rangle$
⟨*proof*⟩

lemma *length-ge2I*: ‹$x \neq y \Longrightarrow x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow length\ xs \geq 2\rangle$
⟨*proof*⟩

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-lits-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* $j\ (W\ L)).\ i \in\#\ dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge$
$K \neq L \wedge L \in set\ (watched\text{-}l\ (N \propto i)) \wedge length\ (N \propto i) \geq 2 \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i, K, b)) \wedge$

$(\forall (i,\ K,\ b) \in \#\ mset\ (take\ i\ (W\ L)\ @\ drop\ j\ (W\ L)).\ b \longrightarrow i \in\#\ dom\text{-}m\ N) \wedge$
$filter\text{-}mset\ (\lambda i.\ i \in\#\ dom\text{-}m\ N)\ (fst\ `\#\ mset\ (take\ i\ (W\ L)\ @\ drop\ j\ (W\ L))) = clause\text{-}to\text{-}update\ L\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ \{\#\})) \wedge$
$(L \neq K \longrightarrow$
$distinct\text{-}watched\ (W\ L) \wedge$
$((\forall (i,\ K,\ b) \in \#\ mset\ (W\ L).\ i \in\#\ dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \wedge K \neq L \wedge L \in set\ (watched\text{-}l(N \propto i)) \wedge length\ (N \propto i) \geq 2 \wedge correctly\text{-}marked\text{-}as\text{-}binary\ N\ (i,\ K,\ b)) \wedge$
$(\forall (i,\ K,\ b) \in \#\ mset\ (W\ L).\ b \longrightarrow i \in\#\ dom\text{-}m\ N) \wedge$
$filter\text{-}mset\ (\lambda i.\ i \in\#\ dom\text{-}m\ N)\ (fst\ `\#\ mset\ (W\ L)) = clause\text{-}to\text{-}update\ L\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ \{\#\},\ \{\#\})))\rangle$
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *all-lits-of-all-atms-of*: ‹*K* ∈# *all-lits N NUE* ⟷ *K* ∈# $\mathcal{L}_{all}$ (*all-atms N NUE*)›
⟨*proof*⟩

lemma $\mathcal{L}_{all}$-*all-atms-all-lits*: ‹*set-mset* ($\mathcal{L}_{all}$ (*all-atms N NE*)) = *set-mset* (*all-lits N NE*)›
⟨*proof*⟩

definition *blits-in-*$\mathcal{L}_{in}$ :: ‹*'v twl-st-wl* ⇒ *bool*› **where**
‹*blits-in-*$\mathcal{L}_{in}$ *S* = (∀ *L* ∈# *all-lits-st S*. ∀ (*i*, *K*, *b*) ∈ *set* (*watched-by S L*). *K* ∈# *all-lits-st S*)›

definition *literals-are-in-*$\mathcal{L}_{in}$ :: ‹*'v multiset* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S* ≡ (*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-st S*) ∧ *blits-in-*$\mathcal{L}_{in}$ *S*)›

lemma *literals-are-in-*$\mathcal{L}_{in}$*-nth*:

fixes $C :: nat$
assumes *dom*: ‹$C \in\# dom\text{-}m\ (get\text{-}clauses\text{-}wl\ S)$› **and**
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}\ S$›
shows ‹*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}\ (mset\ (get\text{-}clauses\text{-}wl\ S \propto C))$›
⟨*proof*⟩

lemma *literals-are-*$\mathcal{L}_{in}$*-set-mset-*$\mathcal{L}_{all}$[*simp*]:
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}\ S \Longrightarrow set\text{-}mset\ (\mathcal{L}_{all}\ (all\text{-}atms\text{-}st\ S)) = set\text{-}mset\ (\mathcal{L}_{all}\ \mathcal{A})$›
⟨*proof*⟩

lemma *is-*$\mathcal{L}_{all}$*-all-lits-st-*$\mathcal{L}_{all}$[*simp*]:
‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}\ (all\text{-}lits\text{-}st\ S) \Longrightarrow$
$set\text{-}mset\ (\mathcal{L}_{all}\ (all\text{-}atms\text{-}st\ S)) = set\text{-}mset\ (\mathcal{L}_{all}\ \mathcal{A})$›
‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}\ (all\text{-}lits\ N\ NUE) \Longrightarrow$
$set\text{-}mset\ (\mathcal{L}_{all}\ (all\text{-}atms\ N\ NUE)) = set\text{-}mset\ (\mathcal{L}_{all}\ \mathcal{A})$›
‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}\ (all\text{-}lits\ N\ NUE) \Longrightarrow$
$set\text{-}mset\ (\mathcal{L}_{all}\ (atm\text{-}of\ `\#\ all\text{-}lits\ N\ NUE)) = set\text{-}mset\ (\mathcal{L}_{all}\ \mathcal{A})$›
⟨*proof*⟩

lemma *in-set-all-atms-iff*:
‹$y \in\# all\text{-}atms\ bu\ bw \longleftrightarrow$
$y \in atms\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ bu) \vee y \in atms\text{-}of\text{-}mm\ bw$›
⟨*proof*⟩

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:‹$w < length\ (watched\text{-}by\ (a,\ b,\ c,\ d,\ e,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ f,\ g)\ K)$›
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*⟩

lemma *all-lits-swap*[*simp*]:
‹$x1 \in\# dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow n' < length\ (x1aa \propto x1) \Longrightarrow$
all-lits
$(x1aa(x1 \hookrightarrow swap\ (x1aa \propto x1)\ n\ n'))$
$(x1cx1d) =$
all-lits x1aa $(x1cx1d)$›
⟨*proof*⟩

lemma *blits-in-*$\mathcal{L}_{in}$*-propagate*:
‹$x1 \in\# dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow n' < length\ (x1aa \propto x1) \Longrightarrow$
blits-in-$\mathcal{L}_{in}$ $(Propagated\ A\ x1'\ \#\ x1b,\ x1aa$
$(x1 \hookrightarrow swap\ (x1aa \propto x1)\ n\ n'),\ D,\ x1c,\ x1d,$
$NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ add\text{-}mset\ A'\ x1e,\ x2e) \longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ $(x1b,\ x1aa,\ D,\ x1c,\ x1d,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ x1e,\ x2e)$›
‹$x1 \in\# dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1) \Longrightarrow n' < length\ (x1aa \propto x1) \Longrightarrow$
blits-in-$\mathcal{L}_{in}$ $(x1b,\ x1aa$
$(x1 \hookrightarrow swap\ (x1aa \propto x1)\ n\ n'),\ D,\ x1c,\ x1d,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ x1e,\ x2e) \longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ $(x1b,\ x1aa,\ D,\ x1c,\ x1d,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ x1e,\ x2e)$›
‹*blits-in-*$\mathcal{L}_{in}$
$(Propagated\ A\ x1'\ \#\ x1b,\ x1aa,\ D,\ x1c,\ x1d,$
$NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ add\text{-}mset\ A'\ x1e,\ x2e) \longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ $(x1b,\ x1aa,\ D,\ x1c,\ x1d,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ x1e,\ x2e)$›
‹$x1' \in\# dom\text{-}m\ x1aa \Longrightarrow n < length\ (x1aa \propto x1') \Longrightarrow n' < length\ (x1aa \propto x1') \Longrightarrow$
$K \in\# all\text{-}lits\text{-}st\ (x1b,\ x1aa,\ D,\ x1c,\ x1d,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ x1e,\ x2e) \Longrightarrow$ *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)›
⟨proof⟩

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⟩

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'')]))›
⟨proof⟩

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, {#}, {#})›
⟨proof⟩

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)))›
⟨proof⟩

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 = (λS 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›
⟨*proof*⟩

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*⟩

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 = {#}› **and**
‹L ≠ None ⟹ get-conflict-wl S ≠ None ⟹ clauses-to-update-l T = {#}› **and**
‹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› **and**
‹get-unit-learned-clss-l T = get-unit-learned-clss-wl S› **and**
‹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›
⟨proof⟩

lemma [twl-st-wl]:
‹(a, a′) ∈ state-wl-l None ⟹
get-learned-clss-l a′ = get-learned-clss-wl a›
⟨proof⟩

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›
⟨proof⟩

lemma correct-watching-set-literals-to-update[simp]:
‹correct-watching (set-literals-to-update-wl WS T′) = correct-watching T′›
⟨proof⟩

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›
⟨proof⟩

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›
⟨proof⟩

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 = (λC (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))›
⟨proof⟩

lemma *state-wl-l-mark-of-is-proped*:
‹$(x,\ y) \in$ *state-wl-l* $b \Longrightarrow$
get-trail-wl $x \neq [] \Longrightarrow$
is-proped (*hd* (*get-trail-l* y)) = *is-proped* (*hd* (*get-trail-wl* x))›
⟨*proof*⟩

We here also update the list of watched clauses *WL*.

declare *twl-st-wl*[*simp*]

lemma
assumes
x2-T: ‹$(x2,\ T) \in$ *state-wl-l* b› **and**
struct: ‹*twl-struct-invs* U› **and**
T-U: ‹$(T,\ U) \in$ *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* $\Longrightarrow$ *literals-are-in-*$\mathcal{L}_{in}$*-trail* $\mathcal{A}_{in}$ (*get-trail-wl x2*)›
(**is** ‹-$\Longrightarrow$ *?trail*›) **and**
literals-are-$\mathcal{L}_{in}$*-literals-are-in-*$\mathcal{L}_{in}$*-conflict*:
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ *x2* $\Longrightarrow$ *get-conflict-wl x2* $\neq$ *None* $\Longrightarrow$ *literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (*the* (*get-conflict-wl x2*))› **and**
conflict-not-tautology:
‹*get-conflict-wl x2* $\neq$ *None* $\Longrightarrow$ ¬*tautology* (*the* (*get-conflict-wl x2*))›
⟨*proof*⟩

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') ⟷
$M = M' \wedge N = N' \wedge NE = NE' \wedge UE = UE' \wedge NEk = NEk' \wedge UEk = UEk' \wedge NS = NS' \wedge US = US' \wedge$
$N0 = N0' \wedge U0 = U0' \wedge WS = WS' \wedge Q = Q'$›

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') ⟷
$N = N' \wedge D = D' \wedge NE = NE' \wedge NEk = NEk' \wedge UEk = UEk' \wedge NS = NS' \wedge US = US' \wedge UE = UE' \wedge$
$N0 = N0' \wedge U0 = U0' \wedge WS = WS' \wedge Q = Q'$›

lemma *equality-except-conflict-wl-get-clauses-wl*:
‹*equality-except-conflict-wl S Y* $\Longrightarrow$ *get-clauses-wl S* = *get-clauses-wl Y*› **and**
equality-except-conflict-wl-get-trail-wl:
‹*equality-except-conflict-wl S Y* $\Longrightarrow$ *get-trail-wl S* = *get-trail-wl Y*› **and**
equality-except-trail-wl-get-conflict-wl:
‹*equality-except-trail-wl S Y* $\Longrightarrow$ *get-conflict-wl S* = *get-conflict-wl Y*› **and**
equality-except-trail-wl-get-clauses-wl:
‹*equality-except-trail-wl S Y*$\Longrightarrow$ *get-clauses-wl S* = *get-clauses-wl Y*› **and**
equality-except-trail-wl-get-clauses0-wl:
‹*equality-except-trail-wl S Y*$\Longrightarrow$ *get-clauses0-wl S* = *get-clauses0-wl Y*›
⟨*proof*⟩

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*⟩

definition *propagate-lit-wl-general* :: ‹'*v literal* ⇒ *nat* ⇒ *nat* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*propagate-lit-wl-general* = (λ*L' 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* = (λ*L' 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* = (λ*L' 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* = (λ*L 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* = (λ*L 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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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* ≤ *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*⟩

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*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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′)›
⟨*proof*⟩

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›
  ⟨proof⟩


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⟩
```

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*⟩

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*›
⟨*proof*⟩

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(λc. correct-watching-except j w L c)*›
⟨*proof*⟩

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) ← 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 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
        }
      }
    }
  }
}›
⟨proof⟩

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
            }
        }
      }
    }
  }›
⟨proof⟩
```

lemma *find-unwatched-l-itself*:
‹(*uncurry2 find-unwatched-l*, *uncurry2 find-unwatched-l*) ∈ *Id* ×$_f$ *Id* ×$_f$ *Id* →$_f$ ⟨⟨*Id*⟩*option-rel*⟩*nres-rel*›
⟨*proof*⟩

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*⟩

lemma *nres-add-unrelated*:
‹*P* ≤ ⇓ {(*S*, *S'*). *R1 S*} *Q* ⟹ *P* ≤ ⇓ {(*S*, *S'*). *R2 S S'*} *Q* ⟹ *P* ≤ ⇓{(*S*, *S'*). *R1 S* ∧ *R2 S S'*} *Q*› **for** *P R1 R2 Q*
⟨*proof*⟩

lemma *nres-add-unrelated2*:
‹*P* ≤ *SPEC R1* ⟹ *P* ≤ ⇓ {(*S*, *S'*). *R2 S S'*} *Q* ⟹ *P* ≤ ⇓{(*S*, *S'*). *R1 S* ∧ *R2 S S'*} *Q*› **for** *P R1 R2 Q*
⟨*proof*⟩

lemma *nres-add-unrelated3*:
‹*P* ≤ ⇓ {(*S*, *S'*). *R1 S*} *Q* ⟹ *P* ≤ ⇓ {(*S*, *S'*). *R2 S*} *Q* ⟹ *P* ≤ ⇓{(*S*, *S'*). *R1 S* ∧ *R2 S*} *Q*› **for** *P R1 R2 Q*
⟨*proof*⟩

lemma *blits-in-*$\mathcal{L}_{in}$*-keep-watch2*: ‹*w* < *length* (*watched-by S L*) ⟹ *blits-in-*$\mathcal{L}_{in}$ *S* ⟹ *blits-in-*$\mathcal{L}_{in}$ (*keep-watch L j w S*)›
⟨*proof*⟩

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'* ≠ {#} ∨ *0* < *n*);
ASSERT (*unit-propagation-inner-loop-l-inv L* (*S'*, *n*));
let - = ();
b ← *SPEC* (λ*b*. (*b* ⟶ *0* < *n*) ∧ (¬ *b* ⟶ *clauses-to-update-l S'* ≠ {#}));
let S' = *S'*;
let b = *b*;
if ¬ *b*
then do {

```
      ASSERT (clauses-to-update-l S' ≠ {#});
      X2 ← select-from-clauses-to-update S';
      ASSERT (unit-propagation-inner-loop-body-l-inv L (snd X2) (fst X2));
      x ← SPEC (λK. K ∈ set (get-clauses-l (fst X2) ∝ 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⟩

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'›
  ⟨proof⟩

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›
  ⟨proof⟩


lemma all-lits-st-keep-watch[simp]: ‹all-lits-st (keep-watch L j w S) = all-lits-st S›
  ⟨proof⟩

lemma
  fixes S :: ‹'v twl-st-wl› and S' :: ‹'v twl-st-l› and L :: ‹'v literal› and w :: nat
```

defines [*simp*]: ‹$C' \equiv fst\ (watched\text{-}by\ S\ L\ !\ w)$›
defines
[*simp*]: ‹$T \equiv remove\text{-}one\text{-}lit\text{-}from\text{-}wq\ C'\ S'$›

defines
[*simp*]: ‹$C'' \equiv get\text{-}clauses\text{-}l\ S' \propto C'$›
assumes
S-S': ‹$(S,\ S') \in state\text{-}wl\text{-}l\ (Some\ (L,\ w))$› **and**
w-le: ‹$w < length\ (watched\text{-}by\ S\ L)$› **and**
j-w: ‹$j \leq w$› **and**
corr-w: ‹$correct\text{-}watching\text{-}except\ j\ w\ L\ S$› **and**
blit-in-lit: ‹$blits\text{-}in\text{-}\mathcal{L}_{in}\ S$› **and**
inner-loop-inv: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L\ (j,\ w,\ S)$› **and**
n: ‹$n = size\ (filter\text{-}mset\ (\lambda(i,\ \text{-}).\ i \notin\# dom\text{-}m\ (get\text{-}clauses\text{-}wl\ S))\ (mset\ (drop\ w\ (watched\text{-}by\ S\ L))))$›
and
confl-S: ‹$get\text{-}conflict\text{-}wl\ S = None$›
shows *unit-propagation-inner-loop-body-wl-int-spec*: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}wl\text{-}int\ L\ j\ w\ S \leq$
$\Downarrow\{((i,\ j,\ T'),\ (T,\ n)).$
$(T',\ T) \in state\text{-}wl\text{-}l\ (Some\ (L,\ j)) \wedge$
$correct\text{-}watching\text{-}except\ i\ j\ L\ T' \wedge$
$blits\text{-}in\text{-}\mathcal{L}_{in}\ T' \wedge$
$j \leq length\ (watched\text{-}by\ T'\ L) \wedge$
$length\ (watched\text{-}by\ S\ L) = length\ (watched\text{-}by\ T'\ L) \wedge$
$i \leq j \wedge$
$(get\text{-}conflict\text{-}wl\ T' = None \longrightarrow$
$n = size\ (filter\text{-}mset\ (\lambda(i,\ \text{-}).\ i \notin\# dom\text{-}m\ (get\text{-}clauses\text{-}wl\ T'))\ (mset\ (drop\ j\ (watched\text{-}by\ T'\ L)))))) \wedge$
$(get\text{-}conflict\text{-}wl\ T' \neq None \longrightarrow n = 0)\}$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L\ (S',\ n))$› (**is** ‹*?propa*› **is** ‹- $\leq \Downarrow$ *?unit* -›)**and**
unit-propagation-inner-loop-body-wl-update:
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}inv\ L\ C'\ T \Longrightarrow$
$mset\ `\#\ (ran\text{-}mf\ ((get\text{-}clauses\text{-}wl\ S)\ (C' \hookrightarrow (swap\ (get\text{-}clauses\text{-}wl\ S \propto C')\ 0$
$(1 - (if\ (get\text{-}clauses\text{-}wl\ S) \propto C' !\ 0 = L\ then\ 0\ else\ 1))))))) =$
$mset\ `\#\ (ran\text{-}mf\ (get\text{-}clauses\text{-}wl\ S))$› (**is** ‹- $\Longrightarrow$ *?eq*›)
⟨*proof*⟩

lemma
fixes S :: ‹$'v\ twl\text{-}st\text{-}wl$› **and** S' :: ‹$'v\ twl\text{-}st\text{-}l$› **and** L :: ‹$'v\ literal$› **and** w :: *nat*
defines [*simp*]: ‹$C' \equiv fst\ (watched\text{-}by\ S\ L\ !\ w)$›
defines
[*simp*]: ‹$T \equiv remove\text{-}one\text{-}lit\text{-}from\text{-}wq\ C'\ S'$›

defines
[*simp*]: ‹$C'' \equiv get\text{-}clauses\text{-}l\ S' \propto C'$›
assumes
S-S': ‹$(S,\ S') \in state\text{-}wl\text{-}l\ (Some\ (L,\ w))$› **and**
w-le: ‹$w < length\ (watched\text{-}by\ S\ L)$› **and**
j-w: ‹$j \leq w$› **and**
corr-w: ‹$correct\text{-}watching\text{-}except\ j\ w\ L\ S$› **and**
blit-in-lit: ‹$blits\text{-}in\text{-}\mathcal{L}_{in}\ S$› **and**
inner-loop-inv: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L\ (j,\ w,\ S)$› **and**
n: ‹$n = size\ (filter\text{-}mset\ (\lambda(i,\ \text{-}).\ i \notin\# dom\text{-}m\ (get\text{-}clauses\text{-}wl\ S))\ (mset\ (drop\ w\ (watched\text{-}by\ S\ L))))$›
and
confl-S: ‹$get\text{-}conflict\text{-}wl\ S = None$›
shows *unit-propagation-inner-loop-body-wl-spec*: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}wl\ L\ j\ w\ S \leq$

⇓{((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))›
⟨proof⟩

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)))›
⟨proof⟩

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⟩

lemma *unit-propagation-inner-loop-wl-loop-alt-def*:
‹unit-propagation-inner-loop-wl-loop L S_0 = do {
ASSERT(L ∈# all-lits-st S_0);
let (- :: nat) = (if get-conflict-wl S_0 = None then remaining-nondom-wl 0 L S_0 else 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)
}
›
⟨proof⟩

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* S_0 = *do* {
(*j*, *w*, *S*) ← *unit-propagation-inner-loop-wl-loop L* S_0;
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*›
⟨*proof*⟩

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-*$\mathcal{L}_{in}$ *T'*} →
⟨{(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'* ∧ *blits-in-*$\mathcal{L}_{in}$ *T'*}⟩ *nres-rel*
› (**is** ‹*?fg* ∈ *?A* → ⟨*?B*⟩*nres-rel*› **is** ‹*?fg* ∈ *?A* → ⟨{(*T'*, *T*). - ∧ *?P T T'*}⟩*nres-rel*›)
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

6.5.3 Decide or Skip

definition *find-unassigned-lit-wl* :: ‹'*v twl-st-wl* ⇒ ('*v twl-st-wl* × '*v literal option*) *nres*› **where**
‹*find-unassigned-lit-wl* = (λ*S*.
SPEC (λ(*T*, *L*). *T* = *S* ∧
(*L* ≠ *None* ⟶
undefined-lit (*get-trail-wl S*) (*the L*) ∧
the L ∈# *all-lits-st S*) ∧
(*L* = *None* ⟶ (∄ *L'*. *undefined-lit* (*get-trail-wl S*) *L'* ∧
L' ∈# *all-lits-st S*)))
)›

definition *decide-wl-or-skip-pre* **where**
‹*decide-wl-or-skip-pre S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧
decide-l-or-skip-pre S' ∧ *blits-in-$\mathcal{L}_{in}$ S*
)›

definition *decide-lit-wl* :: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl*› **where**
‹*decide-lit-wl* = (λ*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* ⇒ (*bool* × '*v twl-st-wl*) *nres*› **where**
‹*decide-wl-or-skip S* = (*do* {
ASSERT(*decide-wl-or-skip-pre S*);
(*S*, *L*) ← *find-unassigned-lit-wl S*;
case L of
None ⇒ *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-$\mathcal{L}_{in}$ T'* ∧
get-conflict-wl T' = *None*} →
⟨{((*b'*, *T'*), (*b*, *T*)). *b'* = *b* ∧
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T' ∧ *blits-in-$\mathcal{L}_{in}$ T'*}⟩*nres-rel*›
⟨*proof*⟩

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* = (λ*S*. *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* = (λ*L 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* = (λ*L 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 S₀ =
    do {
      ASSERT(get-conflict-wl S₀ ≠ None);
      (-, S) ←
        WHILE_T^λ(brk, S). skip-and-resolve-loop-wl-inv S₀ 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, S₀);
      RETURN S
    }
  ›

lemma tl-state-wl-tl-state-l:
  ‹(S, S') ∈ state-wl-l None ⟹ correct-watching S ⟹  blits-in-ℒ_in S ⟹
     mop-tl-state-wl S ≤ ⇓(bool-rel ×_f {(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching S ∧
blits-in-ℒ_in S}) (mop-tl-state-l S')›
  ⟨proof⟩


lemma mop-update-confl-tl-wl-mop-update-confl-tl-l:
  ‹(S, S') ∈ state-wl-l None ⟹ correct-watching S ⟹  blits-in-ℒ_in S ⟹
   ((L, C), (L', C')) ∈ Id ⟹
    mop-update-confl-tl-wl L C S ≤
```

$\Downarrow$(*bool-rel* $\times_f$ {(*S*, *T*). (*S*, *T*) $\in$ *state-wl-l None* $\wedge$ *correct-watching S* $\wedge$ *blits-in-*$\mathcal{L}_{in}$ *S*})
(*mop-update-confl-tl-l L′ C′ S′*)⟩
⟨*proof*⟩

lemma *mop-hd-trail-wl-mop-hd-trail-l*:
⟨(*S*, *S′*) $\in$ *state-wl-l None* $\Longrightarrow$ *correct-watching S* $\Longrightarrow$ *blits-in-*$\mathcal{L}_{in}$ *S* $\Longrightarrow$
mop-hd-trail-wl S
$\leq \Downarrow$ (*Id*)
(*mop-hd-trail-l S′*)⟩
⟨*proof*⟩

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′*)⟩
⟨*proof*⟩

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′*)⟩
⟨*proof*⟩

lemma *skip-and-resolve-loop-wl-spec*:
⟨(*skip-and-resolve-loop-wl*, *skip-and-resolve-loop-l*)
$\in$ {(*T′*::*′v 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′*)} $\rightarrow$
⟨{(*T′*, *T*).
(*T′*, *T*) $\in$ *state-wl-l None* $\wedge$
correct-watching T′ $\wedge$ *blits-in-*$\mathcal{L}_{in}$ *T′*}⟩*nres-rel*⟩
(**is** ⟨*?s* $\in$ *?A* $\rightarrow$ ⟨*?B*⟩*nres-rel*⟩)
⟨*proof*⟩

6.5.5 Backtrack

definition *find-decomp-wl* :: ⟨*′v literal* $\Rightarrow$ *′v twl-st-wl* $\Rightarrow$ *′v twl-st-wl nres*⟩ **where**
⟨*find-decomp-wl* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*).
SPEC(λ*S*. $\exists$ *K M2 M1*. *S* = (*M1*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) $\wedge$
(*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition M*) $\wedge$
get-level M K = *get-maximum-level M* (*the D* − {#−*L*#}) + *1*))⟩

definition *find-lit-of-max-level-wl* :: ⟨*′v twl-st-wl* $\Rightarrow$ *′v literal* $\Rightarrow$ *′v literal nres*⟩ **where**
⟨*find-lit-of-max-level-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *Q*, *W*) *L*.
SPEC(λ*L′*. *L′* $\in$# *remove1-mset* (−*L*) (*the D*) $\wedge$ *get-level M L′* = *get-maximum-level M* (*the D* − {#−*L*#}))⟩

fun *extract-shorter-conflict-wl* :: ⟨*′v twl-st-wl* $\Rightarrow$ *′v twl-st-wl nres*⟩ **where**
⟨*extract-shorter-conflict-wl* (*M*, *N*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) = *SPEC*(λ*S*.
$\exists$ *D′*. *D′* $\subseteq$# *the D* $\wedge$ *S* = (*M*, *N*, *Some D′*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *W*) $\wedge$
clause ‘# *twl-clause-of* ‘# *ran-mf N* + *NE* + *NEk* + *UE* + *UEk* + *NS* + *US* $\models$*pm D′* $\wedge$ −*lit-of* (*hd M*) $\in$# *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*(λ*i*. *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 (λ*E*. *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* = (λ*L 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*))›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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''*)]))›
  ⟨*proof*⟩

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*⟩

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*›
  ⟨*proof*⟩

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*⟩

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*$_0$ ≡ λ(*brk*, *T*).
    (∃ *T'* *S*$_0$'. (*T*, *T'*) ∈ *state-wl-l None* ∧
    (*S*$_0$, *S*$_0$') ∈ *state-wl-l None* ∧
    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*) ← *WHILE*$_T$*cdcl-twl-stgy-prog-wl-inv S*$_0$
      (λ(*brk*, -). ¬*brk*)
      (λ(*brk*, *S*). *do* {
        T ← *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*⟩

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*⟩

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⟩

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 ← 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);
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⟩

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›)
⟨proof⟩

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⟩

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 {

$T \leftarrow$ *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);
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-*$\mathcal{L}_{in}$ S} →$_f$
⟨*bool-rel* ×$_r$ {(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*⟩

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-*$\mathcal{L}_{in}$ S} →$_f$
⟨*bool-rel* ×$_r$ *state-wl-l None*⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* →$_f$ ⟨*?B*⟩ *nres-rel*›)
⟨*proof*⟩

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*⟩

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 = {}›
⟨*proof*⟩

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›
⟨*proof*⟩

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 (A − {#L, L′#}))
(λa. if a = L then [] else W a) ⟷
xa ∈ clauses-pointed-to (set-mset (A − {#L, L′#})) W›
‹∀ L∈set (W L). fst L ∉# dom-m aa ⟹ xa ∈# dom-m aa ⟹
xa ∈ clauses-pointed-to (set-mset (A − {#L′, L#}))
(λa. if a = L then [] else W a) ⟷
xa ∈ clauses-pointed-to (set-mset (A − {#L′, L#})) W›
⟨proof⟩

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 (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›
⟨proof⟩

lemma *negs-remove-Neg*: ‹A ∉# A ⟹ negs A + poss A − {#Neg A, Pos A#} =
negs A + poss A›
⟨proof⟩
lemma *poss-remove-Pos*: ‹A ∉# A ⟹ negs A + poss A − {#Pos A, Neg A#} =
negs A + poss A›
⟨proof⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

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›

⟨*proof*⟩

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*›
⟨*proof*⟩

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 = {#}›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-inprocessing-l-cdcl-twl-l-inp*:
‹*cdcl-twl-inprocessing-l*** *S T* ⟹ *cdcl-twl-restart-l-inp*** *S T*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma ‹*size* (*get-learned-clss-l T*) = *size* (*learned-clss-l* (*get-clauses-l T*))›
⟨*proof*⟩

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*⟩

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 S*$_0$ ≡ (λ(*brk*, *T*, *last-GC*, *last-Restart*, *n*).
  (∃ *S*$_0$' *T'* *n'*.
    (*T*, *T'*) ∈ *twl-st-l None* ∧
    (*S*$_0$, *S*$_0$') ∈ *twl-st-l None* ∧
    cdcl-twl-stgy-restart-prog-inv (*S*$_0$', *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 S*$_0$ =
do {
  (*brk*, *T*, -, -, -) ← *WHILE*$_T$$^{cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0}$
    (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*, *S*$_0$, *size* (*get-all-learned-clss-l S*$_0$), *size* (*get-all-learned-clss-l S*$_0$), *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*›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-abs-l-cdcl-twl-stgy-restart-abs-l*:
‹(*cdcl-twl-stgy-restart-abs-l*, *cdcl-twl-stgy-restart-prog*) ∈

$\{(S :: 'v\ twl\text{-}st\text{-}l,\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S \wedge$
$clauses\text{-}to\text{-}update\text{-}l\ S\ = \{\#\}\} \rightarrow_f$
$\langle\{(S,\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S\}\rangle\ nres\text{-}rel\rangle$
$\langle proof\rangle$

end

lemma *cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l*:
assumes
ST: ‹$(S,\ S') \in twl\text{-}st\text{-}l\ None$› **and**
list-invs: ‹$twl\text{-}list\text{-}invs\ S$› **and**
struct-invs: ‹$twl\text{-}struct\text{-}invs\ S'$› **and**
confl: ‹$get\text{-}conflict\text{-}l\ S = None$› **and**
upd: ‹$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}$› **and**
stgy-invs: ‹$twl\text{-}stgy\text{-}invs\ S'$› **and**
abs-pre: ‹$restart\text{-}prog\text{-}pre\ S'\ last\text{-}GC\ last\text{-}Restart\ brk$›
shows ‹$cdcl\text{-}twl\text{-}full\text{-}restart\text{-}l\text{-}GC\text{-}prog\ S \leq \Downarrow Id\ (SPEC\ (\lambda T.\ cdcl\text{-}twl\text{-}restart\text{-}l\ S\ T))$›
$\langle proof\rangle$

lemma *cdcl-twl-full-restart-inprocess-l-cdcl-twl-restart-l*:
assumes
ST: ‹$(S,\ S') \in twl\text{-}st\text{-}l\ None$› **and**
list-invs: ‹$twl\text{-}list\text{-}invs\ S$› **and**
struct-invs: ‹$twl\text{-}struct\text{-}invs\ S'$› **and**
confl: ‹$get\text{-}conflict\text{-}l\ S = None$› **and**
upd: ‹$clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}$› **and**
stgy-invs: ‹$twl\text{-}stgy\text{-}invs\ S'$› **and**
abs-pre: ‹$restart\text{-}prog\text{-}pre\ S'\ last\text{-}GC\ last\text{-}Restart\ brk$›
shows ‹$cdcl\text{-}twl\text{-}full\text{-}restart\text{-}inprocess\text{-}l\ S \leq \Downarrow Id\ (SPEC\ (\lambda T.\ cdcl\text{-}twl\text{-}restart\text{-}l\text{-}inp^{**}\ S\ T \wedge count\text{-}decided\ (get\text{-}trail\text{-}l\ T) = 0))$› (**is** ‹$- \leq \Downarrow - ?P$›)
$\langle proof\rangle$

context *twl-restart-ops*
begin

definition *restart-prog-l*
$:: 'v\ twl\text{-}st\text{-}l \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool \Rightarrow ('v\ twl\text{-}st\text{-}l \times nat \times nat \times nat)\ nres$
where
‹$restart\text{-}prog\text{-}l\ S\ last\text{-}GC\ last\text{-}Restart\ n\ brk = do\ \{$
$ASSERT(restart\text{-}abs\text{-}l\text{-}pre\ S\ last\text{-}GC\ last\text{-}Restart\ brk);$
$b \leftarrow GC\text{-}required\text{-}l\ S\ last\text{-}GC\ n;$
$b2 \leftarrow restart\text{-}required\text{-}l\ S\ last\text{-}Restart\ n;$
$if\ b2 \wedge \neg brk\ then\ do\ \{$
$T \leftarrow cdcl\text{-}twl\text{-}restart\text{-}l\text{-}prog\ S;$
$RETURN\ (T,\ last\text{-}GC,\ size\ (get\text{-}all\text{-}learned\text{-}clss\text{-}l\ T),\ n)$
$\}$
$else\ if\ b \wedge \neg brk\ then\ do\ \{$
$inp \leftarrow inprocessing\text{-}required\text{-}l\ S;$
$if\ \neg inp\ then\ do\ \{$
$b \leftarrow SPEC(\lambda\text{-}.\ True);$
$T \leftarrow (if\ b\ then\ cdcl\text{-}twl\text{-}full\text{-}restart\text{-}l\text{-}prog\ S\ else\ cdcl\text{-}twl\text{-}full\text{-}restart\text{-}l\text{-}GC\text{-}prog\ S);$
$RETURN\ (T,\ size\ (get\text{-}all\text{-}learned\text{-}clss\text{-}l\ T),\ size\ (get\text{-}all\text{-}learned\text{-}clss\text{-}l\ T),\ n + 1)$
$\}$

```
    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)
}›
⟨proof⟩

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⟩


lemma restart-prog-l-restart-abs-l2:
  ‹(uncurry4 restart-prog-l, uncurry4 restart-abs-l)
  ∈ Id ×f nat-rel ×f nat-rel×f nat-rel ×f bool-rel →f
    ⟨Id ×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⟩

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 S_0 =
 do {
   ebrk ← RES UNIV;
   (ebrk, brk, T, 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);
   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 S_0 =
 do {
   (brk, T, last-GC, last-Restart, n) ← WHILE_T^{cdcl-twl-stgy-restart-abs-l-inv S_0}
     (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, S_0, size (get-all-learned-clss-l S_0), size (get-all-learned-clss-l S_0), 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 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, 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, 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, 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⟩

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⟩

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›
  ⟨proof⟩
```

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*) ← $WHILE_T^{cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ o\ snd}$
(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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

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*)
∈ {(*S*, *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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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›
⟨*proof*⟩

lemma *all-init-lits-of-wl-simps*[*simp*]:

‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \neg \mathit{irred}\ N\ C \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ \mathit{fmdrop}\ C\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{NO\text{-}MATCH}\ \{\#\}\ US \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ \{\#\},\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{NO\text{-}MATCH}\ []\ M \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ ([],\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{irred}\ N\ C \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ \mathit{fmdrop}\ C\ N,\ D,\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C))\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ \mathit{add\text{-}mset}\ E\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{NO\text{-}MATCH}\ \{\#\}\ UEk \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ \{\#\},\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{NO\text{-}MATCH}\ \{\#\}\ U0 \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ \{\#\},\ Q,\ W)$›
‹$\mathit{NO\text{-}MATCH}\ \{\#\}\ UE \Longrightarrow$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ \{\#\},\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
⟨*proof*⟩

lemma *all-learned-lits-of-wl-simps*[*simp*]:
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{irred}\ N\ C \Longrightarrow$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ \mathit{fmdrop}\ C\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›

‹$\mathit{NO\text{-}MATCH}\ []\ M \Longrightarrow$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ ([],\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ \mathit{add\text{-}mset}\ E\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ \mathit{add\text{-}mset}\ E\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \neg \mathit{irred}\ N\ C \Longrightarrow$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ \mathit{fmdrop}\ C\ N,\ D,\ NE,\ \mathit{add\text{-}mset}\ (\mathit{mset}\ (N \propto C))\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) =$
$\mathit{all\text{-}learned\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W)$›
⟨*proof*⟩

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**
‹$\mathit{correct\text{-}watching'}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W) \longleftrightarrow$
$(\forall\, L \in\# \mathit{all\text{-}init\text{-}lits\text{-}of\text{-}wl}\ (M,\ N,\ D,\ NE,\ UE,\ NEk,\ UEk,\ NS,\ US,\ N0,\ U0,\ Q,\ W).$
$\mathit{distinct\text{-}watched}\ (W\ L) \wedge$
$(\forall\, (i,\ K,\ b) \in\# \mathit{mset}\ (W\ L).$
$i \in\# \mathit{dom\text{-}m}\ N \longrightarrow K \in \mathit{set}\ (N \propto i) \wedge K \neq L \wedge \mathit{correctly\text{-}marked\text{-}as\text{-}binary}\ N\ (i,\ K,\ b)) \wedge$
$(\forall\, (i,\ K,\ b) \in\# \mathit{mset}\ (W\ L).$
$b \longrightarrow i \in\# \mathit{dom\text{-}m}\ N) \wedge$

filter-mset ($\lambda i.\ i \in\#$ *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) ⟷
(∀ $L \in\#$ *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* ($W\ L$).
$i \in\#$ *dom-m* $N \longrightarrow K \in$ *set* ($N \propto i$) ∧ $K \neq L$ ∧ *correctly-marked-as-binary* N (i, K, b)) ∧
filter-mset ($\lambda i.\ i \in\#$ *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'* $S \Longrightarrow$ *correct-watching'* S›
⟨*proof*⟩

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* $S \equiv$
set-mset (*dom-m* (*get-clauses-wl* S))
⊆ *clauses-pointed-to* (*set-mset* (*all-init-lits-of-wl* S)) (*get-watched-wl* S) ∧
(∀ $L \in\#$ *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* $S \equiv$
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 ⟷
(∀ $L \in\#$ *all-init-lits-of-wl* S.
∀ (i, K, b) ∈ *set* (*watched-by* $S\ L$). $K \in\#$ *all-init-lits-of-wl* S)›

definition *literals-are-*$\mathcal{L}_{in}$' :: ‹'*v twl-st-wl* ⇒ *bool*› **where**
‹*literals-are-*$\mathcal{L}_{in}$' $S \equiv$
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* $S \equiv$ *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›
⟨*proof*⟩

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)›

⟨*proof*⟩

lemma *literals-are-$\mathcal{L}_{in}$-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B} \Longrightarrow$ *literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ *S* = *literals-are-$\mathcal{L}_{in}$* $\mathcal{B}$ *S*›
⟨*proof*⟩

lemma *all-learned-lits-of-wl-all-lits-st*:
‹*set-mset* (*all-learned-lits-of-wl S*) ⊆ *set-mset* (*all-lits-st S*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *$\mathcal{L}_{all}$-all-atms*:
‹*set-mset* (*$\mathcal{L}_{all}$* (*all-atms-st S*)) = *set-mset* (*all-lits-st S*)›
⟨*proof*⟩

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*⟩

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-$\mathcal{L}_{in}$'* *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*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
∈ {(*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*⟩

definition *cdcl-twl-full-restart-wl-GC-prog-post* :: ‹*'v twl-st-wl ⇒ 'v twl-st-wl ⇒ bool*› **where**
‹*cdcl-twl-full-restart-wl-GC-prog-post 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'* ∧ *correct-watching' T* ∧
set-mset (*all-init-lits-of-wl T*) =
set-mset (*all-lits-st T*) ∧
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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma (**in** −)[*twl-st*,*simp*]: ‹*learned-clss* (*state*$_W$*-of S*) = *get-all-learned-clss S*›
⟨*proof*⟩

lemma (**in** −)[*twl-st*,*simp*]: ‹*init-clss* (*state*$_W$*-of S*) = *get-all-init-clss S*›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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))›
⟨*proof*⟩

lemma *cdcl-twl-local-restart-wl-spec0-cdcl-twl-local-restart-l-spec0*:
‹*(x, y) ∈ {(S, S′). (S, S′) ∈ state-wl-l None ∧ correct-watching′ S ∧ literals-are-$\mathcal{L}_{in}$′ S} ⟹*
cdcl-twl-local-restart-wl-spec0 x
≤ ⇓ {(S, S′). (S, S′) ∈ state-wl-l None ∧ correct-watching′ S ∧ literals-are-$\mathcal{L}_{in}$′ S}
(cdcl-twl-local-restart-l-spec0 y)›
⟨*proof*⟩

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) ∈ {(S, S′). (S, S′) ∈ state-wl-l None ∧ correct-watching′ S ∧ literals-are-$\mathcal{L}_{in}$′ S}*›
shows ‹*(V, Va) ∈ {(S, S′). (S, S′) ∈ state-wl-l None ∧ correct-watching S ∧ blits-in-$\mathcal{L}_{in}$ S}*› **and**
‹*set-mset (all-init-lits-of-wl V) = set-mset (all-lits-st V)*›
⟨*proof*⟩

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*)›
⟨*proof*⟩
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** → *60*)

lemma *GC-remap-all-init-atmsD*:
‹*GC-remap* (*N*, *x*, *m*) (*N'*, *x'*, *m'*) ⟹
all-init-atms N NE + *all-init-atms m NE* = *all-init-atms N' NE* + *all-init-atms m' NE*›
⟨*proof*⟩

lemma *rtranclp-GC-remap-all-init-atmsD*:
‹*GC-remap*** (*N*, *x*, *m*) (*N'*, *x'*, *m'*) ⟹
all-init-atms N NE + *all-init-atms m NE* = *all-init-atms N' NE* + *all-init-atms m' NE*›
⟨*proof*⟩

lemma *rtranclp-GC-remap-all-init-atms*:
‹*GC-remap*** (*x1a*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *x1ad*) ⟹
all-init-atms x1ad NE = *all-init-atms x1a NE*›
⟨*proof*⟩

lemma *GC-remap-all-init-lits*:
‹*GC-remap* (*N*, *m*, *new*) (*N'*, *m'*, *new'*) ⟹
all-init-lits N NE + *all-init-lits new NE* = *all-init-lits N' NE* + *all-init-lits new' NE*›
⟨*proof*⟩

lemma *rtranclp-GC-remap-all-init-lits*:
‹*GC-remap*** (*N*, *m*, *new*) (*N'*, *m'*, *new'*) ⟹
all-init-lits N NE + *all-init-lits new NE* = *all-init-lits N' NE* + *all-init-lits new' NE*›
⟨*proof*⟩

lemma *subsumed-clauses-alt-def*:
‹*subsumed-clauses S* = *subsumed-init-clauses S* + *subsumed-learned-clauses S*›
⟨*proof*⟩

definition *drop-clause-add-move-init-wl* :: ‹'*v twl-st-wl* ⇒ *nat* ⇒ '*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* ∝ *C*)) *NE0*, *UE*, *NEk*, *UEk*, *NS*, {#}, *N0*, *U0*, *Q*, *W*))›

definition *drop-clause-wl* :: ‹'*v twl-st-wl* ⇒ *nat* ⇒ '*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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*))›
⟨*proof*⟩

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*⟩

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*⟩

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-*$\mathcal{L}_{in}$*′ 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-$\mathcal{L}_{in}$' 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-$\mathcal{L}_{in}$' S*} ⟹
replace-annot-l-pre L' C' S' ⟹
replace-annot-wl-pre L C S›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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-$\mathcal{L}_{in}$' S} →$_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching'-nobin S* ∧
literals-are-$\mathcal{L}_{in}$' S}⟩*nres-rel*›
⟨*proof*⟩

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* = (λ*i 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-$\mathcal{L}_{in}$-literals-are-$\mathcal{L}_{in}$-trail-init:
‹*literals-are-in-$\mathcal{L}_{in}$-trail (all-init-atms-st x2) (get-trail-wl x2)*›
(**is** ‹*?trail*›)
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*⟩

definition *collect-valid-indices-wl* :: ‹*'v twl-st-wl* ⇒ *nat list nres*› **where**
‹*collect-valid-indices-wl S* = *SPEC* (λ*N*. *True*)›

definition *mark-to-delete-clauses-wl-inv*
:: ‹*'v twl-st-wl* ⇒ *nat list* ⇒ *nat* × *'v twl-st-wl*× *nat list* ⇒ *bool*›
where
‹*mark-to-delete-clauses-wl-inv* = (λ*S xs0* (*i*, *T*, *xs*).
∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧
mark-to-delete-clauses-l-inv S' xs0 (*i*, *T'*, *xs*) ∧
correct-watching' S∧ *literals-are-*$\mathcal{L}_{in}$*' S* ∧ *literals-are-*$\mathcal{L}_{in}$*' T*)›

definition *mark-to-delete-clauses-wl-pre* :: ‹*'v twl-st-wl* ⇒ *bool*›
where
‹*mark-to-delete-clauses-wl-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *mark-to-delete-clauses-l-pre T* ∧ *literals-are-*$\mathcal{L}_{in}$*' S*)›

definition *mark-garbage-wl*:: ‹*nat* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl*› **where**
‹*mark-garbage-wl* = (λ*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*))›

definition *mark-to-delete-clauses-wl* :: ‹*'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*mark-to-delete-clauses-wl* = (λ*S*. *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*(λ*b*. *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*⟩

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*›
⟨*proof*⟩

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-*$\mathcal{L}_{in}$' *S*} →$_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching'* *S* ∧ *literals-are-*$\mathcal{L}_{in}$' *S*}⟩*nres-rel*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

lemma *mark-to-delete-clauses-l-pre-blits-in-*$\mathcal{L}_{in}$*′*:
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*›
shows ‹*literals-are-*$\mathcal{L}_{in}$*′ S*›
⟨*proof*⟩

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*)
∈ {(*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*⟩

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'*) ∈ *state-wl-l None* ∧ *no-lost-clause-in-WL S* ∧ *literals-are-*$\mathcal{L}_{in}'$ *S*} →$_f$ ⟨{(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching'* *S* ∧ *literals-are-*$\mathcal{L}_{in}'$ *S* ∧
get-unkept-learned-clss-wl S = {#} ∧
get-subsumed-learned-clauses-wl S = {#} ∧
get-learned-clauses0-wl S = {#}}⟩*nres-rel*›
⟨*proof*⟩

definition *mark-to-delete-clauses-GC-wl-inv*
:: ‹'*v twl-st-wl* ⇒ *nat list* ⇒ *nat* × '*v twl-st-wl*× *nat list* ⇒ *bool*›
where
‹*mark-to-delete-clauses-GC-wl-inv* = (λ*S xs0* (*i*, *T*, *xs*).
∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧

mark-to-delete-clauses-l-inv S' xs0 (i, T', xs) ∧
no-lost-clause-in-WL S ∧ literals-are-$\mathcal{L}_{in}$' S ∧ literals-are-$\mathcal{L}_{in}$' T)›

lemma *mark-to-delete-clauses-GC-wl-inv-alt-def*:
‹mark-to-delete-clauses-GC-wl-inv = (λS xs0 (i, T, xs).
∃ S' T'. (S, S') ∈ state-wl-l None ∧ (T, T') ∈ state-wl-l None ∧
mark-to-delete-clauses-l-inv S' xs0 (i, T', xs) ∧
set (get-all-mark-of-propagated (get-trail-wl T)) ⊆ set (get-all-mark-of-propagated (get-trail-wl S))
∪ {0} ∧
no-lost-clause-in-WL S ∧ literals-are-$\mathcal{L}_{in}$' S ∧ literals-are-$\mathcal{L}_{in}$' T)›
⟨proof⟩

definition *mark-to-delete-clauses-GC-wl-pre :: ‹'v twl-st-wl ⇒ bool›*
where
‹mark-to-delete-clauses-GC-wl-pre S ⟷
(∃ T. (S, T) ∈ state-wl-l None ∧ mark-to-delete-clauses-l-GC-pre T ∧
literals-are-$\mathcal{L}_{in}$' S)›

lemma *mark-to-delete-clause-GC-wl-pre-alt-def*:
‹mark-to-delete-clauses-GC-wl-pre S ⟷
(∃ T. (S, T) ∈ state-wl-l None ∧ mark-to-delete-clauses-l-GC-pre T ∧
literals-are-$\mathcal{L}_{in}$' S ∧ set (get-all-mark-of-propagated (get-trail-wl S)) ⊆ {0})›
⟨proof⟩

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 = (λS. 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(λb. 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⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

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›
⟨proof⟩

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⟩

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⟩

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 = (λN 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* *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* *A*)) *WS*) ∧
(∀ *L* ∈ *set W*. *fst L* ∉# *dom-m N*) ∧
0 ∉# *dom-m N′*)›
⟨*proof*⟩

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* = (λ*N 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* ⟹

$xa \in$ *clauses-pointed-to*
(*Neg* ‘ *set-mset* (*remove1-mset A* $\mathcal{A}$) $\cup$ *Pos* ‘ *set-mset* (*remove1-mset A* $\mathcal{A}$))
(*W*(*Neg A* := [], *Pos A* := []))›
⟨*proof*⟩

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* $\mathcal{A}$ + *poss* $\mathcal{A}$)) *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* $\mathcal{A}$) + *poss* (*remove1-mset A* $\mathcal{A}$))) *WS'* ∧
0 ∉# *dom-m N'*
)›
⟨*proof*⟩

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* $\mathcal{A}$ *N0* = (λ($\mathcal{B}$, (*N*, *N'*, *WS*)). $\mathcal{B}$ ⊆# $\mathcal{A}$ ∧
(∀ *A* ∈ *set-mset* $\mathcal{A}$ − *set-mset* $\mathcal{B}$. (*WS* (*Pos A*) = []) ∧ *WS* (*Neg A*) = []) ∧
0 ∉# *dom-m N'* ∧
(∃ *m*. *GC-remap*** (*N0*, (λ-. *None*), *fmempty*) (*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* (*Neg* ‘ *set-mset* $\mathcal{B}$ ∪ *Pos* ‘ *set-mset* $\mathcal{B}$) *WS*))›

definition *cdcl-GC-clauses-prog-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*cdcl-GC-clauses-prog-wl* = (λ(*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*). *do* {
ASSERT(*cdcl-GC-clauses-pre-wl* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*));
$\mathcal{A}$ ← *SPEC*(λ$\mathcal{A}$. *set-mset* $\mathcal{A}$ = *set-mset* (*all-init-atms-st* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*)));
(-, (*N*, *N'*, *WS*)) ← $WHILE_T$*cdcl-GC-clauses-prog-wl-inv* $\mathcal{A}$ N_0
(λ($\mathcal{B}$, -). $\mathcal{B}$ ≠ {#})
(λ($\mathcal{B}$, (*N*, *N'*, *WS*)). *do* {
ASSERT($\mathcal{B}$ ≠ {#});
A ← *SPEC* (λ*A*. *A* ∈# $\mathcal{B}$);
(*N*, *N'*, *WS*) ← *cdcl-GC-clauses-prog-single-wl N WS A N'*;
RETURN (*remove1-mset A* $\mathcal{B}$, (*N*, *N'*, *WS*))
})
($\mathcal{A}$, (N_0, *fmempty*, *WS*));
RETURN (*M*, *N'*, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*)
})›

lemma *no-lost-clause-in-WL-no-lost-clause-in-WL0D*:
‹*no-lost-clause-in-WL S* ⟹ *no-lost-clause-in-WL0 S*›
⟨*proof*⟩

lemma *no-lost-clause-in-WL-alt-def*:
‹*no-lost-clause-in-WL0* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*) ⟷
set-mset (*dom-m* N_0) ⊆ *clauses-pointed-to*
(*Neg* ‘ *set-mset* (*all-init-atms* N_0 (*NE*+*NEk*+*NS*+*N0*)) ∪ *Pos* ‘ *set-mset* (*all-init-atms* N_0 (*NE*+*NEk*+*NS*+*N0*)))
WS›
⟨*proof*⟩

lemma *cdcl-GC-clauses-prog-wl*:
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*)›
shows
‹*cdcl-GC-clauses-prog-wl* (*M*, N_0, *D*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, *N0*, *U0*, *Q*, *WS*) ≤
(*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*⟩

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*⟩

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 ($C \in\#$ *dom-m* $N_0 \wedge$ *count-decided* $M = 0 \wedge D =$ *None* $\wedge$ *no-dup* $M \wedge C \neq 0$);
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* ($N_0 \propto C$);
let irr = *irred* N_0 C;
(unc, b, N) $\leftarrow$ *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* $\Rightarrow$ *'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*)
($\lambda i\ S.$ *if* $i \in\#$ *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) $\subseteq$ *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* $(A \cup B)\ W =$ *clauses-pointed-to* $A\ W \cup$ *clauses-pointed-to* $B\ W$›
⟨*proof*⟩

lemma *clauses-pointed-to-mono*: ‹$A \subseteq B \Longrightarrow$ *clauses-pointed-to* $A\ W \subseteq$ *clauses-pointed-to* $B\ W$›
⟨*proof*⟩

lemma *simplify-clause-with-unit-st-wl-simplify-clause-with-unit-st*:
assumes *ST*: ‹$(S,\ T) \in$ *state-wl-l None*› **and** ‹$(i,j) \in$ *nat-rel*› **and**
point: ‹*no-lost-clause-in-WL* S›
shows
‹*simplify-clause-with-unit-st-wl* i $S \leq \Downarrow \{(S',T).\ (S',T) \in$ *state-wl-l None* $\wedge$
no-lost-clause-in-WL $S' \wedge$
get-watched-wl $S' =$ *get-watched-wl* $S\}$
(*simplify-clause-with-unit-st* j T)›
⟨*proof*⟩

lemma [*twl-st-wl*, *simp*]:
assumes ‹$(\sigma,\ \sigma') \in$ *state-wl-l None*›
shows
all-learned-lits-of-l-all-learned-lits-of-wl:
‹*all-learned-lits-of-l* $\sigma' =$ *all-learned-lits-of-wl* σ› **and**
all-init-lits-of-l-all-init-lits-of-wl:
‹*all-init-lits-of-l* $\sigma' =$ *all-init-lits-of-wl* σ›
⟨*proof*⟩

lemma *simplify-clauses-with-unit-st-wl-simplify-clause-with-unit-st*:
assumes *ST*: ‹$(S,\ T) \in$ *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* $S \leq \Downarrow \{(S',T).\ (S',T) \in$ *state-wl-l None* $\wedge$
no-lost-clause-in-WL $S' \wedge$
literals-are-$\mathcal{L}_{in}$*'* $S' \wedge$ *get-watched-wl* $S' =$ *get-watched-wl* $S\}$
(*simplify-clauses-with-unit-st* T)›

⟨*proof*⟩

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-$\mathcal{L}_{in}$*′ S′* ∧ *get-watched-wl S′* = *get-watched-wl S*}
(*simplify-clauses-with-unit-st T*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*#} =

$\{\#C \in\# A.\ P\ C \wedge C \neq C'\#\}\rangle$
⟨*proof*⟩

lemma *filter-image-mset-swap*: ‹*distinct-mset A* ⟹ *distinct-mset B* ⟹
$\{\#i \in\# A.\ i \in\# B \wedge P\ i\#\} = \{\#i \in\# B.\ i \in\# A \wedge P\ i\#\}\rangle$
⟨*proof*⟩

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*) ›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*)⟩
⟨*proof*⟩

lemma *correct-watching′-leaking-bin-fmdropI-red*:
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*, *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*) ⟹ *correct-watching′-leaking-bin*
(*x1l*, *x1m*, *None*, *x1o*, *x1p*, *x1q*, *add-mset* {#*K′*#} *x1r*, *x1s*, *x1t*, *x1u*, *x1v*,
add-mset (− *K′*) *x1w*, *x2w*)⟩
⟨*proof*⟩

lemma *correct-watching′-leaking-bin-add-unitI*:
assumes ⟨*K′* ∈# *mset* (*x1m* ∝ *C′*)⟩ ⟨*C′* ∈# *dom-m x1m*⟩ ⟨*irred x1m C′*⟩
shows ⟨*correct-watching′-leaking-bin*

(x1l, x1m, None, x1o, x1p, x1q, x1r, x1s, x1t, x1u, x1v, x1w, x2w) ⟹
correct-watching′-leaking-bin
(x1l, x1m, None, x1o, x1p, add-mset {#K′#} x1q, x1r, x1s, x1t, x1u, x1v,
add-mset (− K′) x1w, x2w)›
⟨proof⟩

lemma *binary-clause-subres-wl-binary-clause-subres*:
assumes ‹(C,C′)∈nat-rel› ‹(K,K′)∈Id› ‹(L,L′)∈Id› ‹(S,S′)∈ state-wl-l None›
‹correct-watching′-leaking-bin S›
shows ‹binary-clause-subres-wl C K L S ≤
⇓{(U, V). (U,V)∈ state-wl-l None ∧ correct-watching′-leaking-bin U ∧ get-watched-wl U = get-watched-wl S} (binary-clause-subres C′ K′ L′ S′)›
⟨proof⟩

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›
⟨proof⟩

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) ← $WHILE_T^{\text{deduplicate-binary-clauses-inv-wl S L}}$ (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*⟩

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*⟩

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*⟩

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*)›
⟨*proof*⟩

For binary clauses, we can prove a stronger version of $[\![(?S,\ ?T) \in \textit{state-wl-l None};\ (?i,\ ?j) \in \textit{nat-rel};\ \textit{no-lost-clause-in-WL}\ ?S]\!] \Longrightarrow \textit{simplify-clause-with-unit-st-wl}\ ?i\ ?S \leq \Downarrow \{(S',\ T).\ (S',\ T) \in \textit{state-wl-l None} \wedge \textit{no-lost-clause-in-WL}\ S' \wedge \textit{get-watched-wl}\ S' = \textit{get-watched-wl}\ ?S\}\ (\textit{simplify-clause-with-unit-st}\ ?j\ ?T)$, 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*: ‹$(S,\ T) \in$ *state-wl-l None*› **and** *ij*: ‹$(i,j) \in$ *nat-rel*› **and**
point: ‹*correct-watching'-leaking-bin S*› **and**
‹$i \in\#$ *dom-m* (*get-clauses-wl S*)› **and**
Si: ‹*length* (*get-clauses-wl S* $\propto$ *i*) = 2›
shows
‹*simplify-clause-with-unit-st-wl i S* $\leq \Downarrow$ {(*S'*,*T*). (*S'*,*T*) $\in$ *state-wl-l None* $\wedge$
correct-watching'-leaking-bin S' $\wedge$
get-watched-wl S' = *get-watched-wl S*}
(*simplify-clause-with-unit-st j T*)›
⟨*proof*⟩

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*:
$\bigwedge x\ x'.\ [\![\ (x,x') \in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows *WHILEIT I b f x* $\leq \Downarrow$ {(*a*,*b*). (*a*,*b*) $\in$ *R* $\wedge$ *I a* $\wedge$ *I' b*} (*WHILEIT I' b' f' x'*)
⟨*proof*⟩

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* $\leq$
$\Downarrow$ {(*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*⟩

definition *mark-duplicated-binary-clauses-as-garbage-pre-wl* :: ‹*'v twl-st-wl* $\Rightarrow$ *bool*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-pre-wl* = ($\lambda S.$ ($\exists\ T.$ (*S*,*T*) $\in$ *state-wl-l None* $\wedge$
correct-watching'-leaking-bin S $\wedge$ *literals-are-*$\mathcal{L}_{in}$*' S*))›

definition *mark-duplicated-binary-clauses-as-garbage-wl-inv* :: ‹*'v multiset* $\Rightarrow$ *'v twl-st-wl* $\Rightarrow$ *'v twl-st-wl* $\times$ *'v multiset* $\Rightarrow$ *bool*› **where**
‹*mark-duplicated-binary-clauses-as-garbage-wl-inv* = ($\lambda xs\ S\ (U,\ ys)$.
$\exists\ S'\ U'.\ (U,\ U') \in$ *state-wl-l None* $\wedge$ $(S,\ S') \in$ *state-wl-l None* $\wedge$
mark-duplicated-binary-clauses-as-garbage-inv xs S' (*U'*, *ys*))›

definition *mark-duplicated-binary-clauses-as-garbage-wl* :: ‹*-* $\Rightarrow$ *'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 $\leftarrow$ *SPEC* (λ*Ls*:: *'v multiset*. $\forall$ *L*$\in\#$*Ls*. *L* $\in\#$ *atm-of* '$\#$ *all-init-lits-of-wl S*);
(*S*, -) $\leftarrow$ *WHILE*$_T$*mark-duplicated-binary-clauses-as-garbage-wl-inv Ls S*(λ(*S*, *Ls*). *Ls* $\neq \{\#\} \wedge$ *get-conflict-wl S* = *None*)
(λ(*S*, *Ls*). *do* {
L $\leftarrow$ *SPEC* (λ*L*. *L* $\in\#$ *Ls*);
ASSERT (*L* $\in\#$ *atm-of* '$\#$ *all-init-lits-of-wl S*);
skip $\leftarrow$ *RES* (*UNIV* :: *bool set*);

if skip then RETURN $(S,$ *remove1-mset L Ls*$)$
else do $\{$
$S \leftarrow$ *deduplicate-binary-clauses-wl* $(Pos\ L)\ S$;
$S \leftarrow$ *deduplicate-binary-clauses-wl* $(Neg\ L)\ S$;
RETURN $(S,$ *remove1-mset L Ls*$)$
$\}$
$\})$
$(S,\ Ls)$;
RETURN S
$\}\rangle$

lemma *mark-duplicated-binary-clauses-as-garbage-wl*:
assumes $\langle(S,\ S') \in \{(S,\ T).\ (S,\ T) \in$ *state-wl-l None* $\wedge$ *correct-watching'-leaking-bin S* $\wedge$ *literals-are-*$\mathcal{L}_{in}'$ *S*$\}\rangle$
shows $\langle$*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')$\rangle$
⟨*proof*⟩

Large clauses

definition *subsume-clauses-match-pre* :: $\langle$-$\rangle$ **where**
$\langle$*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* $(mset\ (N \propto C))\rangle$

definition *subsume-clauses-match* :: $\langle$*nat* $\Rightarrow$ *nat* $\Rightarrow$ $(nat,$ *'v literal list* $\times$ *bool) fmap* $\Rightarrow$ *'v subsumption nres*$\rangle$ **where**
$\langle$*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 $L \in$ *set* $(N \propto C')$
then RETURN $(i{+}1,\ st)$
else if $-L \in$ *set* $(N \propto C')$
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
$\}\rangle$

lemma *subset-remove1-mset-notin*:
$\langle b \notin\#\ A \Longrightarrow A \subseteq\#$ *remove1-mset b B* $\longleftrightarrow A \subseteq\# B\rangle$
⟨*proof*⟩

lemma *subsume-clauses-match*:
assumes $\langle$*subsume-clauses-match-pre C C' N*$\rangle$
shows $\langle$*subsume-clauses-match C C' N* $\leq \Downarrow$ *Id* $($*SPEC*$($*try-to-subsume C' C N*$))\rangle$
⟨*proof*⟩

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›
⟨*proof*⟩

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#})›
⟨*proof*⟩

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′#}›
⟨*proof*⟩

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*⟩

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)›
⟨*proof*⟩

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′)›
⟨*proof*⟩

definition *forward-subsumption-one-wl-pre* :: ‹*nat ⇒ nat multiset ⇒ 'v twl-st-wl ⇒ bool*› **where**
‹*forward-subsumption-one-wl-pre = (λC 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 = (λS 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 = (λxs. 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 = (λC 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 (λC'. 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 S_0;
ASSERT (literals-are-$\mathcal{L}_{in}$' S ∧ set-mset (all-init-lits-of-wl S) = set-mset (all-init-lits-of-wl S_0));
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-$\mathcal{L}_{in}$' 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⟩

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))›
⟨proof⟩

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))›
⟨proof⟩

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⟩

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⟩

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 = (λS 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-$\mathcal{L}_{in}$'D: ‹literals-are-$\mathcal{L}_{in}$' T›
⟨proof⟩

lemma *forward-subsumption-all-wl-inv-alt-def*:
‹forward-subsumption-all-wl-inv = (λS 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))›
⟨proof⟩

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*⟩

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*$_T^{\textit{pure-literal-deletion-wl-inv S xs}}$ (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*⟩

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*⟩

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*⟩

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
‹$(S, S') \in$ *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* ∧ $(S,T)\in$ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}'$ *S*} (*pure-literal-elimination-round S'*)›
⟨*proof*⟩

definition *pure-literal-elimination-wl-pre* **where**
‹*pure-literal-elimination-wl-pre S* ⟷
($\exists$ *T*. $(S, T) \in$ *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*). $\exists$ *T U'*. $(S, T) \in$ *state-wl-l None* ∧ $(U, U') \in$ *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*$_T$*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
‹$(S, S') \in$ *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* ∧ $(S,T)\in$ *state-wl-l None* ∧ *literals-are-*$\mathcal{L}_{in}'$ *S*} (*pure-literal-elimination-l S'*)›
⟨*proof*⟩

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'*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩
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*)›

⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *twl-struct-invs-init-twl-struct-invs*:
‹*other-clauses-init W* = {#} ⟹ *twl-struct-invs-init W* ⟹ *twl-struct-invs* (*fst W*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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 +
$get\text{-}unit\text{-}clauses\text{-}l\text{-}init\ SOC$ + $get\text{-}subsumed\text{-}init\text{-}clauses\text{-}l\text{-}init\ SOC$+ $get\text{-}init\text{-}clauses0\text{-}l\text{-}init$
SOC =
$mset\ `\#\ ran\text{-}mf\ (get\text{-}clauses\text{-}l\text{-}init\ SOC')$ + $other\text{-}clauses\text{-}l\text{-}init\ SOC'$ +
$get\text{-}unit\text{-}clauses\text{-}l\text{-}init\ SOC'$ + $get\text{-}subsumed\text{-}init\text{-}clauses\text{-}l\text{-}init\ SOC'$ +
$get\text{-}init\text{-}clauses0\text{-}l\text{-}init\ SOC') \wedge$
$learned\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}l\text{-}init\ SOC) = learned\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}l\text{-}init\ SOC') \wedge$
$get\text{-}learned\text{-}unit\text{-}clauses\text{-}l\text{-}init\ SOC' = get\text{-}learned\text{-}unit\text{-}clauses\text{-}l\text{-}init\ SOC \wedge$
$get\text{-}subsumed\text{-}learned\text{-}clauses\text{-}l\text{-}init\ SOC' = get\text{-}subsumed\text{-}learned\text{-}clauses\text{-}l\text{-}init\ SOC \wedge$
$get\text{-}learned\text{-}clauses0\text{-}l\text{-}init\ SOC' = get\text{-}learned\text{-}clauses0\text{-}l\text{-}init\ SOC \wedge$
$twl\text{-}list\text{-}invs\ (fst\ SOC') \wedge$
$twl\text{-}stgy\text{-}invs\ (fst\ T') \wedge$
$(other\text{-}clauses\text{-}l\text{-}init\ SOC' \neq \{\#\} \longrightarrow get\text{-}conflict\text{-}l\text{-}init\ SOC' \neq None) \wedge$
$(\{\#\} \in\#\ mset\ `\#\ mset\ CS \longrightarrow get\text{-}conflict\text{-}l\text{-}init\ SOC' \neq None) \wedge$
$(get\text{-}conflict\text{-}l\text{-}init\ SOC \neq None \longrightarrow get\text{-}conflict\text{-}l\text{-}init\ SOC = get\text{-}conflict\text{-}l\text{-}init\ SOC'))\rangle$

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*⟩

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*)›
⟨*proof*⟩

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*) ›
⟨*proof*⟩

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*) ›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

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›
⟨proof⟩

lemma [twl-st-l-init]:
‹(V, W) ∈ twl-st-l-init ⟹
count-decided (get-trail-init W) = count-decided (get-trail-l-init V)›
⟨proof⟩

lemma [twl-st-l-init]:
‹(V, W) ∈ twl-st-l-init ⟹get-subsumed-learned-clauses-init W = get-subsumed-learned-clauses-l-init V›
⟨proof⟩

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›
⟨proof⟩

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))›
⟨proof⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

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

$= \{\#\}$
›
⟨*proof*⟩

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*⟩

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*⟩

lemma (**in** −) *twl-stgy-invs-backtrack-lvl-0*:
‹*count-decided* (*get-trail T*) = *0* ⟹ *twl-stgy-invs T*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*⟩

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*) = {#}›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

lemma [*twl-st-l-init*]:
‹*get-trail* (*fst* (*add-to-clauses-init a T*)) = *get-trail-init T*›
⟨*proof*⟩

lemma [*twl-st-l-init*]:
‹*other-clauses-l-init* (*T*, *OC*) = *OC*›
‹*clauses-to-update-l-init* (*T*, *OC*) = *clauses-to-update-l T*›
⟨*proof*⟩

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*⟩

lemma *get-trail-init-add-to-clauses-init*[*simp*]:
‹*get-trail-init* (*add-to-clauses-init a T*) = *get-trail-init T*›
⟨*proof*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

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*⟩

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*)›
  ⟨*proof*⟩

lemma *init-dt-pre-empty-state*:
  ‹*init-dt-pre* [] (([], *fmempty*, *None*, {#}, {#}, {#}, {#}, {#}, {#}, {#},{#}, {#},{#}), {#})›
  ⟨*proof*⟩

lemma *twl-init-invs*:
  ‹*twl-struct-invs-init* (([], {#}, {#}, *None*, {#}, {#}, {#}, {#}, {#},{#}, {#},{#}), {#})›
  ‹*twl-list-invs* ([], *fmempty*, *None*, {#}, {#}, {#}, {#}, {#}, {#}, {#},{#}, {#},{#})›
  ‹*twl-stgy-invs* ([], {#}, {#}, *None*, {#}, {#}, {#}, {#}, {#}, {#}, {#},{#})›
  ⟨*proof*⟩
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* ∈# *dom-m N* is already entailed by *fst* ‘# *mset* (*W L*) = *clause-to-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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

lemma *image-mset-Suc*: ‹*Suc* ‘# {#*C* ∈# *M*. *P C*#} = {#*C* ∈# *Suc* ‘# *M*. *P* (*C*−*1*)#}›
⟨*proof*⟩

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*⟩

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*))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*⟩

definition *rewatch*
:: ‹'*v clauses-l* ⇒ ('*v literal* ⇒ '*v watched*) ⇒ ('*v literal* ⇒ '*v watched*) *nres*›
where
‹*rewatch N W* = *do* {
xs ← *SPEC*(λ*xs*. *set-mset* (*dom-m N*) ⊆ *set xs* ∧ *distinct xs*);
nfoldli
xs
(λ-. *True*)
(λ*i 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*(λ*W*. *correct-watching-init* (*M*, *N*, *C*, *NE*, *UE*, *NEk*, *UEk*, *NS*, *US*, N_0, U_0, *Q*, *W*))›
⟨*proof*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

lemma *out-learned-tl-None*[*simp*]:
‹*out-learned* (*tl aa*) *None ao* ⟷ *out-learned aa None ao*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *minimize-conflict-support-entailed-trail*:
assumes ‹*minimize-conflict-support M C E*› **and** ‹*M* ⊨*as CNot C*›
shows ‹*M* ⊨*as CNot E*›
⟨*proof*⟩

lemma *rtranclp-minimize-conflict-support-entailed-trail*:
assumes ‹(*minimize-conflict-support M*)** *C E*› **and** ‹*M* ⊨*as CNot C*›
shows ‹*M* ⊨*as CNot E*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *wf-minimize-conflict-support*:
shows ‹*wf* {(*C'*, *C*). *minimize-conflict-support M C C'*}›
⟨*proof*⟩
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*⟩

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* (λ*K*. *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*›
⟨*proof*⟩

lemma *filter-to-poslev-empty*[*simp*]:
‹*filter-to-poslev M L* {#} = {#}›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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 ⟨*proof*⟩
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*(λ*cach′*.
     (∀ *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*))›
  ⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *conflict-min-analysis-stack-sorted*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹
sorted (*map* (*index-in-trail M o fst*) *analyse*)›
⟨*proof*⟩
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*)›
⟨*proof*⟩

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*⟩

lemma *conflict-min-analysis-stack-neg*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹
M ⊨*as CNot* (*fst* '# *mset analyse*)›
⟨*proof*⟩

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)›
⟨proof⟩

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 =
$WHILE_T^{\text{lit-redundant-rec-loop-inv } M}$
(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* (λ*s′*. *I′ s′*)⟩ **and**
⟨⋀*s*. *I s* ⟹ *I′ s* ⟹ *b s* ⟹ *f s* ≤ *SPEC* (λ*s′*. *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*⟩

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* ∈# *D*⟩ **and**
M-D: ⟨*M* ⊨*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*⟩

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'*›
⟨*proof*⟩

lemma *true-clss-cls-add-add-mset-self*: ‹*NU* ⊨*p* *add-mset L* (*D'* + *D'*) ⟷ *NU* ⊨*p* *add-mset L D'*›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

definition *set-all-to-list* **where**
‹*set-all-to-list e ys = do* {
S ← *WHILE*$^{\lambda(i,\ xs).\ i \le length\ xs \wedge (\forall x \in set\ (take\ i\ xs).\ x = e) \wedge length\ xs = length\ ys}$
(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*))›
⟨*proof*⟩

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$_T^{lit\text{-}redundant\text{-}rec\text{-}wl\text{-}inv\ M\ NU\ D}$
(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* = []›
⟨*proof*⟩

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*⟩

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*⟩

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'''* ≡ *state*$_W$*-of S''*›
defines
‹*M* ≡ *get-trail-wl S*› **and**
M': ‹*M'* ≡ *trail S'''*› **and**
NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
NU': ‹*NU'* ≡ *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*⟩

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''' ≡ state_W-of S''⟩
  defines
    ⟨M ≡ get-trail-wl S⟩ and
    M': ⟨M' ≡ trail S'''⟩ and
    NU: ⟨NU ≡ get-clauses-wl S⟩ and
    NU': ⟨NU' ≡ mset `# ran-mf NU⟩
  assumes
    S-S': ⟨(S, S') ∈ state-wl-l None⟩ and
    S'-S'': ⟨(S', S'') ∈ twl-st-l None⟩ and
    ⟨M ≡ get-trail-wl S⟩ and
    M': ⟨M' ≡ trail S'''⟩ and
    NU: ⟨NU ≡ get-clauses-wl S⟩ and
    NU': ⟨NU' ≡ mset `# ran-mf NU⟩
  assumes
    struct-invs: ⟨twl-struct-invs S''⟩ and
    add-inv: ⟨twl-list-invs S'⟩ and
    L-D: ⟨L ∈# D⟩ and
    M-D: ⟨M ⊨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⟩

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* $\mathcal{A}$), *uncurry2 mark-failed-lits-wl*) ∈
[λ((*NU*, *analyse*), *cach*). *literals-are-in-*$\mathcal{L}_{in}$*-mm* $\mathcal{A}$ (*mset* ‘# *ran-mf NU*) ∧
mark-failed-lits-stack-inv NU analyse cach]$_f$
Id ×$_f$ *Id* ×$_f$ *Id* → ⟨*Id*⟩*nres-rel*›
⟨*proof*⟩

end