Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

July 17, 2023

Contents

1 Definition of Entailment **5**

1.1 Partial Herbrand Interpretation . . . 5

1.1.1 More Literals . . . 5

1.1.2 Clauses . . . 6

1.1.3 Partial Interpretations . . . 6

1.1.4 Subsumptions . . . 27

1.1.5 Removing Duplicates . . . 28

1.1.6 Set of all Simple Clauses . . . 28

1.1.7 Experiment: Expressing the Entailments as Locales . . . 32

1.1.8 Entailment to be extended . . . 33

1.2 Partial Annotated Herbrand Interpretation . . . 34

1.2.1 Decided Literals . . . 34

1.2.2 Backtracking . . . 40

1.2.3 Decomposition with respect to the First Decided Literals . . . 40

1.2.4 Negation of a Clause . . . 48

1.2.5 Other . . . 53

1.2.6 Extending Entailments to multisets . . . 55

1.2.7 More Lemmas . . . 56

1.2.8 Negation of annotated clauses . . . 56

1.3 Bridging of total and partial Herbrand interpretation . . . 59

2 Normalisation **63**

2.1 Logics . . . 63

2.1.1 Definition and Abstraction . . . 63

2.1.2 Properties of the Abstraction . . . 64

2.1.3 Subformulas and Properties . . . 67

2.1.4 Positions . . . 70

2.2 Semantics over the Syntax . . . 73

Chapter 1

Definition of Entailment

This chapter defines various form of entailment.

end

1.1 Partial Herbrand Interpretation

theory *Partial-Herbrand-Interpretation*
 imports
  Weidenbach-Book-Base.WB-List-More
  Ordered-Resolution-Prover.Clausal-Logic
begin

1.1.1 More Literals

The following lemma is very useful when in the goal appears an axioms like $-L = K$: this lemma allows the simplifier to rewrite L.

lemma *in-image-uminus-uminus*: $\langle a \in uminus\ `\ A \longleftrightarrow -a \in A\rangle$ **for** $a :: \langle 'v\ literal\rangle$
 using *uminus-lit-swap* **by** *auto*

lemma *uminus-lit-swap*: $-\ a = b \longleftrightarrow (a{::}'a\ literal) = -\ b$
 by *auto*

lemma *atm-of-notin-atms-of-iff*: $\langle atm\text{-}of\ L \notin atms\text{-}of\ C' \longleftrightarrow L \notin\# \ C' \wedge -L \notin\# \ C'\rangle$ **for** $L\ C'$
 by (*cases L*) (*auto simp*: *atm-iff-pos-or-neg-lit*)

lemma *atm-of-notin-atms-of-iff-Pos-Neg*:
 $\langle L \notin atms\text{-}of\ C' \longleftrightarrow Pos\ L \notin\# \ C' \wedge Neg\ L \notin\# \ C'\rangle$ **for** $L\ C'$
 by (*auto simp*: *atm-iff-pos-or-neg-lit*)

lemma *atms-of-uminus*[*simp*]: $\langle atms\text{-}of\ (uminus\ `\#\ C) = atms\text{-}of\ C\rangle$
 by (*auto simp*: *atms-of-def image-image*)

lemma *distinct-mset-atm-ofD*:
 $\langle distinct\text{-}mset\ (atm\text{-}of\ `\#\ mset\ xc) \Longrightarrow distinct\ xc\rangle$
 by (*induction xc*) *auto*

lemma *atms-of-cong-set-mset*:
 $\langle set\text{-}mset\ D = set\text{-}mset\ D' \Longrightarrow atms\text{-}of\ D = atms\text{-}of\ D'\rangle$
 by (*auto simp*: *atms-of-def*)

lemma *lit-in-set-iff-atm*:
‹*NO-MATCH* (*Pos x*) *l* ⟹ *NO-MATCH* (*Neg x*) *l* ⟹
  $l \in M \longleftrightarrow (\exists l'. (l = Pos\ l' \wedge Pos\ l' \in M) \vee (l = Neg\ l' \wedge Neg\ l' \in M))$›
by (*cases l*) *auto*

We define here entailment by a set of literals. This is an Herbrand interpretation, but not the same as used for the resolution prover. Both has different properties. One key difference is that such a set can be inconsistent (i.e. containing both L and $-\ L$).

Satisfiability is defined by the existence of a total and consistent model.

lemma *lit-eq-Neg-Pos-iff*:
‹$x \neq Neg\ (atm\text{-}of\ x) \longleftrightarrow is\text{-}pos\ x$›
‹$x \neq Pos\ (atm\text{-}of\ x) \longleftrightarrow is\text{-}neg\ x$›
‹$-x \neq Neg\ (atm\text{-}of\ x) \longleftrightarrow is\text{-}neg\ x$›
‹$-x \neq Pos\ (atm\text{-}of\ x) \longleftrightarrow is\text{-}pos\ x$›
‹$Neg\ (atm\text{-}of\ x) \neq x \longleftrightarrow is\text{-}pos\ x$›
‹$Pos\ (atm\text{-}of\ x) \neq x \longleftrightarrow is\text{-}neg\ x$›
‹$Neg\ (atm\text{-}of\ x) \neq -x \longleftrightarrow is\text{-}neg\ x$›
‹$Pos\ (atm\text{-}of\ x) \neq -x \longleftrightarrow is\text{-}pos\ x$›
by (*cases x*; *auto*; *fail*)+

1.1.2 Clauses

Clauses are set of literals or (finite) multisets of literals.

type-synonym *'v clause-set* = *'v clause set*
type-synonym *'v clauses* = *'v clause multiset*

lemma *is-neg-neg-not-is-neg*: $is\text{-}neg\ (-\ L) \longleftrightarrow \neg\ is\text{-}neg\ L$
by (*cases L*) *auto*

1.1.3 Partial Interpretations

type-synonym *'a partial-interp* = *'a literal set*

definition *true-lit* :: *'a partial-interp* ⇒ *'a literal* ⇒ *bool* (**infix** ⊨*l 50*) **where**
$I \models l\ L \longleftrightarrow L \in I$

declare *true-lit-def*[*simp*]

Consistency

definition *consistent-interp* :: *'a literal set* ⇒ *bool* **where**
$consistent\text{-}interp\ I \longleftrightarrow (\forall L. \neg(L \in I \wedge -\ L \in I))$

lemma *consistent-interp-empty*[*simp*]:
consistent-interp {} **unfolding** *consistent-interp-def* **by** *auto*

lemma *consistent-interp-single*[*simp*]:
consistent-interp {*L*} **unfolding** *consistent-interp-def* **by** *auto*

lemma *Ex-consistent-interp*: ‹*Ex consistent-interp*›
by (*auto simp*: *consistent-interp-def*)

lemma *consistent-interp-subset*:
assumes
$A \subseteq B$ **and**

consistent-interp B
shows *consistent-interp A*
using *assms* **unfolding** *consistent-interp-def* **by** *auto*

lemma *consistent-interp-change-insert*:
$a \notin A \Longrightarrow -a \notin A \Longrightarrow$ *consistent-interp* (*insert* $(-a)$ *A*) $\longleftrightarrow$ *consistent-interp* (*insert a A*)
unfolding *consistent-interp-def* **by** *fastforce*

lemma *consistent-interp-insert-pos*[*simp*]:
$a \notin A \Longrightarrow$ *consistent-interp* (*insert a A*) $\longleftrightarrow$ *consistent-interp A* $\wedge -a \notin A$
unfolding *consistent-interp-def* **by** *auto*

lemma *consistent-interp-insert-not-in*:
consistent-interp A $\Longrightarrow a \notin A \Longrightarrow -a \notin A \Longrightarrow$ *consistent-interp* (*insert a A*)
unfolding *consistent-interp-def* **by** *auto*

lemma *consistent-interp-unionD*: ‹*consistent-interp* $(I \cup I')$ $\Longrightarrow$ *consistent-interp* I'›
unfolding *consistent-interp-def* **by** *auto*

lemma *consistent-interp-insert-iff*:
‹*consistent-interp* (*insert L C*) $\longleftrightarrow$ *consistent-interp C* $\wedge -L \notin C$›
by (*metis consistent-interp-def consistent-interp-insert-pos insert-absorb*)

lemma (**in** −) *distinct-consistent-distinct-atm*:
‹*distinct M* $\Longrightarrow$ *consistent-interp* (*set M*) $\Longrightarrow$ *distinct-mset* (*atm-of* ‘# *mset M*)›
by (*induction M*) (*auto simp*: *atm-of-eq-atm-of*)

Atoms

We define here various lifting of *atm-of* (applied to a single literal) to set and multisets of literals.

definition *atms-of-ms* :: *'a clause set* ⇒ *'a set* **where**
atms-of-ms $\psi s = \bigcup$(*atms-of* ‘ ψs)

lemma *atms-of-mmltiset*[*simp*]:
atms-of (*mset a*) = *atm-of* ‘ *set a*
by (*induct a*) *auto*

lemma *atms-of-ms-mset-unfold*:
atms-of-ms (*mset* ‘ *b*) = ($\bigcup x \in b$. *atm-of* ‘ *set x*)
unfolding *atms-of-ms-def* **by** *simp*

definition *atms-of-s* :: *'a literal set* ⇒ *'a set* **where**
atms-of-s C = *atm-of* ‘ *C*

lemma *atms-of-ms-emtpy-set*[*simp*]:
atms-of-ms {} = {}
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-memtpy*[*simp*]:
atms-of-ms {{#}} = {}
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-mono*:

$A \subseteq B \Longrightarrow$ *atms-of-ms* $A \subseteq$ *atms-of-ms* B
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-finite*[*simp*]:
finite $\psi s \Longrightarrow$ *finite* (*atms-of-ms* ψs)
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-union*[*simp*]:
atms-of-ms $(\psi s \cup \chi s) =$ *atms-of-ms* $\psi s \cup$ *atms-of-ms* χs
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-insert*[*simp*]:
atms-of-ms (*insert* ψs χs) $=$ *atms-of* $\psi s \cup$ *atms-of-ms* χs
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-singleton*[*simp*]: *atms-of-ms* $\{L\} =$ *atms-of* L
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-atms-of-ms-mono*[*simp*]:
$A \in \psi \Longrightarrow$ *atms-of* $A \subseteq$ *atms-of-ms* ψ
unfolding *atms-of-ms-def* **by** *fastforce*

lemma *atms-of-ms-remove-incl*:
shows *atms-of-ms* (*Set.remove* a ψ) $\subseteq$ *atms-of-ms* ψ
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-remove-subset*:
atms-of-ms $(\varphi - \psi) \subseteq$ *atms-of-ms* φ
unfolding *atms-of-ms-def* **by** *auto*

lemma *finite-atms-of-ms-remove-subset*[*simp*]:
finite (*atms-of-ms* A) $\Longrightarrow$ *finite* (*atms-of-ms* $(A - C)$)
using *atms-of-ms-remove-subset*[*of* A C] *finite-subset* **by** *blast*

lemma *atms-of-ms-empty-iff*:
atms-of-ms $A = \{\} \longleftrightarrow A = \{\{\#\}\} \vee A = \{\}$
apply (*rule iffI*)
apply (*metis* (*no-types*, *lifting*) *atms-empty-iff-empty atms-of-atms-of-ms-mono insert-absorb singleton-iff singleton-insert-inj-eq'* *subsetI subset-empty*)
apply (*auto*; *fail*)
done

lemma *in-implies-atm-of-on-atms-of-ms*:
assumes $L \in\# C$ **and** $C \in N$
shows *atm-of* $L \in$ *atms-of-ms* N
using *atms-of-atms-of-ms-mono*[*of* C N] *assms* **by** (*simp add*: *atm-of-lit-in-atms-of subset-iff*)

lemma *in-plus-implies-atm-of-on-atms-of-ms*:
assumes $C+\{\#L\#\} \in N$
shows *atm-of* $L \in$ *atms-of-ms* N
using *in-implies-atm-of-on-atms-of-ms*[*of* - C $+\{\#L\#\}$] *assms* **by** *auto*

lemma *in-m-in-literals*:
assumes *add-mset* A $D \in \psi s$
shows *atm-of* $A \in$ *atms-of-ms* ψs
using *assms* **by** (*auto dest*: *atms-of-atms-of-ms-mono*)

lemma *atms-of-s-union*[*simp*]:
atms-of-s (*Ia* ∪ *Ib*) = *atms-of-s Ia* ∪ *atms-of-s Ib*
unfolding *atms-of-s-def* **by** *auto*

lemma *atms-of-s-single*[*simp*]:
atms-of-s {*L*} = {*atm-of L*}
unfolding *atms-of-s-def* **by** *auto*

lemma *atms-of-s-insert*[*simp*]:
atms-of-s (*insert L Ib*) = {*atm-of L*} ∪ *atms-of-s Ib*
unfolding *atms-of-s-def* **by** *auto*

lemma *in-atms-of-s-decomp*[*iff*]:
P ∈ *atms-of-s I* ⟷ (*Pos P* ∈ *I* ∨ *Neg P* ∈ *I*) (**is** *?P* ⟷ *?Q*)
proof
assume *?P*
then show *?Q* **unfolding** *atms-of-s-def* **by** (*metis image-iff literal.exhaust-sel*)
next
assume *?Q*
then show *?P* **unfolding** *atms-of-s-def* **by** *force*
qed

lemma *atm-of-in-atm-of-set-in-uminus*:
atm-of L′ ∈ *atm-of* ‘ *B* ⟹ *L′* ∈ *B* ∨ − *L′* ∈ *B*
using *atms-of-s-def* **by** (*cases L′*) *fastforce+*

lemma *finite-atms-of-s*[*simp*]:
‹*finite M* ⟹ *finite* (*atms-of-s M*)›
by (*auto simp*: *atms-of-s-def*)

lemma
atms-of-s-empty [*simp*]:
‹*atms-of-s* {} = {}› **and**
atms-of-s-empty-iff[*simp*]:
‹*atms-of-s x* = {} ⟷ *x* = {}›
by (*auto simp*: *atms-of-s-def*)

Totality

definition *total-over-set* :: *′a partial-interp* ⇒ *′a set* ⇒ *bool* **where**
total-over-set I S = (∀ *l*∈*S*. *Pos l* ∈ *I* ∨ *Neg l* ∈ *I*)

definition *total-over-m* :: *′a literal set* ⇒ *′a clause set* ⇒ *bool* **where**
total-over-m I ψs = *total-over-set I* (*atms-of-ms ψs*)

lemma *total-over-set-empty*[*simp*]:
total-over-set I {}
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-m-empty*[*simp*]:
total-over-m I {}
unfolding *total-over-m-def* **by** *auto*

lemma *total-over-set-single*[*iff*]:
total-over-set I {*L*} ⟷ (*Pos L* ∈ *I* ∨ *Neg L* ∈ *I*)

unfolding *total-over-set-def* **by** *auto*

lemma *total-over-set-insert*[*iff*]:
total-over-set I (*insert L Ls*) ⟷ ((*Pos L* ∈ *I* ∨ *Neg L* ∈ *I*) ∧ *total-over-set I Ls*)
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-set-union*[*iff*]:
total-over-set I (*Ls* ∪ *Ls′*) ⟷ (*total-over-set I Ls* ∧ *total-over-set I Ls′*)
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-m-subset*:
A ⊆ *B* ⟹ *total-over-m I B* ⟹ *total-over-m I A*
using *atms-of-ms-mono*[*of A*] **unfolding** *total-over-m-def total-over-set-def* **by** *auto*

lemma *total-over-m-sum*[*iff*]:
shows *total-over-m I* {*C* + *D*} ⟷ (*total-over-m I* {*C*} ∧ *total-over-m I* {*D*})
unfolding *total-over-m-def total-over-set-def* **by** *auto*

lemma *total-over-m-union*[*iff*]:
total-over-m I (*A* ∪ *B*) ⟷ (*total-over-m I A* ∧ *total-over-m I B*)
unfolding *total-over-m-def total-over-set-def* **by** *auto*

lemma *total-over-m-insert*[*iff*]:
total-over-m I (*insert a A*) ⟷ (*total-over-set I* (*atms-of a*) ∧ *total-over-m I A*)
unfolding *total-over-m-def total-over-set-def* **by** *fastforce*

lemma *total-over-m-extension*:
fixes *I* :: *′v literal set* **and** *A* :: *′v clause-set*
assumes *total*: *total-over-m I A*
shows ∃ *I′*. *total-over-m* (*I* ∪ *I′*) (*A*∪*B*)
∧ (∀ *x*∈*I′*. *atm-of x* ∈ *atms-of-ms B* ∧ *atm-of x* ∉ *atms-of-ms A*)
proof −
let *?I′* = {*Pos v* |*v*. *v*∈ *atms-of-ms B* ∧ *v* ∉ *atms-of-ms A*}
have ∀ *x*∈*?I′*. *atm-of x* ∈ *atms-of-ms B* ∧ *atm-of x* ∉ *atms-of-ms A* **by** *auto*
moreover have *total-over-m* (*I* ∪ *?I′*) (*A*∪*B*)
using *total* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
ultimately show *?thesis* **by** *blast*
qed

lemma *total-over-m-consistent-extension*:
fixes *I* :: *′v literal set* **and** *A* :: *′v clause-set*
assumes
total: *total-over-m I A* **and**
cons: *consistent-interp I*
shows ∃ *I′*. *total-over-m* (*I* ∪ *I′*) (*A* ∪ *B*)
∧ (∀ *x*∈*I′*. *atm-of x* ∈ *atms-of-ms B* ∧ *atm-of x* ∉ *atms-of-ms A*) ∧ *consistent-interp* (*I* ∪ *I′*)
proof −
let *?I′* = {*Pos v* |*v*. *v*∈ *atms-of-ms B* ∧ *v* ∉ *atms-of-ms A* ∧ *Pos v* ∉ *I* ∧ *Neg v* ∉ *I*}
have ∀ *x*∈*?I′*. *atm-of x* ∈ *atms-of-ms B* ∧ *atm-of x* ∉ *atms-of-ms A* **by** *auto*
moreover have *total-over-m* (*I* ∪ *?I′*) (*A*∪*B*)
using *total* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
moreover have *consistent-interp* (*I* ∪ *?I′*)
using *cons* **unfolding** *consistent-interp-def* **by** (*intro allI*) (*rename-tac L*, *case-tac L*, *auto*)
ultimately show *?thesis* **by** *blast*
qed

lemma *total-over-set-atms-of-m*[*simp*]:
total-over-set Ia (*atms-of-s Ia*)
unfolding *total-over-set-def atms-of-s-def* **by** (*metis image-iff literal.exhaust-sel*)

lemma *total-over-set-literal-defined*:
assumes *add-mset A D* $\in \psi s$
and *total-over-set I* (*atms-of-ms* ψs)
shows $A \in I \lor -A \in I$
using *assms* **unfolding** *total-over-set-def* **by** (*metis* (*no-types*) *Neg-atm-of-iff in-m-in-literals literal.collapse*(*1*) *uminus-Neg uminus-Pos*)

lemma *tot-over-m-remove*:
assumes *total-over-m* $(I \cup \{L\})$ $\{\psi\}$
and *L*: $L \notin\# \psi$ $-L \notin\# \psi$
shows *total-over-m I* $\{\psi\}$
unfolding *total-over-m-def total-over-set-def*
proof
fix *l*
assume *l*: $l \in$ *atms-of-ms* $\{\psi\}$
then have *Pos* $l \in I \lor$ *Neg* $l \in I \lor l =$ *atm-of L*
using *assms* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
moreover have *atm-of L* $\notin$ *atms-of-ms* $\{\psi\}$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then have *atm-of L* $\in$ *atms-of* ψ **by** *auto*
then have *Pos* (*atm-of L*) $\in\# \psi \lor$ *Neg* (*atm-of L*) $\in\# \psi$
using *atm-imp-pos-or-neg-lit* **by** *metis*
then have $L \in\# \psi \lor - L \in\# \psi$ **by** (*cases L*) *auto*
then show *False* **using** *L* **by** *auto*
qed
ultimately show *Pos* $l \in I \lor$ *Neg* $l \in I$ **using** *l* **by** *metis*
qed

lemma *total-union*:
assumes *total-over-m I* ψ
shows *total-over-m* $(I \cup I')$ ψ
using *assms* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*

lemma *total-union-2*:
assumes *total-over-m I* ψ
and *total-over-m* I' ψ'
shows *total-over-m* $(I \cup I')$ $(\psi \cup \psi')$
using *assms* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*

lemma *total-over-m-alt-def*: ‹*total-over-m I S* $\longleftrightarrow$ *atms-of-ms S* $\subseteq$ *atms-of-s I*›
by (*auto simp*: *total-over-m-def total-over-set-def*)

lemma *total-over-set-alt-def*: ‹*total-over-set M A* $\longleftrightarrow$ $A \subseteq$ *atms-of-s M*›
by (*auto simp*: *total-over-set-def*)

Interpretations

definition *true-cls* :: *'a partial-interp* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool* (**infix** $\models$ *50*) **where**
$I \models C \longleftrightarrow (\exists L \in\# C.\ I \models l\ L)$

lemma *true-cls-empty*[*iff*]: $\neg\ I \models \{\#\}$

unfolding *true-cls-def* **by** *auto*

lemma *true-cls-singleton*[*iff*]: $I \models \{\#L\#\} \longleftrightarrow I \models l\ L$
unfolding *true-cls-def* **by** (*auto split*:*if-split-asm*)

lemma *true-cls-add-mset*[*iff*]: $I \models$ *add-mset a D* $\longleftrightarrow a \in I \vee I \models D$
unfolding *true-cls-def* **by** *auto*

lemma *true-cls-union*[*iff*]: $I \models C + D \longleftrightarrow I \models C \vee I \models D$
unfolding *true-cls-def* **by** *auto*

lemma *true-cls-mono-set-mset*: *set-mset C* $\subseteq$ *set-mset D* $\Longrightarrow I \models C \Longrightarrow I \models D$
unfolding *true-cls-def subset-eq Bex-def* **by** *metis*

lemma *true-cls-mono-leD*[*dest*]: $A \subseteq\# B \Longrightarrow I \models A \Longrightarrow I \models B$
unfolding *true-cls-def* **by** *auto*

lemma
assumes $I \models \psi$
shows
true-cls-union-increase[*simp*]: $I \cup I' \models \psi$ **and**
true-cls-union-increase'[*simp*]: $I' \cup I \models \psi$
using *assms* **unfolding** *true-cls-def* **by** *auto*

lemma *true-cls-mono-set-mset-l*:
assumes $A \models \psi$
and $A \subseteq B$
shows $B \models \psi$
using *assms* **unfolding** *true-cls-def* **by** *auto*

lemma *true-cls-replicate-mset*[*iff*]: $I \models$ *replicate-mset n L* $\longleftrightarrow n \neq 0 \wedge I \models l\ L$
by (*induct n*) *auto*

lemma *true-cls-empty-entails*[*iff*]: $\neg \{\} \models N$
by (*auto simp add*: *true-cls-def*)

lemma *true-cls-not-in-remove*:
assumes $L \notin\# \chi$ **and** $I \cup \{L\} \models \chi$
shows $I \models \chi$
using *assms* **unfolding** *true-cls-def* **by** *auto*

definition *true-clss* :: *'a partial-interp* $\Rightarrow$ *'a clause-set* $\Rightarrow$ *bool* (**infix** $\models s$ *50*) **where**
$I \models s\ CC \longleftrightarrow (\forall C \in CC.\ I \models C)$

lemma *true-clss-empty*[*simp*]: $I \models s \{\}$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-singleton*[*iff*]: $I \models s \{C\} \longleftrightarrow I \models C$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-empty-entails-empty*[*iff*]: $\{\} \models s\ N \longleftrightarrow N = \{\}$
unfolding *true-clss-def* **by** (*auto simp add*: *true-cls-def*)

lemma *true-cls-insert-l* [*simp*]:
$M \models A \Longrightarrow$ *insert L M* $\models A$
unfolding *true-cls-def* **by** *auto*

lemma *true-clss-union*[*iff*]: $I \models s\ CC \cup DD \longleftrightarrow I \models s\ CC \wedge I \models s\ DD$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-insert*[*iff*]: $I \models s$ *insert* $C\ DD \longleftrightarrow I \models C \wedge I \models s\ DD$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-mono*: $DD \subseteq CC \Longrightarrow I \models s\ CC \Longrightarrow I \models s\ DD$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-union-increase*[*simp*]:
assumes $I \models s\ \psi$
shows $I \cup I' \models s\ \psi$
using *assms* **unfolding** *true-clss-def* **by** *auto*

lemma *true-clss-union-increase'*[*simp*]:
assumes $I' \models s\ \psi$
shows $I \cup I' \models s\ \psi$
using *assms* **by** (*auto simp add*: *true-clss-def*)

lemma *true-clss-commute-l*:
$(I \cup I' \models s\ \psi) \longleftrightarrow (I' \cup I \models s\ \psi)$
by (*simp add*: *Un-commute*)

lemma *model-remove*[*simp*]: $I \models s\ N \Longrightarrow I \models s$ *Set.remove a N*
by (*simp add*: *true-clss-def*)

lemma *model-remove-minus*[*simp*]: $I \models s\ N \Longrightarrow I \models s\ N - A$
by (*simp add*: *true-clss-def*)

lemma *notin-vars-union-true-cls-true-cls*:
assumes $\forall x \in I'$. *atm-of* $x \notin$ *atms-of-ms A*
and *atms-of L* $\subseteq$ *atms-of-ms A*
and $I \cup I' \models L$
shows $I \models L$
using *assms* **unfolding** *true-cls-def true-lit-def Bex-def*
by (*metis Un-iff atm-of-lit-in-atms-of contra-subsetD*)

lemma *notin-vars-union-true-clss-true-clss*:
assumes $\forall x \in I'$. *atm-of* $x \notin$ *atms-of-ms A*
and *atms-of-ms L* $\subseteq$ *atms-of-ms A*
and $I \cup I' \models s\ L$
shows $I \models s\ L$
using *assms* **unfolding** *true-clss-def true-lit-def Ball-def*
by (*meson atms-of-atms-of-ms-mono notin-vars-union-true-cls-true-cls subset-trans*)

lemma *true-cls-def-set-mset-eq*:
‹*set-mset* A = *set-mset* $B \Longrightarrow I \models A \longleftrightarrow I \models B$›
by (*auto simp*: *true-cls-def*)

lemma *true-cls-add-mset-strict*: ‹$I \models$ *add-mset* $L\ C \longleftrightarrow L \in I \vee I \models$ (*removeAll-mset L C*)›
using *true-cls-mono-set-mset*[*of* ‹*removeAll-mset L C*› *C I*]
apply (*cases* ‹$L \in\#\ C$›)
apply (*auto dest*: *multi-member-split simp*: *removeAll-notin*)
apply (*metis* (*mono-tags*, *lifting*) *in-multiset-minus-notin-snd in-replicate-mset true-cls-def true-lit-def*)
done

Satisfiability

definition *satisfiable* :: *'a clause set* ⇒ *bool* **where**
satisfiable CC ⟷ (∃ *I*. (*I* ⊨*s CC* ∧ *consistent-interp I* ∧ *total-over-m I CC*))

lemma *satisfiable-single*[*simp*]:
satisfiable {{#*L*#}}
unfolding *satisfiable-def* **by** *fastforce*

lemma *satisfiable-empty*[*simp*]: ‹*satisfiable* {}›
by (*auto simp*: *satisfiable-def Ex-consistent-interp*)

abbreviation *unsatisfiable* :: *'a clause set* ⇒ *bool* **where**
unsatisfiable CC ≡ ¬ *satisfiable CC*

lemma *satisfiable-decreasing*:
assumes *satisfiable* (ψ ∪ ψ')
shows *satisfiable* ψ
using *assms total-over-m-union* **unfolding** *satisfiable-def* **by** *blast*

lemma *satisfiable-def-min*:
satisfiable CC
⟷ (∃ *I*. *I* ⊨*s CC* ∧ *consistent-interp I* ∧ *total-over-m I CC* ∧ *atm-of*'*I* = *atms-of-ms CC*)
(**is** *?sat* ⟷ *?B*)
proof
assume *?B* **then show** *?sat* **by** (*auto simp add*: *satisfiable-def*)
next
assume *?sat*
then obtain *I* **where**
I-CC: *I* ⊨*s CC* **and**
cons: *consistent-interp I* **and**
tot: *total-over-m I CC*
unfolding *satisfiable-def* **by** *auto*
let *?I* = {*P*. *P* ∈ *I* ∧ *atm-of P* ∈ *atms-of-ms CC*}

have *I-CC*: *?I* ⊨*s CC*
using *I-CC in-implies-atm-of-on-atms-of-ms* **unfolding** *true-clss-def Ball-def true-cls-def Bex-def true-lit-def*
by *blast*

moreover have *cons*: *consistent-interp ?I*
using *cons* **unfolding** *consistent-interp-def* **by** *auto*
moreover have *total-over-m ?I CC*
using *tot* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
moreover
have *atms-CC-incl*: *atms-of-ms CC* ⊆ *atm-of*'*I*
using *tot* **unfolding** *total-over-m-def total-over-set-def atms-of-ms-def*
by (*auto simp add*: *atms-of-def atms-of-s-def*[*symmetric*])
have *atm-of* ' *?I* = *atms-of-ms CC*
using *atms-CC-incl* **unfolding** *atms-of-ms-def* **by** *force*
ultimately show *?B* **by** *auto*
qed

lemma *satisfiable-carac*:
(∃ *I*. *consistent-interp I* ∧ *I* ⊨*s* φ) ⟷ *satisfiable* φ (**is** (∃ *I*. *?Q I*) ⟷ *?S*)
proof

assume *?S*
then show $\exists I.$ *?Q I* **unfolding** *satisfiable-def* **by** *auto*
next
assume $\exists I.$ *?Q I*
then obtain *I* **where** *cons*: *consistent-interp I* **and** *I*: $I \models s\ \varphi$ **by** *metis*
let $?I' = \{Pos\ v\ |v.\ v \notin$ *atms-of-s* $I \land v \in$ *atms-of-ms* $\varphi\}$
have *consistent-interp* $(I \cup ?I')$
using *cons* **unfolding** *consistent-interp-def* **by** (*intro allI*) (*rename-tac L*, *case-tac L*, *auto*)
moreover have *total-over-m* $(I \cup ?I')\ \varphi$
unfolding *total-over-m-def total-over-set-def* **by** *auto*
moreover have $I \cup ?I' \models s\ \varphi$
using *I* **unfolding** *Ball-def true-clss-def true-cls-def* **by** *auto*
ultimately show *?S* **unfolding** *satisfiable-def* **by** *blast*
qed

lemma *satisfiable-carac'*[*simp*]: *consistent-interp I* $\Longrightarrow I \models s\ \varphi \Longrightarrow$ *satisfiable* φ
using *satisfiable-carac* **by** *metis*

lemma *unsatisfiable-mono*:
‹$N \subseteq N' \Longrightarrow$ *unsatisfiable N* $\Longrightarrow$ *unsatisfiable N'*›
by (*metis* (*full-types*) *satisfiable-decreasing subset-Un-eq*)

Entailment for Multisets of Clauses

definition *true-cls-mset* :: *'a partial-interp* ⇒ *'a clause multiset* ⇒ *bool* (**infix** ⊨*m 50*) **where**
$I \models m\ CC \longleftrightarrow (\forall C \in\#\ CC.\ I \models C)$

lemma *true-cls-mset-empty*[*simp*]: $I \models m\ \{\#\}$
unfolding *true-cls-mset-def* **by** *auto*

lemma *true-cls-mset-singleton*[*iff*]: $I \models m\ \{\#C\#\} \longleftrightarrow I \models C$
unfolding *true-cls-mset-def* **by** (*auto split*: *if-split-asm*)

lemma *true-cls-mset-union*[*iff*]: $I \models m\ CC + DD \longleftrightarrow I \models m\ CC \land I \models m\ DD$
unfolding *true-cls-mset-def* **by** *fastforce*

lemma *true-cls-mset-add-mset*[*iff*]: $I \models m$ *add-mset C CC* $\longleftrightarrow I \models C \land I \models m\ CC$
unfolding *true-cls-mset-def* **by** *auto*

lemma *true-cls-mset-image-mset*[*iff*]: $I \models m$ *image-mset f A* $\longleftrightarrow (\forall x \in\#\ A.\ I \models f\ x)$
unfolding *true-cls-mset-def* **by** *fastforce*

lemma *true-cls-mset-mono*: *set-mset DD* $\subseteq$ *set-mset CC* $\Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
unfolding *true-cls-mset-def subset-iff* **by** *auto*

lemma *true-clss-set-mset*[*iff*]: $I \models s$ *set-mset CC* $\longleftrightarrow I \models m\ CC$
unfolding *true-clss-def true-cls-mset-def* **by** *auto*

lemma *true-cls-mset-increasing-r*[*simp*]:
$I \models m\ CC \Longrightarrow I \cup J \models m\ CC$
unfolding *true-cls-mset-def* **by** *auto*

theorem *true-cls-remove-unused*:
assumes $I \models \psi$
shows $\{v \in I.$ *atm-of* $v \in$ *atms-of* $\psi\} \models \psi$
using *assms* **unfolding** *true-cls-def atms-of-def* **by** *auto*

theorem *true-clss-remove-unused*:
  assumes $I \models s\ \psi$
  shows $\{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\} \models s\ \psi$
  unfolding *true-clss-def atms-of-def Ball-def*
proof (*intro allI impI*)
  fix x
  assume $x \in \psi$
  then have $I \models x$
    using *assms* **unfolding** *true-clss-def atms-of-def Ball-def* **by** *auto*

  then have $\{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\ x\} \models x$
    by (*simp only*: *true-cls-remove-unused*[*of I*])
  moreover have $\{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\ x\} \subseteq \{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\}$
    using ‹$x \in \psi$› **by** (*auto simp add*: *atms-of-ms-def*)
  ultimately show $\{v \in I.\ atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\} \models x$
    using *true-cls-mono-set-mset-l* **by** *blast*
qed

A simple application of the previous theorem:

lemma *true-clss-union-decrease*:
  assumes II': $I \cup I' \models \psi$
  and H: $\forall v \in I'.\ atm\text{-}of\ v \notin atms\text{-}of\ \psi$
  shows $I \models \psi$
proof −
  let $?I = \{v \in I \cup I'.\ atm\text{-}of\ v \in atms\text{-}of\ \psi\}$
  have $?I \models \psi$ **using** *true-cls-remove-unused* II' **by** *blast*
  moreover have $?I \subseteq I$ **using** H **by** *auto*
  ultimately show *?thesis* **using** *true-cls-mono-set-mset-l* **by** *blast*
qed

lemma *multiset-not-empty*:
  assumes $M \neq \{\#\}$
  and $x \in\# M$
  shows $\exists A.\ x = Pos\ A \lor x = Neg\ A$
  using *assms literal.exhaust-sel* **by** *blast*

lemma *atms-of-ms-empty*:
  fixes ψ :: *'v clause-set*
  assumes $atms\text{-}of\text{-}ms\ \psi = \{\}$
  shows $\psi = \{\} \lor \psi = \{\{\#\}\}$
  using *assms* **by** (*auto simp add*: *atms-of-ms-def*)

lemma *consistent-interp-disjoint*:
 assumes *consI*: *consistent-interp I*
 and *disj*: $atms\text{-}of\text{-}s\ A \cap atms\text{-}of\text{-}s\ I = \{\}$
 and *consA*: *consistent-interp A*
 shows *consistent-interp* $(A \cup I)$
proof (*rule ccontr*)
  assume ¬ *?thesis*
  moreover have $\bigwedge L.\ \neg\ (L \in A \land -L \in I)$
    using *disj* **unfolding** *atms-of-s-def* **by** (*auto simp add*: *rev-image-eqI*)
  ultimately show *False*
    using *consA consI* **unfolding** *consistent-interp-def* **by** (*metis* (*full-types*) *Un-iff*
      literal.exhaust-sel uminus-Neg uminus-Pos)
qed

```
lemma total-remove-unused:
  assumes total-over-m I ψ
  shows total-over-m {v ∈ I. atm-of v ∈ atms-of-ms ψ} ψ
  using assms unfolding total-over-m-def total-over-set-def
  by (metis (lifting) literal.sel(1,2) mem-Collect-eq)

lemma true-cls-remove-hd-if-notin-vars:
  assumes insert a M′⊨ D
  and atm-of a ∉ atms-of D
  shows M′ ⊨ D
  using assms by (auto simp add: atm-of-lit-in-atms-of true-cls-def)

lemma total-over-set-atm-of:
  fixes I :: ′v partial-interp and K :: ′v set
  shows total-over-set I K ⟷ (∀ l ∈ K. l ∈ (atm-of ‘ I))
  unfolding total-over-set-def by (metis atms-of-s-def in-atms-of-s-decomp)

lemma true-cls-mset-true-clss-iff:
  ‹finite C ⟹ I ⊨m mset-set C ⟷ I ⊨s C›
  ‹I ⊨m D ⟷ I ⊨s set-mset D›
  by (auto simp: true-clss-def true-cls-mset-def Ball-def
    dest: multi-member-split)
```

Tautologies

We define tautologies as clause entailed by every total model and show later that is equivalent to containing a literal and its negation.

```
definition tautology (ψ:: ′v clause) ≡ ∀ I. total-over-set I (atms-of ψ) ⟶ I ⊨ ψ

lemma tautology-Pos-Neg[intro]:
  assumes Pos p ∈# A and Neg p ∈# A
  shows tautology A
  using assms unfolding tautology-def total-over-set-def true-cls-def Bex-def
  by (meson atm-iff-pos-or-neg-lit true-lit-def)

lemma tautology-minus[simp]:
  assumes L ∈# A and −L ∈# A
  shows tautology A
  by (metis assms literal.exhaust tautology-Pos-Neg uminus-Neg uminus-Pos)

lemma tautology-exists-Pos-Neg:
  assumes tautology ψ
  shows ∃ p. Pos p ∈# ψ ∧ Neg p ∈# ψ
proof (rule ccontr)
  assume p: ¬ (∃ p. Pos p ∈# ψ ∧ Neg p ∈# ψ)
  let ?I = {−L | L. L ∈# ψ}
  have total-over-set ?I (atms-of ψ)
    unfolding total-over-set-def using atm-imp-pos-or-neg-lit by force
  moreover have ¬ ?I ⊨ ψ
    unfolding true-cls-def true-lit-def Bex-def apply clarify
    using p by (rename-tac x L, case-tac L) fastforce+
  ultimately show False using assms unfolding tautology-def by auto
qed
```

```
lemma tautology-decomp:
  tautology ψ ⟷ (∃p. Pos p ∈# ψ ∧ Neg p ∈# ψ)
  using tautology-exists-Pos-Neg by auto

lemma tautology-union-add-iff[simp]:
  ‹tautology (A ∪# B) ⟷ tautology (A + B)›
  by (auto simp: tautology-decomp)
lemma tautology-add-mset-union-add-iff[simp]:
  ‹tautology (add-mset L (A ∪# B)) ⟷ tautology (add-mset L (A + B))›
  by (auto simp: tautology-decomp)

lemma not-tautology-minus:
  ‹¬tautology A ⟹ ¬tautology (A − B)›
  by (auto simp: tautology-decomp dest: in-diffD)

lemma tautology-false[simp]: ¬tautology {#}
  unfolding tautology-def by auto

lemma tautology-add-mset:
  tautology (add-mset a L) ⟷ tautology L ∨ −a ∈# L
  unfolding tautology-decomp by (cases a) auto

lemma tautology-single[simp]: ‹¬tautology {#L#}›
  by (simp add: tautology-add-mset)

lemma tautology-union:
  ‹tautology (A + B) ⟷ tautology A ∨ tautology B ∨ (∃a. a ∈# A ∧ −a ∈# B)›
  by (metis tautology-decomp tautology-minus uminus-Neg uminus-Pos union-iff)

lemma
  tautology-poss[simp]: ‹¬tautology (poss A)› and
  tautology-negs[simp]: ‹¬tautology (negs A)›
  by (auto simp: tautology-decomp)

lemma tautology-uminus[simp]:
  ‹tautology (uminus ‘# w) ⟷ tautology w›
  by (auto 5 5 simp: tautology-decomp add-mset-eq-add-mset eq-commute[of ‹Pos -› ‹−-›]
     eq-commute[of ‹Neg -› ‹−-›]
    simp flip: uminus-lit-swap
    dest!: multi-member-split)

lemma minus-interp-tautology:
  assumes {−L | L. L∈# χ} ⊨ χ
  shows tautology χ
proof −
  obtain L where L ∈# χ ∧ −L ∈# χ
    using assms unfolding true-cls-def by auto
  then show ?thesis using tautology-decomp literal.exhaust uminus-Neg uminus-Pos by metis
qed

lemma remove-literal-in-model-tautology:
  assumes I ∪ {Pos P} ⊨ φ
  and I ∪ {Neg P} ⊨ φ
  shows I ⊨ φ ∨ tautology φ
  using assms unfolding true-cls-def by auto
```

lemma *tautology-imp-tautology*:
fixes χ χ' :: *'v clause*
assumes ∀ *I*. *total-over-m I* {χ} ⟶ *I* ⊨ χ ⟶ *I* ⊨ χ' **and** *tautology* χ
shows *tautology* χ' **unfolding** *tautology-def*
proof (*intro allI HOL.impI*)
fix *I* ::*'v literal set*
assume *totI*: *total-over-set I* (*atms-of* χ')
let *?I'* = {*Pos v* |*v*. *v*∈ *atms-of* χ ∧ *v* ∉ *atms-of-s I*}
have *totI'*: *total-over-m* (*I* ∪ *?I'*) {χ} **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
then have χ: *I* ∪ *?I'* ⊨ χ **using** *assms*(*2*) **unfolding** *total-over-m-def tautology-def* **by** *simp*
then have *I* ∪ (*?I'*− *I*) ⊨ χ' **using** *assms*(*1*) *totI'* **by** *auto*
moreover have ⋀*L*. *L* ∈# χ' ⟹ *L* ∉ *?I'*
using *totI* **unfolding** *total-over-set-def* **by** (*auto dest*: *pos-lit-in-atms-of*)
ultimately show *I* ⊨ χ' **unfolding** *true-cls-def* **by** *auto*
qed

lemma *not-tautology-mono*: ‹*D'* ⊆# *D* ⟹ ¬*tautology D* ⟹ ¬*tautology D'*›
by (*meson tautology-imp-tautology true-cls-add-mset true-cls-mono-leD*)

lemma *tautology-decomp'*:
‹*tautology C* ⟷ (∃ *L*. *L* ∈# *C* ∧ − *L* ∈# *C*)›
unfolding *tautology-decomp*
apply *auto*
apply (*case-tac L*)
apply *auto*
done

lemma *consistent-interp-tautology*:
‹*consistent-interp* (*set M'*) ⟷ ¬*tautology* (*mset M'*)›
by (*auto simp*: *consistent-interp-def tautology-decomp lit-in-set-iff-atm*)

lemma *consistent-interp-tuatology-mset-set*:
‹*finite x* ⟹ *consistent-interp x* ⟷ ¬*tautology* (*mset-set x*)›
using *ex-mset*[*of* ‹*mset-set x*›]
by (*auto simp*: *consistent-interp-tautology eq-commute*[*of* ‹*mset -*›] *mset-set-eq-mset-iff mset-set-set*)

lemma *tautology-distinct-atm-iff*:
‹*distinct-mset C* ⟹ *tautology C* ⟷ ¬*distinct-mset* (*atm-of* '# *C*)›
by (*induction C*)
(*auto simp*: *tautology-add-mset atm-of-eq-atm-of*
dest: *multi-member-split*)

lemma *not-tautology-minusD*:
‹*tautology* (*A* − *B*) ⟹ *tautology A*›
by (*auto simp*: *tautology-decomp dest*: *in-diffD*)

lemma *tautology-length-ge2*: ‹*tautology C* ⟹ *size C* ≥ *2*›
by (*auto simp*: *tautology-decomp add-mset-eq-add-mset dest*!: *multi-member-split*)

lemma *tautology-add-subset*: ‹*A* ⊆# *Aa* ⟹ *tautology* (*A* + *Aa*) ⟷ *tautology Aa*› **for** *A Aa*
by (*metis mset-subset-eqD subset-mset.add-diff-inverse tautology-minus tautology-union*)

Entailment for clauses and propositions

We also need entailment of clauses by other clauses.

definition *true-cls-cls* :: *'a clause* ⇒ *'a clause* ⇒ *bool* (**infix** $\models f$ *49*) **where**
$\psi \models f\ \chi \longleftrightarrow (\forall I.\ total\text{-}over\text{-}m\ I\ (\{\psi\} \cup \{\chi\}) \longrightarrow consistent\text{-}interp\ I \longrightarrow I \models \psi \longrightarrow I \models \chi)$

definition *true-cls-clss* :: *'a clause* ⇒ *'a clause-set* ⇒ *bool* (**infix** $\models fs$ *49*) **where**
$\psi \models fs\ \chi \longleftrightarrow (\forall I.\ total\text{-}over\text{-}m\ I\ (\{\psi\} \cup \chi) \longrightarrow consistent\text{-}interp\ I \longrightarrow I \models \psi \longrightarrow I \models s\ \chi)$

definition *true-clss-cls* :: *'a clause-set* ⇒ *'a clause* ⇒ *bool* (**infix** $\models p$ *49*) **where**
$N \models p\ \chi \longleftrightarrow (\forall I.\ total\text{-}over\text{-}m\ I\ (N \cup \{\chi\}) \longrightarrow consistent\text{-}interp\ I \longrightarrow I \models s\ N \longrightarrow I \models \chi)$

definition *true-clss-clss* :: *'a clause-set* ⇒ *'a clause-set* ⇒ *bool* (**infix** $\models ps$ *49*) **where**
$N \models ps\ N' \longleftrightarrow (\forall I.\ total\text{-}over\text{-}m\ I\ (N \cup N') \longrightarrow consistent\text{-}interp\ I \longrightarrow I \models s\ N \longrightarrow I \models s\ N')$

lemma *true-cls-cls-refl*[*simp*]:
  $A \models f\ A$
  unfolding *true-cls-cls-def* **by** *auto*

lemma *true-clss-cls-empty-empty*[*iff*]:
  ‹$\{\} \models p\ \{\#\} \longleftrightarrow False$›
  unfolding *true-clss-cls-def consistent-interp-def* **by** *auto*

lemma *true-cls-cls-insert-l*[*simp*]:
  $a \models f\ C \Longrightarrow insert\ a\ A \models p\ C$
  unfolding *true-cls-cls-def true-clss-cls-def true-clss-def* **by** *fastforce*

lemma *true-cls-clss-empty*[*iff*]:
  $N \models fs\ \{\}$
  unfolding *true-cls-clss-def* **by** *auto*

lemma *true-prop-true-clause*[*iff*]:
  $\{\varphi\} \models p\ \psi \longleftrightarrow \varphi \models f\ \psi$
  unfolding *true-cls-cls-def true-clss-cls-def* **by** *auto*

lemma *true-clss-clss-true-clss-cls*[*iff*]:
  $N \models ps\ \{\psi\} \longleftrightarrow N \models p\ \psi$
  unfolding *true-clss-clss-def true-clss-cls-def* **by** *auto*

lemma *true-clss-clss-true-cls-clss*[*iff*]:
  $\{\chi\} \models ps\ \psi \longleftrightarrow \chi \models fs\ \psi$
  unfolding *true-clss-clss-def true-cls-clss-def* **by** *auto*

lemma *true-clss-clss-empty*[*simp*]:
  $N \models ps\ \{\}$
  unfolding *true-clss-clss-def* **by** *auto*

lemma *true-clss-cls-subset*:
  $A \subseteq B \Longrightarrow A \models p\ CC \Longrightarrow B \models p\ CC$
  unfolding *true-clss-cls-def total-over-m-union* **by** (*simp add*: *total-over-m-subset true-clss-mono*)

This version of $[\![?A \subseteq\ ?B;\ ?A \models p\ ?CC]\!] \Longrightarrow\ ?B \models p\ ?CC$ is useful as intro rule.

lemma (**in** −)*true-clss-cls-subsetI*: ‹$I \models p\ A \Longrightarrow I \subseteq I' \Longrightarrow I' \models p\ A$›
  by (*simp add*: *true-clss-cls-subset*)

lemma *true-clss-cs-mono-l*[*simp*]:
  $A \models p\ CC \Longrightarrow A \cup B \models p\ CC$
  by (*auto intro*: *true-clss-cls-subset*)

lemma *true-clss-cs-mono-l2*[*simp*]:
$B \models p\ CC \Longrightarrow A \cup B \models p\ CC$
by (*auto intro*: *true-clss-cls-subset*)

lemma *true-clss-cls-mono-r*[*simp*]:
$A \models p\ CC \Longrightarrow A \models p\ CC + CC'$
unfolding *true-clss-cls-def total-over-m-union total-over-m-sum* **by** *blast*

lemma *true-clss-cls-mono-r'*[*simp*]:
$A \models p\ CC' \Longrightarrow A \models p\ CC + CC'$
unfolding *true-clss-cls-def total-over-m-union total-over-m-sum* **by** *blast*

lemma *true-clss-cls-mono-add-mset*[*simp*]:
$A \models p\ CC \Longrightarrow A \models p$ *add-mset* $L\ CC$
using *true-clss-cls-mono-r*[*of* $A\ CC$ *add-mset* L {#}] **by** *simp*

lemma *true-clss-clss-union-l*[*simp*]:
$A \models ps\ CC \Longrightarrow A \cup B \models ps\ CC$
unfolding *true-clss-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-clss-union-l-r*[*simp*]:
$B \models ps\ CC \Longrightarrow A \cup B \models ps\ CC$
unfolding *true-clss-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-cls-in*[*simp*]:
$CC \in A \Longrightarrow A \models p\ CC$
unfolding *true-clss-cls-def true-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-cls-insert-l*[*simp*]:
$A \models p\ C \Longrightarrow$ *insert a* $A \models p\ C$
unfolding *true-clss-cls-def true-clss-def* **using** *total-over-m-union*
by (*metis Un-iff insert-is-Un sup.commute*)

lemma *true-clss-clss-insert-l*[*simp*]:
$A \models ps\ C \Longrightarrow$ *insert a* $A \models ps\ C$
unfolding *true-clss-cls-def true-clss-clss-def true-clss-def* **by** *blast*

lemma *true-clss-clss-union-and*[*iff*]:
$A \models ps\ C \cup D \longleftrightarrow (A \models ps\ C \wedge A \models ps\ D)$
proof
{
fix $A\ C\ D$ :: *'a clause-set*
assume A: $A \models ps\ C \cup D$
have $A \models ps\ C$
unfolding *true-clss-clss-def true-clss-cls-def insert-def total-over-m-insert*
proof (*intro allI impI*)
fix I
assume
totAC: *total-over-m* I $(A \cup C)$ **and**
cons: *consistent-interp* I **and**
I: $I \models s\ A$
then have *tot*: *total-over-m* $I\ A$ **and** *tot'*: *total-over-m* $I\ C$ **by** *auto*
obtain I' **where**
tot': *total-over-m* $(I \cup I')$ $(A \cup C \cup D)$ **and**
cons': *consistent-interp* $(I \cup I')$ **and**
H: $\forall x \in I'$. *atm-of* $x \in$ *atms-of-ms* $D \wedge$ *atm-of* $x \notin$ *atms-of-ms* $(A \cup C)$

using *total-over-m-consistent-extension*[*OF - cons*, *of* $A \cup C$] *tot tot'* **by** *blast*
moreover have $I \cup I' \models s\ A$ **using** I **by** *simp*
ultimately have $I \cup I' \models s\ C \cup D$ **using** A **unfolding** *true-clss-clss-def* **by** *auto*
then have $I \cup I' \models s\ C \cup D$ **by** *auto*
then show $I \models s\ C$ **using** *notin-vars-union-true-clss-true-clss*[*of* I'] H **by** *auto*
qed
} **note** $H = this$
assume $A \models ps\ C \cup D$
then show $A \models ps\ C \wedge A \models ps\ D$ **using** H[*of* A] *Un-commute*[*of* C D] **by** *metis*
next
assume $A \models ps\ C \wedge A \models ps\ D$
then show $A \models ps\ C \cup D$
unfolding *true-clss-clss-def* **by** *auto*
qed

lemma *true-clss-clss-insert*[*iff*]:
$A \models ps\ insert\ L\ Ls \longleftrightarrow (A \models p\ L \wedge A \models ps\ Ls)$
using *true-clss-clss-union-and*[*of* A $\{L\}$ Ls] **by** *auto*

lemma *true-clss-clss-subset*:
$A \subseteq B \Longrightarrow A \models ps\ CC \Longrightarrow B \models ps\ CC$
by (*metis subset-Un-eq true-clss-clss-union-l*)

Better suited as intro rule:

lemma *true-clss-clss-subsetI*:
$A \models ps\ CC \Longrightarrow A \subseteq B \Longrightarrow B \models ps\ CC$
by (*metis subset-Un-eq true-clss-clss-union-l*)

lemma *union-trus-clss-clss*[*simp*]: $A \cup B \models ps\ B$
unfolding *true-clss-clss-def* **by** *auto*

lemma *true-clss-clss-remove*[*simp*]:
$A \models ps\ B \Longrightarrow A \models ps\ B - C$
by (*metis Un-Diff-Int true-clss-clss-union-and*)

lemma *true-clss-clss-subsetE*:
$N \models ps\ B \Longrightarrow A \subseteq B \Longrightarrow N \models ps\ A$
by (*metis sup.orderE true-clss-clss-union-and*)

lemma *true-clss-clss-in-imp-true-clss-cls*:
assumes $N \models ps\ U$
and $A \in U$
shows $N \models p\ A$
using *assms mk-disjoint-insert* **by** *fastforce*

lemma *all-in-true-clss-clss*: $\forall x \in B.\ x \in A \Longrightarrow A \models ps\ B$
unfolding *true-clss-clss-def true-clss-def* **by** *auto*

lemma *true-clss-clss-left-right*:
assumes $A \models ps\ B$
and $A \cup B \models ps\ M$
shows $A \models ps\ M \cup B$
using *assms* **unfolding** *true-clss-clss-def* **by** *auto*

lemma *true-clss-clss-generalise-true-clss-clss*:
$A \cup C \models ps\ D \Longrightarrow B \models ps\ C \Longrightarrow A \cup B \models ps\ D$

proof −
  assume *a1*: $A \cup C$ ⊨*ps* D
  assume B ⊨*ps* C
  then have *f2*: $\bigwedge M.\ M \cup B$ ⊨*ps* C
    by (*meson true-clss-clss-union-l-r*)
  have $\bigwedge M.\ C \cup (M \cup A)$ ⊨*ps* D
    using *a1* **by** (*simp add*: *Un-commute sup-left-commute*)
  then show *?thesis*
    using *f2* **by** (*metis* (*no-types*) *Un-commute true-clss-clss-left-right true-clss-clss-union-and*)
qed

lemma *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*:
  assumes *D*: N ⊨*p add-mset* $(-L)$ D
  and *C*: N ⊨*p add-mset* L C
  shows N ⊨*p* $D + C$
  unfolding *true-clss-cls-def*
proof (*intro allI impI*)
  fix I
  assume
    tot: *total-over-m* I $(N \cup \{D + C\})$ **and**
    consistent-interp I **and**
    I ⊨*s* N
  {
    assume *L*: $L \in I \vee -L \in I$
    then have *total-over-m* I $\{D + \{\#-\ L\#\}\}$
      using *tot* **by** (*cases L*) *auto*
    then have $I \models D + \{\#-\ L\#\}$ **using** *D* ‹I ⊨*s* N› *tot* ‹*consistent-interp I*›
      unfolding *true-clss-cls-def* **by** *auto*
    moreover
      have *total-over-m* I $\{C + \{\#L\#\}\}$
        using *L tot* **by** (*cases L*) *auto*
      then have $I \models C + \{\#L\#\}$
        using *C* ‹I ⊨*s* N› *tot* ‹*consistent-interp I*› **unfolding** *true-clss-cls-def* **by** *auto*
    ultimately have $I \models D + C$ **using** ‹*consistent-interp I*› *consistent-interp-def* **by** *fastforce*
  }
  moreover {
    assume *L*: $L \notin I \wedge -L \notin I$
    let $?I' = I \cup \{L\}$
    have *consistent-interp* $?I'$ **using** *L* ‹*consistent-interp I*› **by** *auto*
    moreover have *total-over-m* $?I'$ $\{$*add-mset* $(-L)$ $D\}$
      using *tot* **unfolding** *total-over-m-def total-over-set-def* **by** (*auto simp add*: *atms-of-def*)
    moreover have *total-over-m* $?I'$ N **using** *tot* **using** *total-union* **by** *blast*
    moreover have $?I'$ ⊨*s* N **using** ‹I ⊨*s* N› **using** *true-clss-union-increase* **by** *blast*
    ultimately have $?I' \models$ *add-mset* $(-L)$ D
      using *D* **unfolding** *true-clss-cls-def* **by** *blast*
    then have $?I' \models D$ **using** *L* **by** *auto*
    moreover
      have *total-over-set* I (*atms-of* $(D + C)$) **using** *tot* **by** *auto*
      then have $L \notin\# D \wedge -L \notin\# D$
        using *L* **unfolding** *total-over-set-def atms-of-def* **by** (*cases L*) *force+*
    ultimately have $I \models D + C$ **unfolding** *true-cls-def* **by** *auto*
  }
  ultimately show $I \models D + C$ **by** *blast*
qed

lemma *true-cls-union-mset*[*iff*]: $I \models C \cup\# D \longleftrightarrow I \models C \vee I \models D$

unfolding *true-cls-def* **by** *force*

lemma *true-clss-cls-sup-iff-add*: $N \models p\ C \cup\#\ D \longleftrightarrow N \models p\ C + D$
by (*auto simp*: *true-clss-cls-def*)

lemma *true-clss-cls-union-mset-true-clss-cls-or-not-true-clss-cls-or*:
assumes
D: $N \models p$ *add-mset* $(-L)$ D **and**
C: $N \models p$ *add-mset* L C
shows $N \models p\ D \cup\#\ C$
using *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*[*OF assms*]
by (*subst true-clss-cls-sup-iff-add*)

lemma *true-clss-cls-tautology-iff*:
‹{} ⊨p *a* ⟷ *tautology a*› (**is** ‹*?A* ⟷ *?B*›)
proof
assume *?A*
then have *H*: ‹*total-over-set I* (*atms-of a*) ⟹ *consistent-interp I* ⟹ *I* ⊨ *a*› **for** *I*
by (*auto simp*: *true-clss-cls-def tautology-decomp add-mset-eq-add-mset*
dest!: *multi-member-split*)
show *?B*
unfolding *tautology-def*
proof (*intro allI impI*)
fix *I*
assume *tot*: ‹*total-over-set I* (*atms-of a*)›
let *?Iinter* = ‹*I* ∩ *uminus* ‘ *I*›
let *?I* = ‹*I* − *?Iinter* ∪ *Pos* ‘ *atm-of* ‘ *?Iinter*›
have ‹*total-over-set ?I* (*atms-of a*)›
using *tot* **by** (*force simp*: *total-over-set-def image-image Clausal-Logic.uminus-lit-swap*
simp: *image-iff*)
moreover have ‹*consistent-interp ?I*›
unfolding *consistent-interp-def image-iff*
apply *clarify*
subgoal for *L*
apply (*cases L*)
apply (*auto simp*: *consistent-interp-def uminus-lit-swap image-iff*)
apply (*case-tac xa*; *auto*; *fail*)+
done
done
ultimately have ‹*?I* ⊨ *a*›
using *H*[*of ?I*] **by** *fast*
moreover have ‹*?I* ⊆ *I*›
apply (*rule*)
subgoal for *x* **by** (*cases x*; *auto*; *rename-tac xb*; *case-tac xb*; *auto*)
done
ultimately show ‹*I* ⊨ *a*›
by (*blast intro*: *true-cls-mono-set-mset-l*)
qed
next
assume *?B*
then show ‹*?A*›
by (*auto simp*: *true-clss-cls-def tautology-decomp add-mset-eq-add-mset*
dest!: *multi-member-split*)
qed

```
lemma true-cls-mset-empty-iff [simp]: ‹{} ⊨m C ⟷ C = {#}›
  by (cases C) auto

lemma true-clss-mono-left:
  ‹I ⊨s A ⟹ I ⊆ J ⟹ J ⊨s A›
  by (metis sup.orderE true-clss-union-increase')

lemma true-cls-remove-alien:
  ‹I ⊨ N ⟷ {L. L ∈ I ∧ atm-of L ∈ atms-of N} ⊨ N›
  by (auto simp: true-cls-def dest: multi-member-split)

lemma true-clss-remove-alien:
  ‹I ⊨s N ⟷ {L. L ∈ I ∧ atm-of L ∈ atms-of-ms N} ⊨s N›
  by (auto simp: true-clss-def true-cls-def in-implies-atm-of-on-atms-of-ms
    dest: multi-member-split)

lemma true-clss-alt-def:
  ‹N ⊨p χ ⟷
    (∀ I. atms-of-s I = atms-of-ms (N ∪ {χ}) ⟶ consistent-interp I ⟶ I ⊨s N ⟶ I ⊨ χ)›
  apply (rule iffI)
  subgoal
    unfolding total-over-set-alt-def true-clss-cls-def total-over-m-alt-def
    by auto
  subgoal
    unfolding total-over-set-alt-def true-clss-cls-def total-over-m-alt-def
    apply (intro conjI impI allI)
    subgoal for I
      using consistent-interp-subset[of ‹{L ∈ I. atm-of L ∈ atms-of-ms (N ∪ {χ})}› I]
      true-clss-mono-left[of ‹{L ∈ I. atm-of L ∈ atms-of-ms N}› N
         ‹{L ∈ I. atm-of L ∈ atms-of-ms (N ∪ {χ})}›]
      true-clss-remove-alien[of I N]
    by (drule-tac x = ‹{L ∈ I. atm-of L ∈ atms-of-ms (N ∪ {χ})}› in spec)
      (auto dest: true-cls-mono-set-mset-l)
    done
  done

lemma true-clss-alt-def2:
  assumes ‹¬tautology χ›
  shows ‹N ⊨p χ ⟷ (∀ I. atms-of-s I = atms-of-ms N ⟶ consistent-interp I ⟶ I ⊨s N ⟶ I ⊨
χ)› (is ‹?A ⟷ ?B›)
proof (rule iffI)
  assume ?A
  then have H:
      ‹⋀I. atms-of-ms (N ∪ {χ}) ⊆ atms-of-s I ⟶
    consistent-interp I ⟶ I ⊨s N ⟶ I ⊨ χ›
    unfolding total-over-set-alt-def total-over-m-alt-def true-clss-cls-def by blast
  show ?B
    unfolding total-over-set-alt-def total-over-m-alt-def true-clss-cls-def
  proof (intro conjI impI allI)
    fix I :: ‹'a literal set›
    assume
      atms: ‹atms-of-s I = atms-of-ms N› and
      cons: ‹consistent-interp I› and
      ‹I ⊨s N›
    let ?I1 = ‹I ∪ uminus ` {L ∈ set-mset χ. atm-of L ∉ atms-of-s I}›
    have ‹atms-of-ms (N ∪ {χ}) ⊆ atms-of-s ?I1›
```

```
      by (auto simp add: atms in-image-uminus-uminus atm-iff-pos-or-neg-lit)
    moreover have ‹consistent-interp ?I1›
      using cons assms by (auto simp: consistent-interp-def)
        (rename-tac x; case-tac x; auto; fail)+
    moreover have ‹?I1 ⊨s N›
      using ‹I ⊨s N› by auto
    ultimately have ‹?I1 ⊨ χ›
      using H[of ?I1] by auto
    then show ‹I ⊨ χ›
      using assms by (auto simp: true-cls-def)
  qed
next
  assume ?B
  show ?A
    unfolding total-over-m-alt-def true-clss-alt-def
  proof (intro conjI impI allI)
    fix I :: ‹'a literal set›
    assume
      atms: ‹atms-of-s I = atms-of-ms (N ∪ {χ})› and
      cons: ‹consistent-interp I› and
      ‹I ⊨s N›
    let ?I1 = ‹{L ∈ I. atm-of L ∈ atms-of-ms N}›
    have ‹atms-of-s ?I1 = atms-of-ms N›
      using atms by (auto simp add: in-image-uminus-uminus atm-iff-pos-or-neg-lit)
    moreover have ‹consistent-interp ?I1›
      using cons assms by (auto simp: consistent-interp-def)
    moreover have ‹?I1 ⊨s N›
      using ‹I ⊨s N› by (subst (asm)true-clss-remove-alien)
    ultimately have ‹?I1 ⊨ χ›
      using ‹?B› by auto
    then show ‹I ⊨ χ›
      using assms by (auto simp: true-cls-def)
  qed
qed

lemma true-clss-restrict-iff:
  assumes ‹¬tautology χ›
  shows ‹N ⊨p χ ⟷ N ⊨p {#L ∈# χ. atm-of L ∈ atms-of-ms N#}› (is ‹?A ⟷ ?B›)
  apply (subst true-clss-alt-def2[OF assms])
  apply (subst true-clss-alt-def2)
  subgoal using not-tautology-mono[OF - assms] by (auto dest: not-tautology-minus)
  apply (rule HOL.iff-allI)
  apply (auto 5 5 simp: true-cls-def atms-of-s-def dest!: multi-member-split)
  done
```

This is a slightly restrictive theorem, that encompasses most useful cases. The assumption ¬ *tautology C* can be removed if the model *I* is total over the clause.

```
lemma true-clss-cls-true-clss-true-cls:
  assumes ‹N ⊨p C›
    ‹I ⊨s N› and
    cons: ‹consistent-interp I› and
    tauto: ‹¬tautology C›
  shows ‹I ⊨ C›
proof -
  let ?I = ‹I ∪ uminus ` {L ∈ set-mset C. atm-of L ∉ atms-of-s I}›
  let ?I2 = ‹?I ∪ Pos ` {L ∈ atms-of-ms N. L ∉ atms-of-s ?I}›
```

have ‹*total-over-m ?I2* (*N* ∪ {*C*})›
  by (*auto simp*: *total-over-m-alt-def atms-of-def in-image-uminus-uminus*
    dest!: *multi-member-split*)
moreover have ‹*consistent-interp ?I2*›
  using *cons tauto* **unfolding** *consistent-interp-def*
  apply (*intro allI*)
  apply (*case-tac L*)
  by (*auto simp*: *uminus-lit-swap eq-commute*[*of* ‹*Pos* -› ‹− -›]
    eq-commute[*of* ‹*Neg* -› ‹− -›])
moreover have ‹*?I2* ⊨*s N*›
  using ‹*I* ⊨*s N*› **by** *auto*
ultimately have ‹*?I2* ⊨ *C*›
  using *assms*(*1*) **unfolding** *true-clss-cls-def* **by** *fast*
then show *?thesis*
  using *tauto*
  by (*subst* (*asm*) *true-cls-remove-alien*)
    (*auto simp*: *true-cls-def in-image-uminus-uminus*)
qed

1.1.4 Subsumptions

lemma *subsumption-total-over-m*:
  assumes $A \subseteq\# B$
  shows *total-over-m I* {*B*} ⟹ *total-over-m I* {*A*}
  using *assms* **unfolding** *subset-mset-def total-over-m-def total-over-set-def*
  by (*auto simp add*: *mset-subset-eq-exists-conv*)

lemma *atms-of-replicate-mset-replicate-mset-uminus*[*simp*]:
  atms-of (*D* − *replicate-mset* (*count D L*) *L* − *replicate-mset* (*count D* (−*L*)) (−*L*))
  = *atms-of D* − {*atm-of L*}
  by (*auto simp*: *atm-of-eq-atm-of atms-of-def in-diff-count dest*: *in-diffD*)

lemma *subsumption-chained*:
  assumes
    ∀ *I*. *total-over-m I* {*D*} ⟶ *I* ⊨ *D* ⟶ *I* ⊨ φ **and**
    $C \subseteq\# D$
  shows (∀ *I*. *total-over-m I* {*C*} ⟶ *I* ⊨ *C* ⟶ *I* ⊨ φ) ∨ *tautology* φ
  using *assms*
proof (*induct card* {*Pos v* | *v*. *v* ∈ *atms-of D* ∧ *v* ∉ *atms-of C*} *arbitrary*: *D*
    rule: *nat-less-induct-case*)
  case *0* **note** *n* = *this*(*1*) **and** *H* = *this*(*2*) **and** *incl* = *this*(*3*)
  then have *atms-of D* ⊆ *atms-of C* **by** *auto*
  then have ∀ *I*. *total-over-m I* {*C*} ⟶ *total-over-m I* {*D*}
    unfolding *total-over-m-def total-over-set-def* **by** *auto*
  moreover have ∀ *I*. *I* ⊨ *C* ⟶ *I* ⊨ *D* **using** *incl true-cls-mono-leD* **by** *blast*
  ultimately show *?case* **using** *H* **by** *auto*
next
  case (*Suc n D*) **note** *IH* = *this*(*1*) **and** *card* = *this*(*2*) **and** *H* = *this*(*3*) **and** *incl* = *this*(*4*)
  let *?atms* = {*Pos v* |*v*. *v* ∈ *atms-of D* ∧ *v* ∉ *atms-of C*}
  have *finite ?atms* **by** *auto*
  then obtain *L* **where** *L*: *L* ∈ *?atms*
    using *card* **by** (*metis* (*no-types*, *lifting*) *Collect-empty-eq card-0-eq mem-Collect-eq*
      nat.simps(*3*))
  let *?D'* = *D* − *replicate-mset* (*count D L*) *L* − *replicate-mset* (*count D* (−*L*)) (−*L*)
  have *atms-of-D*: *atms-of-ms* {*D*} ⊆ *atms-of-ms* {*?D'*} ∪ {*atm-of L*}
    using *atms-of-replicate-mset-replicate-mset-uminus* **by** *force*

{
  fix *I*
  assume *total-over-m I* {*?D′*}
  then have *tot*: *total-over-m* (*I* ∪ {*L*}) {*D*}
    unfolding *total-over-m-def total-over-set-def* **using** *atms-of-D* **by** *auto*

  assume *IDL*: *I* ⊨ *?D′*
  then have *insert L I* ⊨ *D* **unfolding** *true-cls-def* **by** (*fastforce dest*: *in-diffD*)
  then have *insert L I* ⊨ *φ* **using** *H tot* **by** *auto*

  moreover
    have *tot′*: *total-over-m* (*I* ∪ {−*L*}) {*D*}
      using *tot* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
    have *I* ∪ {−*L*} ⊨ *D* **using** *IDL* **unfolding** *true-cls-def* **by** (*force dest*: *in-diffD*)
    then have *I* ∪ {−*L*} ⊨ *φ* **using** *H tot′* **by** *auto*
  ultimately have *I* ⊨ *φ* ∨ *tautology φ*
    using *L remove-literal-in-model-tautology* **by** *force*
} **note** *H′* = *this*

have *L* ∉# *C* **and** −*L* ∉# *C* **using** *L atm-iff-pos-or-neg-lit* **by** *force+*
then have *C-in-D′*: *C* ⊆# *?D′* **using** ‹*C* ⊆# *D*› **by** (*auto simp*: *subseteq-mset-def not-in-iff*)
have *card* {*Pos v* |*v*. *v* ∈ *atms-of ?D′* ∧ *v* ∉ *atms-of C*} <
  card {*Pos v* |*v*. *v* ∈ *atms-of D* ∧ *v* ∉ *atms-of C*}
  using *L* **unfolding** *atms-of-replicate-mset-replicate-mset-uminus*[*of D L*]
  by (*auto intro*!: *psubset-card-mono*)
then show *?case*
  using *IH C-in-D′ H′* **unfolding** *card*[*symmetric*] **by** *blast*
qed

1.1.5 Removing Duplicates

lemma *tautology-remdups-mset*[*iff*]:
  tautology (*remdups-mset C*) ⟷ *tautology C*
  unfolding *tautology-decomp* **by** *auto*

lemma *atms-of-remdups-mset*[*simp*]: *atms-of* (*remdups-mset C*) = *atms-of C*
  unfolding *atms-of-def* **by** *auto*

lemma *true-cls-remdups-mset*[*iff*]: *I* ⊨ *remdups-mset C* ⟷ *I* ⊨ *C*
  unfolding *true-cls-def* **by** *auto*

lemma *true-clss-cls-remdups-mset*[*iff*]: *A* ⊨*p remdups-mset C* ⟷ *A* ⊨*p C*
  unfolding *true-clss-cls-def total-over-m-def* **by** *auto*

1.1.6 Set of all Simple Clauses

A simple clause with respect to a set of atoms is such that

1. its atoms are included in the considered set of atoms;
2. it is not a tautology;
3. it does not contains duplicate literals.

   It corresponds to the clauses that cannot be simplified away in a calculus without considering the other clauses.

```
definition simple-clss :: 'v set ⇒ 'v clause set where
simple-clss atms = {C. atms-of C ⊆ atms ∧ ¬tautology C ∧ distinct-mset C}

lemma simple-clss-empty[simp]:
  simple-clss {} = {{#}}
  unfolding simple-clss-def by auto

lemma simple-clss-insert:
  assumes l ∉ atms
  shows simple-clss (insert l atms) =
    ((+) {#Pos l#}) ‘ (simple-clss atms)
    ∪ ((+) {#Neg l#}) ‘ (simple-clss atms)
    ∪ simple-clss atms(is ?I = ?U)
proof (standard; standard)
  fix C
  assume C ∈ ?I
  then have
    atms: atms-of C ⊆ insert l atms and
    taut: ¬tautology C and
    dist: distinct-mset C
    unfolding simple-clss-def by auto
  have H: ⋀x. x ∈# C ⟹ atm-of x ∈ insert l atms
    using atm-of-lit-in-atms-of atms by blast
  consider
      (Add) L where L ∈# C and L = Neg l ∨ L = Pos l
    | (No) Pos l ∉# C Neg l ∉# C
    by auto
  then show C ∈ ?U
    proof cases
      case Add
      then have LCL: L ∉# C − {#L#}
        using dist unfolding distinct-mset-def by (auto simp: not-in-iff)
      have LC: −L ∉# C
        using taut Add by auto
      obtain aa :: 'a where
       f4: (aa ∈ atms-of (remove1-mset L C) ⟶ aa ∈ atms) ⟶ atms-of (remove1-mset L C) ⊆ atms
        by (meson subset-iff)
      obtain ll :: 'a literal where
        aa ∉ atm-of ‘ set-mset (remove1-mset L C) ∨ aa = atm-of ll ∧ ll ∈# remove1-mset L C
        by blast
      then have atms-of (C − {#L#}) ⊆ atms
        using f4 Add LCL LC H unfolding atms-of-def by (metis H in-diffD insertE
          literal.exhaust-sel uminus-Neg uminus-Pos)
      moreover have ¬ tautology (C − {#L#})
        using taut by (metis Add(1) insert-DiffM tautology-add-mset)
      moreover have distinct-mset (C − {#L#})
        using dist by auto
      ultimately have (C − {#L#}) ∈ simple-clss atms
        using Add unfolding simple-clss-def by auto
      moreover have C = {#L#} + (C − {#L#})
        using Add by (auto simp: multiset-eq-iff)
      ultimately show ?thesis using Add by blast
    next
      case No
      then have C ∈ simple-clss atms
        using taut atms dist unfolding simple-clss-def
```

```
        by (auto simp: atm-iff-pos-or-neg-lit split: if-split-asm dest!: H)
      then show ?thesis by blast
    qed
next
  fix C
  assume C ∈ ?U
  then consider
      (Add) L C' where C = {#L#} + C' and C' ∈ simple-clss atms and
        L = Pos l ∨ L = Neg l
    | (No) C ∈ simple-clss atms
    by auto
  then show C ∈ ?I
    proof cases
      case No
      then show ?thesis unfolding simple-clss-def by auto
    next
      case (Add L C') note C' = this(1) and C = this(2) and L = this(3)
      then have
        atms: atms-of C' ⊆ atms and
        taut: ¬tautology C' and
        dist: distinct-mset C'
        unfolding simple-clss-def by auto
      have atms-of C ⊆ insert l atms
        using atms C' L by auto
      moreover have ¬ tautology C
        using taut C' L assms atms by (metis union-mset-add-mset-left add.left-neutral
            neg-lit-in-atms-of pos-lit-in-atms-of subsetCE tautology-add-mset
            uminus-Neg uminus-Pos)
      moreover have distinct-mset C
        using dist C' L by (metis union-mset-add-mset-left add.left-neutral assms atms
            distinct-mset-add-mset neg-lit-in-atms-of pos-lit-in-atms-of subsetCE)
      ultimately show ?thesis unfolding simple-clss-def by blast
    qed
qed

lemma simple-clss-finite:
  fixes atms :: 'v set
  assumes finite atms
  shows finite (simple-clss atms)
  using assms by (induction rule: finite-induct) (auto simp: simple-clss-insert)

lemma simple-clssE:
  assumes
    x ∈ simple-clss atms
  shows atms-of x ⊆ atms ∧ ¬tautology x ∧ distinct-mset x
  using assms unfolding simple-clss-def by auto

lemma cls-in-simple-clss:
  shows {#} ∈ simple-clss s
  unfolding simple-clss-def by auto

lemma simple-clss-card:
  fixes atms :: 'v set
  assumes finite atms
  shows card (simple-clss atms) ≤ (3::nat) ^ (card atms)
  using assms
```

proof (*induct atms rule*: *finite-induct*)
  case *empty*
  then show *?case* **by** *auto*
next
  case (*insert l C*) **note** *fin* = *this*(*1*) **and** *l* = *this*(*2*) **and** *IH* = *this*(*3*)
  have *notin*:
    $\bigwedge C'$. *add-mset* (*Pos l*) $C' \notin$ *simple-clss C*
    $\bigwedge C'$. *add-mset* (*Neg l*) $C' \notin$ *simple-clss C*
    using *l* **unfolding** *simple-clss-def* **by** *auto*
  have *H*: $\bigwedge C'$ *D*. {#*Pos l*#} + C' = {#*Neg l*#} + *D* $\Longrightarrow$ *D* $\in$ *simple-clss C* $\Longrightarrow$ *False*
    proof −
      fix C' *D*
      assume $C'D$: {#*Pos l*#} + C' = {#*Neg l*#} + *D* **and** *D*: *D* $\in$ *simple-clss C*
      then have *Pos l* $\in$# *D*
        by (*auto simp*: *add-mset-eq-add-mset-ne*)
      then have *l* $\in$ *atms-of D*
        by (*simp add*: *atm-iff-pos-or-neg-lit*)
      then show *False* **using** *D l* **unfolding** *simple-clss-def* **by** *auto*
    qed
  let *?P* = ((+) {#*Pos l*#}) ‘ (*simple-clss C*)
  let *?N* = ((+) {#*Neg l*#}) ‘ (*simple-clss C*)
  let *?O* = *simple-clss C*
  have *card* (*?P* $\cup$ *?N* $\cup$ *?O*) = *card* (*?P* $\cup$ *?N*) + *card ?O*
    apply (*subst card-Un-disjoint*)
    using *l fin* **by** (*auto simp*: *simple-clss-finite notin*)
  moreover have *card* (*?P* $\cup$ *?N*) = *card ?P* + *card ?N*
    apply (*subst card-Un-disjoint*)
    using *l fin H* **by** (*auto simp*: *simple-clss-finite notin*)
  moreover
    have *card ?P* = *card ?O*
      using *inj-on-iff-eq-card*[*of ?O* (+) {#*Pos l*#}]
      by (*auto simp*: *fin simple-clss-finite inj-on-def*)
  moreover have *card ?N* = *card ?O*
      using *inj-on-iff-eq-card*[*of ?O* (+) {#*Neg l*#}]
      by (*auto simp*: *fin simple-clss-finite inj-on-def*)
  moreover have (*3*::*nat*) ^ *card* (*insert l C*) = *3* ^ (*card C*) + *3* ^ (*card C*) + *3* ^ (*card C*)
    using *l* **by** (*simp add*: *fin mult-2-right numeral-3-eq-3*)
  ultimately show *?case* **using** *IH l* **by** (*auto simp*: *simple-clss-insert*)
qed

lemma *simple-clss-mono*:
  assumes *incl*: *atms* $\subseteq$ *atms'*
  shows *simple-clss atms* $\subseteq$ *simple-clss atms'*
  using *assms* **unfolding** *simple-clss-def* **by** *auto*

lemma *distinct-mset-not-tautology-implies-in-simple-clss*:
  assumes *distinct-mset* χ **and** $\neg$*tautology* χ
  shows $\chi \in$ *simple-clss* (*atms-of* χ)
  using *assms* **unfolding** *simple-clss-def* **by** *auto*

lemma *simplified-in-simple-clss*:
  assumes *distinct-mset-set* ψ **and** $\forall \chi \in \psi$. $\neg$*tautology* χ
  shows $\psi \subseteq$ *simple-clss* (*atms-of-ms* ψ)
  using *assms* **unfolding** *simple-clss-def*
  by (*auto simp*: *distinct-mset-set-def atms-of-ms-def*)

lemma *simple-clss-element-mono*:
‹$x \in$ *simple-clss* $A \Longrightarrow y \subseteq\# x \Longrightarrow y \in$ *simple-clss* A›
by (*auto simp*: *simple-clss-def atms-of-def intro*: *distinct-mset-mono*
dest: *not-tautology-mono*)

1.1.7 Experiment: Expressing the Entailments as Locales

locale *entail* =
fixes *entail* :: $'a$ *set* $\Rightarrow$ $'b$ $\Rightarrow$ *bool* (**infix** $\models e$ *50*)
assumes *entail-insert*[*simp*]: $I \neq \{\} \Longrightarrow$ *insert* L $I \models e\ x \longleftrightarrow \{L\} \models e\ x \lor I \models e\ x$
assumes *entail-union*[*simp*]: $I \models e\ A \Longrightarrow I \cup I' \models e\ A$
begin

definition *entails* :: $'a$ *set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ *bool* (**infix** $\models es$ *50*) **where**
$I \models es\ A \longleftrightarrow (\forall\, a \in A.\ I \models e\ a)$

lemma *entails-empty*[*simp*]:
$I \models es\ \{\}$
unfolding *entails-def* **by** *auto*

lemma *entails-single*[*iff*]:
$I \models es\ \{a\} \longleftrightarrow I \models e\ a$
unfolding *entails-def* **by** *auto*

lemma *entails-insert-l*[*simp*]:
$M \models es\ A \Longrightarrow$ *insert* L $M \models es\ A$
unfolding *entails-def* **by** (*metis Un-commute entail-union insert-is-Un*)

lemma *entails-union*[*iff*]: $I \models es\ CC \cup DD \longleftrightarrow I \models es\ CC \land I \models es\ DD$
unfolding *entails-def* **by** *blast*

lemma *entails-insert*[*iff*]: $I \models es$ *insert* $C\ DD \longleftrightarrow I \models e\ C \land I \models es\ DD$
unfolding *entails-def* **by** *blast*

lemma *entails-insert-mono*: $DD \subseteq CC \Longrightarrow I \models es\ CC \Longrightarrow I \models es\ DD$
unfolding *entails-def* **by** *blast*

lemma *entails-union-increase*[*simp*]:
assumes $I \models es\ \psi$
shows $I \cup I' \models es\ \psi$
using *assms* **unfolding** *entails-def* **by** *auto*

lemma *true-clss-commute-l*:
$I \cup I' \models es\ \psi \longleftrightarrow I' \cup I \models es\ \psi$
by (*simp add*: *Un-commute*)

lemma *entails-remove*[*simp*]: $I \models es\ N \Longrightarrow I \models es$ *Set.remove* $a\ N$
by (*simp add*: *entails-def*)

lemma *entails-remove-minus*[*simp*]: $I \models es\ N \Longrightarrow I \models es\ N - A$
by (*simp add*: *entails-def*)

end

interpretation *true-cls*: *entail true-cls*
by *standard* (*auto simp add*: *true-cls-def*)

1.1.8 Entailment to be extended

In some cases we want a more general version of entailment to have for example $\{\} \models \{\#L, -L\#\}$. This is useful when the model we are building might not be total (the literal L might have been definitely removed from the set of clauses), but we still want to have a property of entailment considering that theses removed literals are not important.

We can given a model I consider all the natural extensions: C is entailed by an extended I, if for all total extension of I, this model entails C.

definition *true-clss-ext* :: *'a literal set* ⇒ *'a clause set* ⇒ *bool* (**infix** ⊨*sext 49*)
where
I ⊨*sext N* ⟷ (∀ *J*. *I* ⊆ *J* ⟶ *consistent-interp J* ⟶ *total-over-m J N* ⟶ *J* ⊨*s N*)

lemma *true-clss-imp-true-cls-ext*:
  I⊨*s N* ⟹ *I* ⊨*sext N*
  unfolding *true-clss-ext-def* **by** (*metis sup.orderE true-clss-union-increase'*)

lemma *true-clss-ext-decrease-right-remove-r*:
  assumes *I* ⊨*sext N*
  shows *I* ⊨*sext N* − {*C*}
  unfolding *true-clss-ext-def*
proof (*intro allI impI*)
  fix *J*
  assume
    I ⊆ *J* **and**
    cons: *consistent-interp J* **and**
    tot: *total-over-m J* (*N* − {*C*})
  let *?J* = *J* ∪ {*Pos* (*atm-of P*)|*P*. *P* ∈# *C* ∧ *atm-of P* ∉ *atm-of* ‘ *J*}
  have *I* ⊆ *?J* **using** ‹*I* ⊆ *J*› **by** *auto*
  moreover have *consistent-interp ?J*
    using *cons* **unfolding** *consistent-interp-def* **apply** (*intro allI*)
    by (*rename-tac L*, *case-tac L*) (*fastforce simp add*: *image-iff*)+
  moreover have *total-over-m ?J N*
    using *tot* **unfolding** *total-over-m-def total-over-set-def atms-of-ms-def*
    apply *clarify*
    apply (*rename-tac l a*, *case-tac a* ∈ *N* − {*C*})
      apply (*auto*; *fail*)
    using *atms-of-s-def atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*
    by (*fastforce simp*: *atms-of-def*)
  ultimately have *?J* ⊨*s N*
    using *assms* **unfolding** *true-clss-ext-def* **by** *blast*
  then have *?J* ⊨*s N* − {*C*} **by** *auto*
  have {*v* ∈ *?J*. *atm-of v* ∈ *atms-of-ms* (*N* − {*C*})} ⊆ *J*
    using *tot* **unfolding** *total-over-m-def total-over-set-def*
    by (*auto intro*!: *rev-image-eqI*)
  then show *J* ⊨*s N* − {*C*}
    using *true-clss-remove-unused*[*OF* ‹*?J* ⊨*s N* − {*C*}›] **unfolding** *true-clss-def*
    by (*meson true-cls-mono-set-mset-l*)
qed

lemma *consistent-true-clss-ext-satisfiable*:
  assumes *consistent-interp I* **and** *I* ⊨*sext A*
  shows *satisfiable A*
  by (*metis Un-empty-left assms satisfiable-carac subset-Un-eq sup.left-idem*
    total-over-m-consistent-extension total-over-m-empty true-clss-ext-def)

lemma *not-consistent-true-clss-ext*:
assumes ¬*consistent-interp I*
shows *I* ⊨*sext A*
by (*meson assms consistent-interp-subset true-clss-ext-def*)

lemma *inj-on-Pos*: ‹*inj-on Pos A*› **and**
inj-on-Neg: ‹*inj-on Neg A*›
by (*auto simp*: *inj-on-def*)

lemma *inj-on-uminus-lit*: ‹*inj-on uminus A*› **for** *A* :: ‹*'a literal set*›
by (*auto simp*: *inj-on-def*)

end

1.2 Partial Annotated Herbrand Interpretation

We here define decided literals (that will be used in both DPLL and CDCL) and the entailment corresponding to it.

theory *Partial-Annotated-Herbrand-Interpretation*
imports
Partial-Herbrand-Interpretation
begin

1.2.1 Decided Literals

Definition

datatype (*'v*, *'w*, *'mark*) *annotated-lit* =
is-decided: *Decided* (*lit-dec*: *'v*) |
is-proped: *Propagated* (*lit-prop*: *'w*) (*mark-of*: *'mark*)

type-synonym (*'v*, *'w*, *'mark*) *annotated-lits* = ‹(*'v*, *'w*, *'mark*) *annotated-lit list*›
type-synonym (*'v*, *'mark*) *ann-lit* = ‹(*'v literal*, *'v literal*, *'mark*) *annotated-lit*›

lemma *ann-lit-list-induct*[*case-names Nil Decided Propagated*]:
assumes
‹*P* []› **and**
‹⋀*L xs*. *P xs* ⟹ *P* (*Decided L* # *xs*)› **and**
‹⋀*L m xs*. *P xs* ⟹ *P* (*Propagated L m* # *xs*)›
shows ‹*P xs*›
using *assms* **apply** (*induction xs*, *simp*)
by (*rename-tac a xs*, *case-tac a*) *auto*

lemma *is-decided-ex-Decided*:
‹*is-decided L* ⟹ (⋀*K*. *L* = *Decided K* ⟹ *P*) ⟹ *P*›
by (*cases L*) *auto*

lemma *is-propedE*: ‹*is-proped L* ⟹ (⋀*K C*. *L* = *Propagated K C* ⟹ *P*) ⟹ *P*›
by (*cases L*) *auto*

lemma *is-decided-no-proped-iff*: ‹*is-decided L* ⟷ ¬*is-proped L*›
by (*cases L*) *auto*

lemma *not-is-decidedE*:
‹¬*is-decided E* ⟹ (⋀*K C*. *E* = *Propagated K C* ⟹ *thesis*) ⟹ *thesis*›
by (*cases E*) *auto*

type-synonym (*'v*, *'m*) *ann-lits* = ‹(*'v*, *'m*) *ann-lit list*›

fun *lit-of* :: ‹(*'a*, *'a*, *'mark*) *annotated-lit* ⇒ *'a*› **where**
‹*lit-of* (*Decided L*) = *L*› |
‹*lit-of* (*Propagated L* -) = *L*›

definition *lits-of* :: ‹(*'a*, *'b*) *ann-lit set* ⇒ *'a literal set*› **where**
‹*lits-of Ls* = *lit-of* ‘ *Ls*›

abbreviation *lits-of-l* :: ‹(*'a*, *'b*) *ann-lits* ⇒ *'a literal set*› **where**
‹*lits-of-l Ls* ≡ *lits-of* (*set Ls*)›

lemma *lits-of-l-empty*[*simp*]:
‹*lits-of* {} = {}›
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-insert*[*simp*]:
‹*lits-of* (*insert L Ls*) = *insert* (*lit-of L*) (*lits-of Ls*)›
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-l-Un*[*simp*]:
‹*lits-of* (*l* ∪ *l'*) = *lits-of l* ∪ *lits-of l'*›
unfolding *lits-of-def* **by** *auto*

lemma *finite-lits-of-def*[*simp*]:
‹*finite* (*lits-of-l L*)›
unfolding *lits-of-def* **by** *auto*

abbreviation *unmark* **where**
‹*unmark* ≡ (λ*a*. {#*lit-of a*#})›

abbreviation *unmark-s* **where**
‹*unmark-s M* ≡ *unmark* ‘ *M*›

abbreviation *unmark-l* **where**
‹*unmark-l M* ≡ *unmark-s* (*set M*)›

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of*[*simp*]:
‹*atms-of-ms* (*unmark-l M'*) = *atm-of* ‘ *lits-of-l M'*›
unfolding *atms-of-ms-def lits-of-def* **by** *auto*

lemma *lits-of-l-empty-is-empty*[*iff*]:
‹*lits-of-l M* = {} ⟷ *M* = []›
by (*induct M*) (*auto simp*: *lits-of-def*)

lemma *in-unmark-l-in-lits-of-l-iff*: ‹{#*L*#} ∈ *unmark-l M* ⟷ *L* ∈ *lits-of-l M*›
by (*induction M*) *auto*

lemma *atm-lit-of-set-lits-of-l*:
(λ*l*. *atm-of* (*lit-of l*)) ‘ *set xs* = *atm-of* ‘ *lits-of-l xs*
unfolding *lits-of-def* **by** *auto*

Entailment

definition *true-annot* :: ‹(′a, ′m) ann-lits ⇒ ′a clause ⇒ bool› (**infix** ⊨a *49*) **where**
‹I ⊨a C ⟷ (lits-of-l I) ⊨ C›

definition *true-annots* :: ‹(′a, ′m) ann-lits ⇒ ′a clause-set ⇒ bool› (**infix** ⊨as *49*) **where**
‹I ⊨as CC ⟷ (∀ C ∈ CC. I ⊨a C)›

lemma *true-annot-empty-model*[*simp*]:
‹¬[] ⊨a ψ›
unfolding *true-annot-def true-cls-def* **by** *simp*

lemma *true-annot-empty*[*simp*]:
‹¬I ⊨a {#}›
unfolding *true-annot-def true-cls-def* **by** *simp*

lemma *empty-true-annots-def*[*iff*]:
‹[] ⊨as ψ ⟷ ψ = {}›
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-empty*[*simp*]:
‹I ⊨as {}›
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-single-true-annot*[*iff*]:
‹I ⊨as {C} ⟷ I ⊨a C›
unfolding *true-annots-def* **by** *auto*

lemma *true-annot-insert-l*[*simp*]:
‹M ⊨a A ⟹ L # M ⊨a A›
unfolding *true-annot-def* **by** *auto*

lemma *true-annots-insert-l* [*simp*]:
‹M ⊨as A ⟹ L # M ⊨as A›
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-union*[*iff*]:
‹M ⊨as A ∪ B ⟷ (M ⊨as A ∧ M ⊨as B)›
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-insert*[*iff*]:
‹M ⊨as insert a A ⟷ (M ⊨a a ∧ M ⊨as A)›
unfolding *true-annots-def* **by** *auto*

lemma *true-annot-append-l*:
‹M ⊨a A ⟹ M′ @ M ⊨a A›
unfolding *true-annot-def* **by** *auto*

lemma *true-annots-append-l*:
‹M ⊨as A ⟹ M′ @ M ⊨as A›
unfolding *true-annots-def* **by** (*auto simp*: *true-annot-append-l*)

Link between ⊨*as* and ⊨*s*:

lemma *true-annots-true-cls*:
‹I ⊨as CC ⟷ lits-of-l I ⊨s CC›
unfolding *true-annots-def Ball-def true-annot-def true-clss-def* **by** *auto*

lemma *in-lit-of-true-annot*:
  ‹$a \in$ *lits-of-l* $M \longleftrightarrow M \models a$ {#a#}›
  unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annot-lit-of-notin-skip*:
  ‹L # $M \models a$ $A \Longrightarrow$ *lit-of* $L \notin$# $A \Longrightarrow M \models a$ A›
  unfolding *true-annot-def true-cls-def* **by** *auto*

lemma *true-clss-singleton-lit-of-implies-incl*:
  ‹$I \models s$ *unmark-l MLs* $\Longrightarrow$ *lits-of-l MLs* $\subseteq I$›
  unfolding *true-clss-def lits-of-def* **by** *auto*

lemma *true-annot-true-clss-cls*:
  ‹*MLs* $\models a$ $\psi \Longrightarrow$ *set* (*map unmark MLs*) $\models p$ ψ›
  unfolding *true-annot-def true-clss-cls-def true-cls-def*
  by (*auto dest*: *true-clss-singleton-lit-of-implies-incl*)

lemma *true-annots-true-clss-cls*:
  ‹*MLs* $\models as$ $\psi \Longrightarrow$ *set* (*map unmark MLs*) $\models ps$ ψ›
  by (*auto*
    dest: *true-clss-singleton-lit-of-implies-incl*
    simp add: *true-clss-def true-annots-def true-annot-def lits-of-def true-cls-def*
    true-clss-clss-def)

lemma *true-annots-decided-true-cls*[*iff*]:
  ‹*map Decided M* $\models as$ $N \longleftrightarrow$ *set M* $\models s$ N›
proof −
  have ∗: ‹*lit-of* ‘ *Decided* ‘ *set M* = *set M*› **unfolding** *lits-of-def* **by** *force*
  show *?thesis* **by** (*simp add*: *true-annots-true-cls* ∗ *lits-of-def*)
qed

lemma *true-annot-singleton*[*iff*]: ‹$M \models a$ {#L#} $\longleftrightarrow L \in$ *lits-of-l* M›
  unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annots-true-clss-clss*:
  ‹$A \models as$ $\Psi \Longrightarrow$ *unmark-l* $A \models ps$ Ψ›
  unfolding *true-clss-clss-def true-annots-def true-clss-def*
  by (*auto dest*!: *true-clss-singleton-lit-of-implies-incl*
    simp: *lits-of-def true-annot-def true-cls-def*)

lemma *true-annot-commute*:
  ‹M @ $M' \models a$ $D \longleftrightarrow M'$ @ $M \models a$ D›
  unfolding *true-annot-def* **by** (*simp add*: *Un-commute*)

lemma *true-annots-commute*:
  ‹M @ $M' \models as$ $D \longleftrightarrow M'$ @ $M \models as$ D›
  unfolding *true-annots-def* **by** (*auto simp*: *true-annot-commute*)

lemma *true-annot-mono*[*dest*]:
  ‹*set* $I \subseteq$ *set* $I' \Longrightarrow I \models a$ $N \Longrightarrow I' \models a$ N›
  using *true-cls-mono-set-mset-l* **unfolding** *true-annot-def lits-of-def*
  by (*metis* (*no-types*) *Un-commute Un-upper1 image-Un sup.orderE*)

lemma *true-annots-mono*:
  ‹*set* $I \subseteq$ *set* $I' \Longrightarrow I \models as$ $N \Longrightarrow I' \models as$ N›

unfolding *true-annots-def* **by** *auto*

Defined and Undefined Literals

We introduce the functions *defined-lit* and *undefined-lit* to know whether a literal is defined with respect to a list of decided literals (aka a trail in most cases).

Remark that *undefined* already exists and is a completely different Isabelle function.

definition *defined-lit* :: ‹('*a literal*, '*a literal*, '*m*) *annotated-lits* ⇒ '*a literal* ⇒ *bool*›
  where
‹*defined-lit I L* ⟷ (*Decided L* ∈ *set I*) ∨ (∃ *P*. *Propagated L P* ∈ *set I*)
  ∨ (*Decided* (−*L*) ∈ *set I*) ∨ (∃ *P*. *Propagated* (−*L*) *P* ∈ *set I*)›

abbreviation *undefined-lit* :: ‹('*a literal*, '*a literal*, '*m*) *annotated-lits* ⇒ '*a literal* ⇒ *bool*›
where ‹*undefined-lit I L* ≡ ¬*defined-lit I L*›

lemma *defined-lit-rev*[*simp*]:
  ‹*defined-lit* (*rev M*) *L* ⟷ *defined-lit M L*›
  unfolding *defined-lit-def* **by** *auto*

lemma *atm-imp-decided-or-proped*:
  assumes ‹*x* ∈ *set I*›
  shows
    ‹(*Decided* (− *lit-of x*) ∈ *set I*)
    ∨ (*Decided* (*lit-of x*) ∈ *set I*)
    ∨ (∃ *l*. *Propagated* (− *lit-of x*) *l* ∈ *set I*)
    ∨ (∃ *l*. *Propagated* (*lit-of x*) *l* ∈ *set I*)›
  using *assms* **by** (*metis* (*full-types*) *lit-of.elims*)

lemma *literal-is-lit-of-decided*:
  assumes ‹*L* = *lit-of x*›
  shows ‹(*x* = *Decided L*) ∨ (∃ *l'*. *x* = *Propagated L l'*)›
  using *assms* **by** (*cases x*) *auto*

lemma *true-annot-iff-decided-or-true-lit*:
  ‹*defined-lit I L* ⟷ (*lits-of-l I* ⊨*l L* ∨ *lits-of-l I* ⊨*l* −*L*)›
  unfolding *defined-lit-def* **by** (*auto simp add*: *lits-of-def rev-image-eqI*
    dest!: *literal-is-lit-of-decided*)

lemma *consistent-inter-true-annots-satisfiable*:
  ‹*consistent-interp* (*lits-of-l I*) ⟹ *I* ⊨*as N* ⟹ *satisfiable N*›
  by (*simp add*: *true-annots-true-cls*)

lemma *defined-lit-map*:
  ‹*defined-lit Ls L* ⟷ *atm-of L* ∈ (λ*l*. *atm-of* (*lit-of l*)) ‘ *set Ls*›
 unfolding *defined-lit-def* **apply** (*rule iffI*)
   using *image-iff* **apply** *fastforce*
 by (*fastforce simp add*: *atm-of-eq-atm-of dest*: *atm-imp-decided-or-proped*)

lemma *defined-lit-uminus*[*iff*]:
  ‹*defined-lit I* (−*L*) ⟷ *defined-lit I L*›
  unfolding *defined-lit-def* **by** *auto*

lemma *Decided-Propagated-in-iff-in-lits-of-l*:
  ‹*defined-lit I L* ⟷ (*L* ∈ *lits-of-l I* ∨ −*L* ∈ *lits-of-l I*)›
  unfolding *lits-of-def* **by** (*metis lits-of-def true-annot-iff-decided-or-true-lit true-lit-def*)

lemma *consistent-add-undefined-lit-consistent*[*simp*]:
assumes
‹*consistent-interp* (*lits-of-l Ls*)› **and**
‹*undefined-lit Ls L*›
shows ‹*consistent-interp* (*insert L* (*lits-of-l Ls*))›
using *assms* **unfolding** *consistent-interp-def* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)

lemma *decided-empty*[*simp*]:
‹¬*defined-lit* [] *L*›
unfolding *defined-lit-def* **by** *simp*

lemma *undefined-lit-single*[*iff*]:
‹*defined-lit* [*L*] *K* ⟷ *atm-of* (*lit-of L*) = *atm-of K*›
by (*auto simp*: *defined-lit-map*)

lemma *undefined-lit-cons*[*iff*]:
‹*undefined-lit* (*L* # *M*) *K* ⟷ *atm-of* (*lit-of L*) ≠ *atm-of K* ∧ *undefined-lit M K*›
by (*auto simp*: *defined-lit-map*)

lemma *undefined-lit-append*[*iff*]:
‹*undefined-lit* (*M* @ *M'*) *K* ⟷ *undefined-lit M K* ∧ *undefined-lit M' K*›
by (*auto simp*: *defined-lit-map*)

lemma *defined-lit-cons*:
‹*defined-lit* (*L* # *M*) *K* ⟷ *atm-of* (*lit-of L*) = *atm-of K* ∨ *defined-lit M K*›
by (*auto simp*: *defined-lit-map*)

lemma *defined-lit-append*:
‹*defined-lit* (*M* @ *M'*) *K* ⟷ *defined-lit M K* ∨ *defined-lit M' K*›
by (*auto simp*: *defined-lit-map*)

lemma *in-lits-of-l-defined-litD*: ‹*L-max* ∈ *lits-of-l M* ⟹ *defined-lit M L-max*›
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)

lemma *undefined-notin*: ‹*undefined-lit M* (*lit-of x*) ⟹ *x* ∉ *set M*› **for** *x M*
by (*metis in-lits-of-l-defined-litD insert-iff lits-of-insert mk-disjoint-insert*)

lemma *uminus-lits-of-l-definedD*:
‹−*x* ∈ *lits-of-l M* ⟹ *defined-lit M x*›
by (*simp add*: *Decided-Propagated-in-iff-in-lits-of-l*)

lemma *defined-lit-Neg-Pos-iff*:
‹*defined-lit M* (*Neg A*) ⟷ *defined-lit M* (*Pos A*)›
by (*simp add*: *defined-lit-map*)

lemma *defined-lit-Pos-atm-iff*[*simp*]:
‹*defined-lit M1* (*Pos* (*atm-of x*)) ⟷ *defined-lit M1 x*›
by (*cases x*) (*auto simp*: *defined-lit-Neg-Pos-iff*)

lemma *defined-lit-mono*:
‹*defined-lit M2 L* ⟹ *set M2* ⊆ *set M3* ⟹ *defined-lit M3 L*›
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)

lemma *defined-lit-nth*:
‹*n* < *length M2* ⟹ *defined-lit M2* (*lit-of* (*M2* ! *n*))›

by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l lits-of-def*)

1.2.2 Backtracking

fun *backtrack-split* :: ‹('*a*, '*v*, '*m*) *annotated-lits*
⇒ ('*a*, '*v*, '*m*) *annotated-lits* × ('*a*, '*v*, '*m*) *annotated-lits*› **where**
‹*backtrack-split* [] = ([], [])› |
‹*backtrack-split* (*Propagated L P* # *mlits*) = *apfst* ((#) (*Propagated L P*)) (*backtrack-split mlits*)› |
‹*backtrack-split* (*Decided L* # *mlits*) = ([], *Decided L* # *mlits*)›

lemma *backtrack-split-fst-not-decided*: ‹*a* ∈ *set* (*fst* (*backtrack-split l*)) ⟹ ¬*is-decided a*›
by (*induct l rule*: *ann-lit-list-induct*) *auto*

lemma *backtrack-split-snd-hd-decided*:
‹*snd* (*backtrack-split l*) ≠ [] ⟹ *is-decided* (*hd* (*snd* (*backtrack-split l*)))›
by (*induct l rule*: *ann-lit-list-induct*) *auto*

lemma *backtrack-split-list-eq*[*simp*]:
‹*fst* (*backtrack-split l*) @ (*snd* (*backtrack-split l*)) = *l*›
by (*induct l rule*: *ann-lit-list-induct*) *auto*

lemma *backtrack-snd-empty-not-decided*:
‹*backtrack-split M* = (*M''*, []) ⟹ ∀ *l*∈*set M*. ¬ *is-decided l*›
by (*metis append-Nil2 backtrack-split-fst-not-decided backtrack-split-list-eq snd-conv*)

lemma *backtrack-split-some-is-decided-then-snd-has-hd*:
‹∃ *l*∈*set M*. *is-decided l* ⟹ ∃ *M' L' M''*. *backtrack-split M* = (*M''*, *L'* # *M'*)›
by (*metis backtrack-snd-empty-not-decided list.exhaust prod.collapse*)

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:
‹*backtrack-split M* = (*takeWhile* (*Not o is-decided*) *M*, *dropWhile* (*Not o is-decided*) *M*)›
by (*induction M rule*: *ann-lit-list-induct*) *auto*

1.2.3 Decomposition with respect to the First Decided Literals

In this section we define a function that returns a decomposition with the first decided literal. This function is useful to define the backtracking of DPLL.

Definition

The pattern *get-all-ann-decomposition* [] = [([], [])] is necessary otherwise, we can call the *hd* function in the other pattern.

fun *get-all-ann-decomposition* :: ‹('*a*, '*b*, '*m*) *annotated-lits*
⇒ (('*a*, '*b*, '*m*) *annotated-lits* × ('*a*, '*b*, '*m*) *annotated-lits*) *list*› **where**
‹*get-all-ann-decomposition* (*Decided L* # *Ls*) =
(*Decided L* # *Ls*, []) # *get-all-ann-decomposition Ls*› |
‹*get-all-ann-decomposition* (*Propagated L P*# *Ls*) =
(*apsnd* ((#) (*Propagated L P*)) (*hd* (*get-all-ann-decomposition Ls*)))
tl (*get-all-ann-decomposition Ls*)› |
‹*get-all-ann-decomposition* [] = [([], [])]›

value ‹*get-all-ann-decomposition* [*Propagated A5 B5*, *Decided C4*, *Propagated A3 B3*, *Propagated A2 B2*, *Decided C1*, *Propagated A0 B0*]›

Now we can prove several simple properties about the function.

lemma *get-all-ann-decomposition-never-empty*[*iff*]:
‹*get-all-ann-decomposition M* = [] ⟷ *False*›
by (*induct M*, *simp*) (*rename-tac a xs*, *case-tac a*, *auto*)

lemma *get-all-ann-decomposition-never-empty-sym*[*iff*]:
‹[] = *get-all-ann-decomposition M* ⟷ *False*›
using *get-all-ann-decomposition-never-empty*[*of M*] **by** *presburger*

lemma *get-all-ann-decomposition-decomp*:
‹*hd* (*get-all-ann-decomposition S*) = (*a*, *c*) ⟹ *S* = *c* @ *a*›
proof (*induct S arbitrary*: *a c*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x A*)
then show *?case* **by** (*cases x*; *cases* ‹*hd* (*get-all-ann-decomposition A*)›) *auto*
qed

lemma *get-all-ann-decomposition-backtrack-split*:
‹*backtrack-split S* = (*M*, *M'*) ⟷ *hd* (*get-all-ann-decomposition S*) = (*M'*, *M*)›
proof (*induction S arbitrary*: *M M'*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a S*)
then show *?case* **using** *backtrack-split-takeWhile-dropWhile* **by** (*cases a*) *force+*
qed

lemma *get-all-ann-decomposition-Nil-backtrack-split-snd-Nil*:
‹*get-all-ann-decomposition S* = [([], *A*)] ⟹ *snd* (*backtrack-split S*) = []›
by (*simp add*: *get-all-ann-decomposition-backtrack-split sndI*)

This functions says that the first element is either empty or starts with a decided element of the list.

lemma *get-all-ann-decomposition-length-1-fst-empty-or-length-1*:
assumes ‹*get-all-ann-decomposition M* = (*a*, *b*) # []›
shows ‹*a* = [] ∨ (*length a* = *1* ∧ *is-decided* (*hd a*) ∧ *hd a* ∈ *set M*)›
using *assms*
proof (*induct M arbitrary*: *a b rule*: *ann-lit-list-induct*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Decided L mark*)
then show *?case* **by** *simp*
next
case (*Propagated L mark M*)
then show *?case* **by** (*cases* ‹*get-all-ann-decomposition M*›) *force+*
qed

lemma *get-all-ann-decomposition-fst-empty-or-hd-in-M*:
assumes ‹*get-all-ann-decomposition M* = (*a*, *b*) # *l*›
shows ‹*a* = [] ∨ (*is-decided* (*hd a*) ∧ *hd a* ∈ *set M*)›
using *assms*

```
proof (induct M arbitrary: a b rule: ann-lit-list-induct)
  case Nil
  then show ?case by auto
next
  case (Decided L ann xs)
  then show ?case by auto
next
  case (Propagated L m xs) note IH = this(1) and d = this(2)
  then show ?case
    using IH[of ‹fst (hd (get-all-ann-decomposition xs))› ‹snd (hd(get-all-ann-decomposition xs))›]
    by (cases ‹get-all-ann-decomposition xs›; cases a) auto
qed

lemma get-all-ann-decomposition-snd-not-decided:
  assumes ‹(a, b) ∈ set (get-all-ann-decomposition M)›
  and ‹L ∈ set b›
  shows ‹¬is-decided L›
  using assms apply (induct M arbitrary: a b rule: ann-lit-list-induct, simp)
  by (rename-tac L' xs a b, case-tac ‹get-all-ann-decomposition xs›; fastforce)+

lemma tl-get-all-ann-decomposition-skip-some:
  assumes ‹x ∈ set (tl (get-all-ann-decomposition M1))›
  shows ‹x ∈ set (tl (get-all-ann-decomposition (M0 @ M1)))›
  using assms
  by (induct M0 rule: ann-lit-list-induct)
     (auto simp add: list.set-sel(2))

lemma hd-get-all-ann-decomposition-skip-some:
  assumes ‹(x, y) = hd (get-all-ann-decomposition M1)›
  shows ‹(x, y) ∈ set (get-all-ann-decomposition (M0 @ Decided K # M1))›
  using assms
proof (induction M0 rule: ann-lit-list-induct)
  case Nil
  then show ?case by auto
next
  case (Decided L M0)
  then show ?case by auto
next
  case (Propagated L C M0) note xy = this(1)[OF this(2−)] and hd = this(2)
  then show ?case
    by (cases ‹get-all-ann-decomposition (M0 @ Decided K # M1)›)
       (auto dest!: get-all-ann-decomposition-decomp
          arg-cong[of ‹get-all-ann-decomposition -› - hd])
qed

lemma in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend:
  ‹(a, b) ∈ set (get-all-ann-decomposition M') ⟹
    ∃ b'. (a, b' @ b) ∈ set (get-all-ann-decomposition (M @ M'))›
  apply (induction M rule: ann-lit-list-induct)
    apply (metis append-Nil)
   apply auto[]
  by (rename-tac L' m xs, case-tac ‹get-all-ann-decomposition (xs @ M')›) auto

lemma in-get-all-ann-decomposition-decided-or-empty:
  assumes ‹(a, b) ∈ set (get-all-ann-decomposition M)›
  shows ‹a = [] ∨ (is-decided (hd a))›
```

```
  using assms
proof (induct M arbitrary: a b rule: ann-lit-list-induct)
  case Nil then show ?case by simp
next
  case (Decided l M)
  then show ?case by auto
next
  case (Propagated l mark M)
  then show ?case by (cases ‹get-all-ann-decomposition M›) force+
qed

lemma get-all-ann-decomposition-remove-undecided-length:
  assumes ‹∀ l ∈ set M'. ¬is-decided l›
  shows ‹length (get-all-ann-decomposition (M' @ M'')) = length (get-all-ann-decomposition M'')›
  using assms by (induct M' arbitrary: M'' rule: ann-lit-list-induct) auto

lemma get-all-ann-decomposition-not-is-decided-length:
  assumes ‹∀ l ∈ set M'. ¬is-decided l›
  shows ‹1 + length (get-all-ann-decomposition (Propagated (−L) P # M))
 = length (get-all-ann-decomposition (M' @ Decided L # M))›
 using assms get-all-ann-decomposition-remove-undecided-length by fastforce

lemma get-all-ann-decomposition-last-choice:
  assumes ‹tl (get-all-ann-decomposition (M' @ Decided L # M)) ≠ []›
  and ‹∀ l ∈ set M'. ¬is-decided l›
  and ‹hd (tl (get-all-ann-decomposition (M' @ Decided L # M))) = (M0', M0)›
  shows ‹hd (get-all-ann-decomposition (Propagated (−L) P # M)) = (M0', Propagated (−L) P #
M0)›
  using assms by (induct M' rule: ann-lit-list-induct) auto

lemma get-all-ann-decomposition-except-last-choice-equal:
  assumes ‹∀ l ∈ set M'. ¬is-decided l›
  shows ‹tl (get-all-ann-decomposition (Propagated (−L) P # M))
 = tl (tl (get-all-ann-decomposition (M' @ Decided L # M)))›
  using assms by (induct M' rule: ann-lit-list-induct) auto

lemma get-all-ann-decomposition-hd-hd:
  assumes ‹get-all-ann-decomposition Ls = (M, C) # (M0, M0') # l›
  shows ‹tl M = M0' @ M0 ∧ is-decided (hd M)›
  using assms
proof (induct Ls arbitrary: M C M0 M0' l)
  case Nil
  then show ?case by simp
next
  case (Cons a Ls M C M0 M0' l) note IH = this(1) and g = this(2)
  { fix L ann level
    assume a: ‹a = Decided L›
    have ‹Ls = M0' @ M0›
      using g a by (force intro: get-all-ann-decomposition-decomp)
    then have ‹tl M = M0' @ M0 ∧ is-decided (hd M)› using g a by auto
  }
  moreover {
    fix L P
    assume a: ‹a = Propagated L P›
    have ‹tl M = M0' @ M0 ∧ is-decided (hd M)›
      using IH Cons.prems unfolding a by (cases ‹get-all-ann-decomposition Ls›) auto
```

}
ultimately show *?case* **by** (*cases a*) *auto*
qed

lemma *get-all-ann-decomposition-exists-prepend*[*dest*]:
assumes ‹(*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)›
shows ‹∃ *c*. *M* = *c* @ *b* @ *a*›
using *assms* **apply** (*induct M rule*: *ann-lit-list-induct*)
apply *simp*
by (*rename-tac L′ xs*, *case-tac* ‹*get-all-ann-decomposition xs*›;
auto dest!: *arg-cong*[*of* ‹*get-all-ann-decomposition -*› *- hd*]
get-all-ann-decomposition-decomp)+

lemma *get-all-ann-decomposition-incl*:
assumes ‹(*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)›
shows ‹*set b* ⊆ *set M*› **and** ‹*set a* ⊆ *set M*›
using *assms get-all-ann-decomposition-exists-prepend* **by** *fastforce*+

lemma *get-all-ann-decomposition-exists-prepend′*:
assumes ‹(*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)›
obtains *c* **where** ‹*M* = *c* @ *b* @ *a*›
using *assms* **apply** (*induct M rule*: *ann-lit-list-induct*)
apply *auto*[*1*]
by (*rename-tac L′ xs*, *case-tac* ‹*hd* (*get-all-ann-decomposition xs*)›,
auto dest!: *get-all-ann-decomposition-decomp simp add*: *list.set-sel*(*2*))+

lemma *union-in-get-all-ann-decomposition-is-subset*:
assumes ‹(*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)›
shows ‹*set a* ∪ *set b* ⊆ *set M*›
using *assms* **by** *force*

lemma *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons*:
‹∃ *c″*. (*Decided K* # *c*, *c″*) ∈ *set* (*get-all-ann-decomposition* (*c′* @ *Decided K* # *c*))›
apply (*induction c′ rule*: *ann-lit-list-induct*)
apply *auto*[*2*]
apply (*rename-tac L xs*,
case-tac ‹*hd* (*get-all-ann-decomposition* (*xs* @ *Decided K* # *c*))›)
apply (*case-tac* ‹*get-all-ann-decomposition* (*xs* @ *Decided K* # *c*)›)
by *auto*

lemma *fst-get-all-ann-decomposition-prepend-not-decided*:
assumes ‹∀ *m*∈*set MS*. ¬ *is-decided m*›
shows ‹*set* (*map fst* (*get-all-ann-decomposition M*))
= *set* (*map fst* (*get-all-ann-decomposition* (*MS* @ *M*)))›
using *assms* **apply** (*induction MS rule*: *ann-lit-list-induct*)
apply *auto*[*2*]
by (*rename-tac L m xs*; *case-tac* ‹*get-all-ann-decomposition* (*xs* @ *M*)›) *simp-all*

lemma *no-decision-get-all-ann-decomposition*:
‹∀ *l*∈*set M*. ¬ *is-decided l* ⟹ *get-all-ann-decomposition M* = [([], *M*)]›
by (*induction M rule*: *ann-lit-list-induct*) *auto*

Entailment of the Propagated by the Decided Literal

lemma *get-all-ann-decomposition-snd-union*:
‹*set M* = ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*)) ∪ {*L* |*L*. *is-decided L* ∧ *L* ∈ *set M*}›

```
  (is ‹?M M = ?U M ∪ ?Ls M›)
proof (induct M rule: ann-lit-list-induct)
  case Nil
  then show ?case by simp
next
  case (Decided L M) note IH = this(1)
  then have ‹Decided L ∈ ?Ls (Decided L # M)› by auto
  moreover have ‹?U (Decided L # M) = ?U M› by auto
  moreover have ‹?M M = ?U M ∪ ?Ls M› using IH by auto
  ultimately show ?case by auto
next
  case (Propagated L m M)
  then show ?case by (cases ‹(get-all-ann-decomposition M)›) auto
qed


definition all-decomposition-implies :: ‹'a clause set
  ⇒ (('a, 'm) ann-lits × ('a, 'm) ann-lits) list ⇒ bool› where
 ‹all-decomposition-implies N S ⟷ (∀ (Ls, seen) ∈ set S. unmark-l Ls ∪ N ⊨ps unmark-l seen)›

lemma all-decomposition-implies-empty[iff]:
  ‹all-decomposition-implies N []› unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-single[iff]:
  ‹all-decomposition-implies N [(Ls, seen)] ⟷ unmark-l Ls ∪ N ⊨ps unmark-l seen›
  unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-append[iff]:
  ‹all-decomposition-implies N (S @ S')
    ⟷ (all-decomposition-implies N S ∧ all-decomposition-implies N S')›
  unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-cons-pair[iff]:
  ‹all-decomposition-implies N ((Ls, seen) # S')
    ⟷ (all-decomposition-implies N [(Ls, seen)] ∧ all-decomposition-implies N S')›
  unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-cons-single[iff]:
  ‹all-decomposition-implies N (l # S') ⟷
    (unmark-l (fst l) ∪ N ⊨ps unmark-l (snd l) ∧
      all-decomposition-implies N S')›
  unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-trail-is-implied:
  assumes ‹all-decomposition-implies N (get-all-ann-decomposition M)›
  shows ‹N ∪ {unmark L |L. is-decided L ∧ L ∈ set M}
    ⊨ps unmark ‘ ⋃(set ‘ snd ‘ set (get-all-ann-decomposition M))›
using assms
proof (induct ‹length (get-all-ann-decomposition M)› arbitrary: M)
  case 0
  then show ?case by auto
next
  case (Suc n) note IH = this(1) and length = this(2) and decomp = this(3)
  consider
      (le1) ‹length (get-all-ann-decomposition M) ≤ 1›
```

```
  | (gt1) ‹length (get-all-ann-decomposition M) > 1›
  by arith
then show ?case
  proof cases
    case le1
    then obtain a b where g: ‹get-all-ann-decomposition M = (a, b) # []›
      by (cases ‹get-all-ann-decomposition M›) auto
    moreover {
      assume ‹a = []›
      then have ?thesis using Suc.prems g by auto
    }
    moreover {
      assume l: ‹length a = 1› and m: ‹is-decided (hd a)› and hd: ‹hd a ∈ set M›
      then have ‹unmark (hd a) ∈ {unmark L |L. is-decided L ∧ L ∈ set M}› by auto
      then have H: ‹unmark-l a ∪ N ⊆ N ∪ {unmark L |L. is-decided L ∧ L ∈ set M}›
        using l by (cases a) auto
      have f1: ‹unmark-l a ∪ N ⊨ps unmark-l b›
        using decomp unfolding all-decomposition-implies-def g by simp
      have ?thesis
        apply (rule true-clss-clss-subset) using f1 H g by auto
    }
    ultimately show ?thesis
      using get-all-ann-decomposition-length-1-fst-empty-or-length-1 by blast
  next
    case gt1
    then obtain Ls0 seen0 M′ where
      Ls0: ‹get-all-ann-decomposition M = (Ls0, seen0) # get-all-ann-decomposition M′› and
      length′: ‹length (get-all-ann-decomposition M′) = n› and
      M′-in-M: ‹set M′ ⊆ set M›
      using length by (induct M rule: ann-lit-list-induct) (auto simp: subset-insertI2)
    let ?d = ‹⋃(set ‘ snd ‘ set (get-all-ann-decomposition M′))›
    let ?unM = ‹{unmark L |L. is-decided L ∧ L ∈ set M}›
    let ?unM′ = ‹{unmark L |L. is-decided L ∧ L ∈ set M′}›
    {
      assume ‹n = 0›
      then have ‹get-all-ann-decomposition M′ = []› using length′ by auto
      then have ?thesis using Suc.prems unfolding all-decomposition-implies-def Ls0 by auto
    }
    moreover {
      assume n: ‹n > 0›
      then obtain Ls1 seen1 l where
        Ls1: ‹get-all-ann-decomposition M′ = (Ls1, seen1) # l›
        using length′ by (induct M′ rule: ann-lit-list-induct) auto

      have ‹all-decomposition-implies N (get-all-ann-decomposition M′)›
        using decomp unfolding Ls0 by auto
      then have N: ‹N ∪ ?unM′ ⊨ps unmark-s ?d›
        using IH length′ by auto
      have l: ‹N ∪ ?unM′ ⊆ N ∪ ?unM›
        using M′-in-M by auto
      from true-clss-clss-subset[OF this N]
      have ΨN: ‹N ∪ ?unM ⊨ps unmark-s ?d› by auto
      have ‹is-decided (hd Ls0)› and LS: ‹tl Ls0 = seen1 @ Ls1›
        using get-all-ann-decomposition-hd-hd[of M] unfolding Ls0 Ls1 by auto

      have LSM: ‹seen1 @ Ls1 = M′› using get-all-ann-decomposition-decomp[of M′] Ls1 by auto
```

have *M′*: ‹*set M′* = *?d* ∪ {*L* |*L*. *is-decided L* ∧ *L* ∈ *set M′*}›
  using *get-all-ann-decomposition-snd-union* **by** *auto*

{
  assume ‹*Ls0* ≠ []›
  then have ‹*hd Ls0* ∈ *set M*›
    using *get-all-ann-decomposition-fst-empty-or-hd-in-M Ls0* **by** *blast*
  then have ‹*N* ∪ *?unM* ⊨*p unmark* (*hd Ls0*)›
    using ‹*is-decided* (*hd Ls0*)› **by** (*metis* (*mono-tags*, *lifting*) *UnCI mem-Collect-eq true-clss-cls-in*)
} **note** *hd-Ls0* = *this*

have *l*: ‹*unmark* ‘ (*?d* ∪ {*L* |*L*. *is-decided L* ∧ *L* ∈ *set M′*}) = *unmark-s ?d* ∪ *?unM′*›
  by *auto*
have ‹*N* ∪ *?unM′* ⊨*ps unmark* ‘ (*?d* ∪ {*L* |*L*. *is-decided L* ∧ *L* ∈ *set M′*})›
  unfolding *l* **using** *N* **by** (*auto simp*: *all-in-true-clss-clss*)
then have *t*: ‹*N* ∪ *?unM′* ⊨*ps unmark-l* (*tl Ls0*)›
  using *M′* **unfolding** *LS LSM* **by** *auto*
then have ‹*N* ∪ *?unM* ⊨*ps unmark-l* (*tl Ls0*)›
  using *M′-in-M true-clss-clss-subset*[*OF - t*, *of* ‹*N* ∪ *?unM*›] **by** *auto*
then have ‹*N* ∪ *?unM* ⊨*ps unmark-l Ls0*›
  using *hd-Ls0* **by** (*cases Ls0*) *auto*

moreover have ‹*unmark-l Ls0* ∪ *N* ⊨*ps unmark-l seen0*›
  using *decomp* **unfolding** *Ls0* **by** *simp*
moreover have ‹⋀*M Ma*. (*M*::′*a clause set*) ∪ *Ma* ⊨*ps M*›
  by (*simp add*: *all-in-true-clss-clss*)
ultimately have Ψ: ‹*N* ∪ *?unM* ⊨*ps unmark-l seen0*›
  by (*meson true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r*)

moreover have ‹*unmark* ‘ (*set seen0* ∪ *?d*) = *unmark-l seen0* ∪ *unmark-s ?d*›
  by *auto*
ultimately have *?thesis* **using** Ψ*N* **unfolding** *Ls0* **by** *simp*
}
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *all-decomposition-implies-propagated-lits-are-implied*:
  assumes ‹*all-decomposition-implies N* (*get-all-ann-decomposition M*)›
  shows ‹*N* ∪ {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set M*} ⊨*ps unmark-l M*›
    (**is** ‹*?I* ⊨*ps ?A*›)
proof −
  have ‹*?I* ⊨*ps unmark-s* {*L* |*L*. *is-decided L* ∧ *L* ∈ *set M*}›
    by (*auto intro*: *all-in-true-clss-clss*)
  moreover have ‹*?I* ⊨*ps unmark* ‘ ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*))›
    using *all-decomposition-implies-trail-is-implied assms* **by** *blast*
  ultimately have ‹*N* ∪ {*unmark m* |*m*. *is-decided m* ∧ *m* ∈ *set M*}
    ⊨*ps unmark* ‘ ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*))
      ∪ *unmark* ‘ {*m* |*m*. *is-decided m* ∧ *m* ∈ *set M*}›
      by *blast*
  then show *?thesis*
    by (*metis* (*no-types*) *get-all-ann-decomposition-snd-union*[*of M*] *image-Un*)
qed

lemma *all-decomposition-implies-insert-single*:

‹*all-decomposition-implies N M* ⟹ *all-decomposition-implies* (*insert C N*) *M*›
unfolding *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-union*:
‹*all-decomposition-implies N M* ⟹ *all-decomposition-implies* (*N* ∪ *N'*) *M*›
unfolding *all-decomposition-implies-def sup.assoc*[*symmetric*] **by** (*auto intro*: *true-clss-clss-union-l*)

lemma *all-decomposition-implies-mono*:
‹*N* ⊆ *N'* ⟹ *all-decomposition-implies N M* ⟹ *all-decomposition-implies N' M*›
by (*metis all-decomposition-implies-union le-iff-sup*)

lemma *all-decomposition-implies-mono-right*:
‹*all-decomposition-implies I* (*get-all-ann-decomposition* (*M'* @ *M*)) ⟹
all-decomposition-implies I (*get-all-ann-decomposition M*)›
apply (*induction M' arbitrary*: *M rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal by *auto*
subgoal for *L C M' M*
by (*cases* ‹*get-all-ann-decomposition* (*M'* @ *M*)›) *auto*
done

1.2.4 Negation of a Clause

We define the negation of a *'a clause*: it converts a single clause to a set of clauses, where each clause is a single literal (whose negation is in the original clause).

definition *CNot* :: ‹*'v clause* ⇒ *'v clause-set*› **where**
‹*CNot* ψ = { {#−*L*#} | *L*. *L* ∈# ψ }›

lemma *finite-CNot*[*simp*]: ‹*finite* (*CNot C*)›
by (*auto simp*: *CNot-def*)

lemma *in-CNot-uminus*[*iff*]:
shows ‹{#*L*#} ∈ *CNot* ψ ⟷ −*L* ∈# ψ›
unfolding *CNot-def* **by** *force*

lemma
shows
CNot-add-mset[*simp*]: ‹*CNot* (*add-mset L* ψ) = *insert* {#−*L*#} (*CNot* ψ)› **and**
CNot-empty[*simp*]: ‹*CNot* {#} = {}› **and**
CNot-plus[*simp*]: ‹*CNot* (*A* + *B*) = *CNot A* ∪ *CNot B*›
unfolding *CNot-def* **by** *auto*

lemma *CNot-eq-empty*[*iff*]:
‹*CNot D* = {} ⟷ *D* = {#}›
unfolding *CNot-def* **by** (*auto simp add*: *multiset-eqI*)

lemma *in-CNot-implies-uminus*:
assumes ‹*L* ∈# *D*› **and** ‹*M* ⊨*as CNot D*›
shows ‹*M* ⊨*a* {#−*L*#}› **and** ‹−*L* ∈ *lits-of-l M*›
using *assms* **by** (*auto simp*: *true-annots-def true-annot-def CNot-def*)

lemma *CNot-remdups-mset*[*simp*]:
‹*CNot* (*remdups-mset A*) = *CNot A*›
unfolding *CNot-def* **by** *auto*

```
lemma Ball-CNot-Ball-mset[simp]:
 ‹(∀ x∈CNot D. P x) ⟷ (∀ L∈# D. P {#−L#})›
 unfolding CNot-def by auto

lemma consistent-CNot-not:
  assumes ‹consistent-interp I›
  shows ‹I ⊨s CNot φ ⟹ ¬I ⊨ φ›
  using assms unfolding consistent-interp-def true-clss-def true-cls-def by auto

lemma total-not-true-cls-true-clss-CNot:
  assumes ‹total-over-m I {φ}› and ‹¬I ⊨ φ›
  shows ‹I ⊨s CNot φ›
  using assms unfolding total-over-m-def total-over-set-def true-clss-def true-cls-def CNot-def
    apply clarify
  by (rename-tac x L, case-tac L) (force intro: pos-lit-in-atms-of neg-lit-in-atms-of)+

lemma total-not-CNot:
  assumes ‹total-over-m I {φ}› and ‹¬I ⊨s CNot φ›
  shows ‹I ⊨ φ›
  using assms total-not-true-cls-true-clss-CNot by auto

lemma atms-of-ms-CNot-atms-of[simp]:
  ‹atms-of-ms (CNot C) = atms-of C›
  unfolding atms-of-ms-def atms-of-def CNot-def by fastforce

lemma true-clss-clss-contradiction-true-clss-cls-false:
  ‹C ∈ D ⟹ D ⊨ps CNot C ⟹ D ⊨p {#}›
  unfolding true-clss-clss-def true-clss-cls-def total-over-m-def
  by (metis Un-commute atms-of-empty atms-of-ms-CNot-atms-of atms-of-ms-insert atms-of-ms-union
    consistent-CNot-not insert-absorb sup-bot.left-neutral true-clss-def)

lemma true-annots-CNot-all-atms-defined:
  assumes ‹M ⊨as CNot T› and a1: ‹L ∈# T›
  shows ‹atm-of L ∈ atm-of ‘ lits-of-l M›
  by (metis assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton)

lemma true-annots-CNot-all-uminus-atms-defined:
  assumes ‹M ⊨as CNot T› and a1: ‹−L ∈# T›
  shows ‹atm-of L ∈ atm-of ‘ lits-of-l M›
  by (metis assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton)

lemma true-clss-clss-false-left-right:
  assumes ‹{{#L#}} ∪ B ⊨p {#}›
  shows ‹B ⊨ps CNot {#L#}›
  unfolding true-clss-clss-def true-clss-cls-def
proof (intro allI impI)
  fix I
  assume
    tot: ‹total-over-m I (B ∪ CNot {#L#})› and
    cons: ‹consistent-interp I› and
    I: ‹I ⊨s B›
  have ‹total-over-m I ({{#L#}} ∪ B)› using tot by auto
  then have ‹¬I ⊨s insert {#L#} B›
    using assms cons unfolding true-clss-cls-def by simp
  then show ‹I ⊨s CNot {#L#}›
    using tot I by (cases L) auto
```

qed

lemma *true-annots-true-cls-def-iff-negation-in-model*:
‹*M* ⊨*as CNot C* ⟷ (∀ *L* ∈# *C*. −*L* ∈ *lits-of-l M*)›
unfolding *CNot-def true-annots-true-cls true-clss-def* **by** *auto*

lemma *true-clss-def-iff-negation-in-model*:
‹*M* ⊨*s CNot C* ⟷ (∀ *l* ∈# *C*. −*l* ∈ *M*)›
by (*auto simp*: *CNot-def true-clss-def*)

lemma *true-annots-CNot-definedD*:
‹*M* ⊨*as CNot C* ⟹ ∀ *L* ∈# *C*. *defined-lit M L*›
unfolding *true-annots-true-cls-def-iff-negation-in-model*
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)

lemma *true-annot-CNot-diff*:
‹*I* ⊨*as CNot C* ⟹ *I* ⊨*as CNot* (*C* − *C*′)›
by (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model dest*: *in-diffD*)

lemma *CNot-mset-replicate*[*simp*]:
‹*CNot* (*mset* (*replicate n L*)) = (*if n* = *0 then* {} *else* {{#−*L*#}})›
by (*induction n*) *auto*

lemma *consistent-CNot-not-tautology*:
‹*consistent-interp M* ⟹ *M* ⊨*s CNot D* ⟹ ¬*tautology D*›
by (*metis atms-of-ms-CNot-atms-of consistent-CNot-not satisfiable-carac*′ *satisfiable-def tautology-def total-over-m-def*)

lemma *atms-of-ms-CNot-atms-of-ms*: ‹*atms-of-ms* (*CNot CC*) = *atms-of-ms* {*CC*}›
by *simp*

lemma *total-over-m-CNot-toal-over-m*[*simp*]:
‹*total-over-m I* (*CNot C*) = *total-over-set I* (*atms-of C*)›
unfolding *total-over-m-def total-over-set-def* **by** *auto*

lemma *true-clss-cls-plus-CNot*:
assumes
CC-L: ‹*A* ⊨*p add-mset L CC*› **and**
CNot-CC: ‹*A* ⊨*ps CNot CC*›
shows ‹*A* ⊨*p* {#*L*#}›
unfolding *true-clss-clss-def true-clss-cls-def CNot-def total-over-m-def*
proof (*intro allI impI*)
fix *I*
assume
tot: ‹*total-over-set I* (*atms-of-ms* (*A* ∪ {{#*L*#}}))› **and**
cons: ‹*consistent-interp I*› **and**
I: ‹*I* ⊨*s A*›
let *?I* = ‹*I* ∪ {*Pos P*|*P*. *P* ∈ *atms-of CC* ∧ *P* ∉ *atm-of* ‘ *I*}›
have *cons*′: ‹*consistent-interp ?I*›
using *cons* **unfolding** *consistent-interp-def*
by (*auto simp*: *uminus-lit-swap atms-of-def rev-image-eqI*)
have *I*′: ‹*?I* ⊨*s A*›
using *I true-clss-union-increase* **by** *blast*
have *tot-CNot*: ‹*total-over-m ?I* (*A* ∪ *CNot CC*)›
using *tot atms-of-s-def* **by** (*fastforce simp*: *total-over-m-def total-over-set-def*)

```
  then have tot-I-A-CC-L: ‹total-over-m ?I (A ∪ {add-mset L CC})›
    using tot unfolding total-over-m-def total-over-set-atm-of by auto
  then have ‹?I ⊨ add-mset L CC› using CC-L cons' I' unfolding true-clss-cls-def by blast
  moreover
    have ‹?I ⊨s CNot CC› using CNot-CC cons' I' tot-CNot unfolding true-clss-clss-def by auto
    then have ‹¬A ⊨p CC›
      by (metis (no-types, lifting) I' atms-of-ms-CNot-atms-of-ms atms-of-ms-union cons'
        consistent-CNot-not tot-CNot total-over-m-def true-clss-cls-def)
    then have ‹¬?I ⊨ CC› using ‹?I ⊨s CNot CC› cons' consistent-CNot-not by blast
  ultimately have ‹?I ⊨ {#L#}› by blast
  then show ‹I ⊨ {#L#}›
    by (metis (no-types, lifting) atms-of-ms-union cons' consistent-CNot-not tot total-not-CNot
      total-over-m-def total-over-set-union true-clss-union-increase)
qed

lemma true-annots-CNot-lit-of-notin-skip:
  assumes LM: ‹L # M ⊨as CNot A› and LA: ‹lit-of L ∉# A› ‹−lit-of L ∉# A›
  shows ‹M ⊨as CNot A›
  using LM unfolding true-annots-def Ball-def
proof (intro allI impI)
  fix l
  assume H: ‹∀x. x ∈ CNot A ⟶ L # M ⊨a x › and l: ‹l ∈ CNot A›
  then have ‹L # M ⊨a l› by auto
  then show ‹M ⊨a l› using LA l by (cases L) (auto simp: CNot-def)
 qed

lemma true-clss-clss-union-false-true-clss-clss-cnot:
  ‹A ∪ {B} ⊨ps {{#}} ⟷ A ⊨ps CNot B›
  using total-not-CNot consistent-CNot-not unfolding total-over-m-def true-clss-clss-def
  by fastforce

lemma true-annot-remove-hd-if-notin-vars:
  assumes ‹a # M'⊨a D› and ‹atm-of (lit-of a) ∉ atms-of D›
  shows ‹M' ⊨a D›
  using assms true-cls-remove-hd-if-notin-vars unfolding true-annot-def by auto

lemma true-annot-remove-if-notin-vars:
  assumes ‹M @ M'⊨a D› and ‹∀x∈atms-of D. x ∉ atm-of ‘ lits-of-l M›
  shows ‹M' ⊨a D›
  using assms by (induct M) (auto dest: true-annot-remove-hd-if-notin-vars)

lemma true-annots-remove-if-notin-vars:
  assumes ‹M @ M'⊨as D› and ‹∀x∈atms-of-ms D. x ∉ atm-of ‘ lits-of-l M›
  shows ‹M' ⊨as D› unfolding true-annots-def
  using assms unfolding true-annots-def atms-of-ms-def
  by (force dest: true-annot-remove-if-notin-vars)

lemma all-variables-defined-not-imply-cnot:
  assumes
    ‹∀s ∈ atms-of-ms {B}. s ∈ atm-of ‘ lits-of-l A› and
    ‹¬ A ⊨a B›
  shows ‹A ⊨as CNot B›
  unfolding true-annot-def true-annots-def Ball-def CNot-def true-lit-def
proof (clarify, rule ccontr)
  fix L
```

assume *LB*: ‹*L* ∈# *B*› **and** *L-false*: ‹¬ *lits-of-l A* ⊨ {#}› **and** *L-A*: ‹− *L* ∉ *lits-of-l A*›
then have ‹*atm-of L* ∈ *atm-of* ‘ *lits-of-l A*›
using *assms*(*1*) **by** (*simp add*: *atm-of-lit-in-atms-of lits-of-def*)
then have ‹*L* ∈ *lits-of-l A* ∨ −*L* ∈ *lits-of-l A*›
using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* **by** *metis*
then have ‹*L* ∈ *lits-of-l A*› **using** *L-A* **by** *auto*
then show *False*
using *LB assms*(*2*) **unfolding** *true-annot-def true-lit-def true-cls-def Bex-def*
by *blast*
qed

lemma *CNot-union-mset*[*simp*]:
‹*CNot* (*A* ∪# *B*) = *CNot A* ∪ *CNot B*›
unfolding *CNot-def* **by** *auto*

lemma *true-clss-clss-true-clss-cls-true-clss-clss*:
assumes
‹*A* ⊨*ps unmark-l M*› **and** ‹*M* ⊨*as D*›
shows ‹*A* ⊨*ps D*›
by (*meson assms total-over-m-union true-annots-true-cls true-annots-true-clss-clss true-clss-clss-def true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r*)

lemma *true-clss-clss-CNot-true-clss-cls-unsatisfiable*:
assumes ‹*A* ⊨*ps CNot D*› **and** ‹*A* ⊨*p D*›
shows ‹*unsatisfiable A*›
using *assms*(*2*) **unfolding** *true-clss-clss-def true-clss-cls-def satisfiable-def*
by (*metis* (*full-types*) *Un-absorb Un-empty-right assms*(*1*) *atms-of-empty atms-of-ms-emtpy-set total-over-m-def total-over-m-insert total-over-m-union true-cls-empty true-clss-cls-def true-clss-clss-generalise-true-clss-clss true-clss-clss-true-clss-cls true-clss-clss-union-false-true-clss-clss-cnot*)

lemma *true-clss-cls-neg*:
‹*N* ⊨*p I* ⟷ *N* ∪ (λ*L*. {#−*L*#}) ‘ *set-mset I* ⊨*p* {#}›
proof −
have [*simp*]: ‹(λ*L*. {#− *L*#}) ‘ *set-mset I* = *CNot I*› **for** *I*
by (*auto simp*: *CNot-def*)
have [*iff*]: ‹ *total-over-m Ia* ((λ*L*. {#− *L*#}) ‘ *set-mset I*) ⟷
total-over-set Ia (*atms-of I*)› **for** *Ia*
by (*auto simp*: *total-over-m-def*
total-over-set-def atms-of-ms-def atms-of-def)
show *?thesis*
by (*auto simp*: *true-clss-cls-def consistent-CNot-not*
total-not-CNot)
qed

lemma *all-decomposition-implies-conflict-DECO-clause*:
assumes ‹*all-decomposition-implies N* (*get-all-ann-decomposition M*)› **and**
‹*M* ⊨*as CNot C*› **and**
‹*C* ∈ *N*›
shows ‹*N* ⊨*p* (*uminus o lit-of*) ‘# (*filter-mset is-decided* (*mset M*))›
(**is** ‹*?I* ⊨*p ?A*›)
proof −
have ‹{*unmark m* |*m*. *is-decided m* ∧ *m* ∈ *set M*} =
unmark-s {*L* ∈ *set M*. *is-decided L*}›
by *auto*

have ‹*N* ∪ *unmark-s* {*L* ∈ *set M*. *is-decided L*} ⊨*p* {#}›
  by (*metis* (*mono-tags*, *lifting*) *UnCI*
    ‹{*unmark m* |*m*. *is-decided m* ∧ *m* ∈ *set M*} = *unmark-s* {*L* ∈ *set M*. *is-decided L*}›
    all-decomposition-implies-propagated-lits-are-implied assms
    true-clss-clss-contradiction-true-clss-cls-false true-clss-clss-true-clss-cls-true-clss-clss)
then show *?thesis*
  apply (*subst true-clss-cls-neg*)
  by (*auto simp*: *image-image*)
qed

1.2.5 Other

definition ‹*no-dup L* ≡ *distinct* (*map* (λ*l*. *atm-of* (*lit-of l*)) *L*)›

lemma *no-dup-nil*[*simp*]:
  ‹*no-dup* []›
  by (*auto simp*: *no-dup-def*)

lemma *no-dup-cons*[*simp*]:
  ‹*no-dup* (*L* # *M*) ⟷ *undefined-lit M* (*lit-of L*) ∧ *no-dup M*›
  by (*auto simp*: *no-dup-def defined-lit-map*)

lemma *no-dup-append-cons*[*iff*]:
  ‹*no-dup* (*M* @ *L* # *M'*) ⟷ *undefined-lit* (*M* @ *M'*) (*lit-of L*) ∧ *no-dup* (*M* @ *M'*)›
  by (*auto simp*: *no-dup-def defined-lit-map*)

lemma *no-dup-append-append-cons*[*iff*]:
  ‹*no-dup* (*M0* @ *M* @ *L* # *M'*) ⟷ *undefined-lit* (*M0* @ *M* @ *M'*) (*lit-of L*) ∧ *no-dup* (*M0* @ *M* @ *M'*)›
  by (*auto simp*: *no-dup-def defined-lit-map*)

lemma *no-dup-rev*[*simp*]:
  ‹*no-dup* (*rev M*) ⟷ *no-dup M*›
  by (*auto simp*: *rev-map*[*symmetric*] *no-dup-def*)

lemma *no-dup-appendD*:
  ‹*no-dup* (*a* @ *b*) ⟹ *no-dup b*›
  by (*auto simp*: *no-dup-def*)

lemma *no-dup-appendD1*:
  ‹*no-dup* (*a* @ *b*) ⟹ *no-dup a*›
  by (*auto simp*: *no-dup-def*)

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
  assumes ‹*no-dup M*›
  shows ‹*length M* = *card* (*atm-of* ‘ *lits-of-l M*)›
  using *assms* **unfolding** *lits-of-def* **by** (*induct M*) (*auto simp add*: *image-image no-dup-def*)

lemma *distinct-consistent-interp*:
  ‹*no-dup M* ⟹ *consistent-interp* (*lits-of-l M*)›
proof (*induct M*)
  case *Nil*
  show *?case* **by** *auto*
next
  case (*Cons L M*)
  then have *a1*: ‹*consistent-interp* (*lits-of-l M*)› **by** *auto*

```
  have ‹undefined-lit M (lit-of L)›
     using Cons.prems by auto
  then show ?case
    using a1 by simp
qed

lemma same-mset-no-dup-iff:
  ‹mset M = mset M' ⟹ no-dup M ⟷ no-dup M'›
  by (auto simp: no-dup-def same-mset-distinct-iff)

lemma distinct-get-all-ann-decomposition-no-dup:
  assumes ‹(a, b) ∈ set (get-all-ann-decomposition M)›
  and ‹no-dup M›
  shows ‹no-dup (a @ b)›
  using assms by (force simp: no-dup-def)

lemma true-annots-lit-of-notin-skip:
  assumes ‹L # M ⊨as CNot A›
  and ‹−lit-of L ∉# A›
  and ‹no-dup (L # M)›
  shows ‹M ⊨as CNot A›
proof −
  have ‹∀ l ∈# A. −l ∈ lits-of-l (L # M)›
    using assms(1) in-CNot-implies-uminus(2) by blast
  moreover {
    have ‹undefined-lit M (lit-of L)›
      using assms(3) by force
    then have ‹− lit-of L ∉ lits-of-l M›
      by (simp add: Decided-Propagated-in-iff-in-lits-of-l) }
  ultimately have ‹∀ l ∈# A. −l ∈ lits-of-l M›
    using assms(2) by (metis insert-iff list.simps(15) lits-of-insert uminus-of-uminus-id)
  then show ?thesis by (auto simp add: true-annots-def)
qed

lemma no-dup-imp-distinct: ‹no-dup M ⟹ distinct M›
  by (induction M) (auto simp: defined-lit-map)

lemma no-dup-tlD: ‹no-dup a ⟹ no-dup (tl a)›
  unfolding no-dup-def by (cases a) auto

lemma defined-lit-no-dupD:
  ‹defined-lit M1 L ⟹ no-dup (M2 @ M1) ⟹ undefined-lit M2 L›
  ‹defined-lit M1 L ⟹ no-dup (M2' @ M2 @ M1) ⟹ undefined-lit M2' L›
  ‹defined-lit M1 L ⟹ no-dup (M2' @ M2 @ M1) ⟹ undefined-lit M2 L›
  by (auto simp: defined-lit-map no-dup-def)

lemma no-dup-consistentD:
  ‹no-dup M ⟹ L ∈ lits-of-l M ⟹ −L ∉ lits-of-l M›
  using consistent-interp-def distinct-consistent-interp by blast

lemma no-dup-not-tautology: ‹no-dup M ⟹ ¬tautology (image-mset lit-of (mset M))›
  by (induction M) (auto simp: tautology-add-mset uminus-lit-swap defined-lit-def
     dest: atm-imp-decided-or-proped)

lemma no-dup-distinct: ‹no-dup M ⟹ distinct-mset (image-mset lit-of (mset M))›
  by (induction M) (auto simp: uminus-lit-swap defined-lit-def
```

dest: *atm-imp-decided-or-proped*)

lemma *no-dup-not-tautology-uminus*: ‹*no-dup M* ⟹ ¬*tautology* {#−*lit-of L*. *L* ∈# *mset M*#}›
by (*induction M*) (*auto simp*: *tautology-add-mset uminus-lit-swap defined-lit-def*
dest: *atm-imp-decided-or-proped*)

lemma *no-dup-distinct-uminus*: ‹*no-dup M* ⟹ *distinct-mset* {#−*lit-of L*. *L* ∈# *mset M*#}›
by (*induction M*) (*auto simp*: *uminus-lit-swap defined-lit-def*
dest: *atm-imp-decided-or-proped*)

lemma *no-dup-map-lit-of*: ‹*no-dup M* ⟹ *distinct* (*map lit-of M*)›
apply (*induction M*)
apply (*auto simp*: *dest*: *no-dup-imp-distinct*)
by (*meson distinct.simps*(*2*) *no-dup-cons no-dup-imp-distinct*)

lemma *no-dup-alt-def*:
‹*no-dup M* ⟷ *distinct-mset* {#*atm-of* (*lit-of x*). *x* ∈# *mset M*#}›
by (*auto simp*: *no-dup-def simp flip*: *distinct-mset-mset-distinct*)

lemma *no-dup-append-in-atm-notin*:
assumes ‹*no-dup* (*M* @ *M'*)› **and** ‹*L* ∈ *lits-of-l M'*›
shows ‹*undefined-lit M L*›
using *assms* **by** (*auto simp add*: *atm-lit-of-set-lits-of-l no-dup-def*
defined-lit-map)

lemma *no-dup-uminus-append-in-atm-notin*:
assumes ‹*no-dup* (*M* @ *M'*)› **and** ‹−*L* ∈ *lits-of-l M'*›
shows ‹*undefined-lit M L*›
using *Decided-Propagated-in-iff-in-lits-of-l assms defined-lit-no-dupD*(*1*) **by** *blast*

1.2.6 Extending Entailments to multisets

We have defined previous entailment with respect to sets, but we also need a multiset version depending on the context. The conversion is simple using the function *set-mset* (in this direction, there is no loss of information).

abbreviation *true-annots-mset* (**infix** ⊨*asm 50*) **where**
‹*I* ⊨*asm C* ≡ *I* ⊨*as* (*set-mset C*)›

abbreviation *true-clss-clss-m* :: ‹*'v clause multiset* ⇒ *'v clause multiset* ⇒ *bool*› (**infix** ⊨*psm 50*)
where
‹*I* ⊨*psm C* ≡ *set-mset I* ⊨*ps* (*set-mset C*)›

Analog of theorem *true-clss-clss-subsetE*

lemma *true-clss-clssm-subsetE*: ‹*N* ⊨*psm B* ⟹ *A* ⊆# *B* ⟹ *N* ⊨*psm A*›
using *set-mset-mono true-clss-clss-subsetE* **by** *blast*

abbreviation *true-clss-cls-m*:: ‹*'a clause multiset* ⇒ *'a clause* ⇒ *bool*› (**infix** ⊨*pm 50*) **where**
‹*I* ⊨*pm C* ≡ *set-mset I* ⊨*p C*›

abbreviation *distinct-mset-mset* :: ‹*'a multiset multiset* ⇒ *bool*› **where**
‹*distinct-mset-mset* Σ ≡ *distinct-mset-set* (*set-mset* Σ)›

abbreviation *all-decomposition-implies-m* **where**
‹*all-decomposition-implies-m A B* ≡ *all-decomposition-implies* (*set-mset A*) *B*›

abbreviation *atms-of-mm* :: ‹*'a clause multiset* ⇒ *'a set*› **where**
‹*atms-of-mm U* ≡ *atms-of-ms* (*set-mset U*)›

Other definition using *Union-mset*

lemma *atms-of-mm-alt-def*: ‹*atms-of-mm U* = *set-mset* ($\sum_{\#}$ (*image-mset* (*image-mset atm-of*) *U*))›
  unfolding *atms-of-ms-def* **by** (*auto simp*: *atms-of-def*)

abbreviation *true-clss-m*:: ‹*'a partial-interp* ⇒ *'a clause multiset* ⇒ *bool*› (**infix** ⊨*sm 50*) **where**
‹*I* ⊨*sm C* ≡ *I* ⊨*s set-mset C*›

abbreviation *true-clss-ext-m* (**infix** ⊨*sextm 49*) **where**
‹*I* ⊨*sextm C* ≡ *I* ⊨*sext set-mset C*›

lemma *true-clss-cls-cong-set-mset*:
  ‹*N* ⊨*pm D* ⟹ *set-mset D* = *set-mset D'* ⟹ *N* ⊨*pm D'*›
  by (*auto simp add*: *true-clss-cls-def true-cls-def atms-of-cong-set-mset*[*of D D'*])

1.2.7 More Lemmas

lemma *no-dup-cannot-not-lit-and-uminus*:
  ‹*no-dup M* ⟹ − *lit-of xa* = *lit-of x* ⟹ *x* ∈ *set M* ⟹ *xa* ∉ *set M*›
  by (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id' no-dup-def*)

lemma *atms-of-ms-single-atm-of*[*simp*]:
  ‹*atms-of-ms* {*unmark L* |*L*. *P L*} = *atm-of* ‘ {*lit-of L* |*L*. *P L*}›
  unfolding *atms-of-ms-def* **by** *force*

lemma *true-cls-mset-restrict*:
  ‹{*L* ∈ *I*. *atm-of L* ∈ *atms-of-mm N*} ⊨*m N* ⟷ *I* ⊨*m N*›
  by (*auto simp*: *true-cls-mset-def true-cls-def*
    dest!: *multi-member-split*)

lemma *true-clss-restrict*:
  ‹{*L* ∈ *I*. *atm-of L* ∈ *atms-of-mm N*} ⊨*sm N* ⟷ *I* ⊨*sm N*›
  by (*auto simp*: *true-clss-def true-cls-def*
    dest!: *multi-member-split*)

lemma *total-over-m-atms-incl*:
  assumes ‹*total-over-m M* (*set-mset N*)›
  shows
    ‹*x* ∈ *atms-of-mm N* ⟹ *x* ∈ *atms-of-s M*›
  by (*meson assms contra-subsetD total-over-m-alt-def*)

lemma *true-clss-restrict-iff*:
  assumes ‹¬*tautology χ*›
  shows ‹*N* ⊨*p χ* ⟷ *N* ⊨*p* {#*L* ∈# *χ*. *atm-of L* ∈ *atms-of-ms N*#}› (**is** ‹*?A* ⟷ *?B*›)
  apply (*subst true-clss-alt-def2*[*OF assms*])
  apply (*subst true-clss-alt-def2*)
  subgoal using *not-tautology-mono*[*OF - assms*] **by** (*auto dest*: *not-tautology-minus*)
  apply (*rule HOL.iff-allI*)
  apply (*auto 5 5 simp*: *true-cls-def atms-of-s-def dest*!: *multi-member-split*)
  done

1.2.8 Negation of annotated clauses

definition *negate-ann-lits* :: ‹(*'v literal*, *'v literal*, *'mark*) *annotated-lits* ⇒ *'v literal multiset*› **where**

‹*negate-ann-lits M* = (λ*L*. − *lit-of L*) ‘# *mset M*›

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

lemma *entails-CNot-negate-ann-lits*:
‹*M* ⊨*as CNot D* ⟷ *set-mset D* ⊆ *set-mset* (*negate-ann-lits M*)›
by (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model*
negate-ann-lits-def lits-of-def uminus-lit-swap
dest!: *multi-member-split*)

Pointwise negation of a clause:

definition *pNeg* :: ‹′*v clause* ⇒ ′*v clause*› **where**
‹*pNeg C* = {#−*D*. *D* ∈# *C*#}›

lemma *pNeg-simps*:
‹*pNeg* (*add-mset A C*) = *add-mset* (−*A*) (*pNeg C*)›
‹*pNeg* (*C* + *D*) = *pNeg C* + *pNeg D*›
by (*auto simp*: *pNeg-def*)

lemma *atms-of-pNeg*[*simp*]: ‹*atms-of* (*pNeg C*) = *atms-of C*›
by (*auto simp*: *pNeg-def atms-of-def image-image*)

lemma *negate-ann-lits-pNeg-lit-of*: ‹*negate-ann-lits* = *pNeg o image-mset lit-of o mset*›
by (*intro ext*) (*auto simp*: *negate-ann-lits-def pNeg-def*)

lemma *negate-ann-lits-empty-iff*: ‹*negate-ann-lits M* ≠ {#} ⟷ *M* ≠ []›
by (*auto simp*: *negate-ann-lits-def*)

lemma *atms-of-negate-ann-lits*[*simp*]: ‹*atms-of* (*negate-ann-lits M*) = *atm-of* ‘ (*lits-of-l M*)›
unfolding *negate-ann-lits-def lits-of-def atms-of-def* **by** (*auto simp*: *image-image*)

lemma *tautology-pNeg*[*simp*]:
‹*tautology* (*pNeg C*) ⟷ *tautology C*›
by (*auto 5 5 simp*: *tautology-decomp pNeg-def*
uminus-lit-swap add-mset-eq-add-mset eq-commute[*of* ‹*Neg* -› ‹− -›] *eq-commute*[*of* ‹*Pos* -› ‹− -›]
dest!: *multi-member-split*)

lemma *pNeg-convolution*[*simp*]:
‹*pNeg* (*pNeg C*) = *C*›
by (*auto simp*: *pNeg-def*)

lemma *pNeg-minus*[*simp*]: ‹*pNeg* (*A* − *B*) = *pNeg A* − *pNeg B*›
unfolding *pNeg-def*
by (*subst image-mset-minus-inj-on*) (*auto simp*: *inj-on-def*)

lemma *pNeg-empty*[*simp*]: ‹*pNeg* {#} = {#}›
unfolding *pNeg-def*
by (*auto simp*: *inj-on-def*)

lemma *pNeg-replicate-mset*[*simp*]: ‹*pNeg* (*replicate-mset n L*) = *replicate-mset n* (−*L*)›
unfolding *pNeg-def* **by** *auto*

lemma *distinct-mset-pNeg-iff*[*iff*]: ‹*distinct-mset* (*pNeg x*) ⟷ *distinct-mset x*›
unfolding *pNeg-def*
by (*rule distinct-image-mset-inj*) (*auto simp*: *inj-on-def*)

lemma *pNeg-simple-clss-iff* [*simp*]:
‹*pNeg M* ∈ *simple-clss N* ⟷ *M* ∈ *simple-clss N*›
by (*auto simp*: *simple-clss-def*)

lemma *atms-of-ms-pNeg*[*simp*]: ‹*atms-of-ms* (*pNeg* ‘ *N*) = *atms-of-ms N*›
unfolding *atms-of-ms-def pNeg-def* **by** (*auto simp*: *image-image atms-of-def*)

definition *DECO-clause* :: ‹(′*v*, ′*a*) *ann-lits* ⇒ ′*v clause*› **where**
‹*DECO-clause M* = (*uminus o lit-of*) ‘# (*filter-mset is-decided* (*mset M*))›

lemma
DECO-clause-cons-Decide[*simp*]:
‹*DECO-clause* (*Decided L* # *M*) = *add-mset* (−*L*) (*DECO-clause M*)› **and**
DECO-clause-cons-Proped[*simp*]:
‹*DECO-clause* (*Propagated L C* # *M*) = *DECO-clause M*›
by (*auto simp*: *DECO-clause-def*)

lemma *no-dup-distinct-mset*[*intro*!]:
assumes *n-d*: ‹*no-dup M*›
shows ‹*distinct-mset* (*negate-ann-lits M*)›
unfolding *negate-ann-lits-def no-dup-def*
proof (*subst distinct-image-mset-inj*)
show ‹*inj-on* (λ*L*. − *lit-of L*) (*set-mset* (*mset M*))›
unfolding *inj-on-def Ball-def*
proof (*intro allI impI*, *rule ccontr*)
fix *L L*′
assume
L: ‹*L* ∈# *mset M*› **and**
L′: ‹*L*′ ∈# *mset M*› **and**
lit: ‹− *lit-of L* = − *lit-of L*′› **and**
LL′: ‹*L* ≠ *L*′›
have ‹*atm-of* (*lit-of L*) = *atm-of* (*lit-of L*′)›
using *lit* **by** *auto*
moreover have ‹*atm-of* (*lit-of L*) ∈# (λ*l*. *atm-of* (*lit-of l*)) ‘# *mset M*›
using *L* **by** *auto*
moreover have ‹*atm-of* (*lit-of L*′) ∈# (λ*l*. *atm-of* (*lit-of l*)) ‘# *mset M*›
using *L*′ **by** *auto*
ultimately show *False*
using *assms LL*′ *L L*′ **unfolding** *distinct-mset-mset-distinct*[*symmetric*] *mset-map no-dup-def*
apply − **apply** (*rule distinct-image-mset-not-equal*[*of L L*′ ‹(λ*l*. *atm-of* (*lit-of l*))›])
by *auto*
qed
next
show ‹*distinct-mset* (*mset M*)›
using *no-dup-imp-distinct*[*OF n-d*] **by** *simp*
qed

lemma *in-negate-trial-iff*: ‹*L* ∈# *negate-ann-lits M* ⟷ − *L* ∈ *lits-of-l M*›
unfolding *negate-ann-lits-def lits-of-def* **by** (*auto simp*: *uminus-lit-swap*)

lemma *negate-ann-lits-cons*[*simp*]:
‹*negate-ann-lits* (*L* # *M*) = *add-mset* (− *lit-of L*) (*negate-ann-lits M*)›
by (*auto simp*: *negate-ann-lits-def*)

lemma *uminus-simple-clss-iff* [*simp*]:
‹*uminus* ‘# *M* ∈ *simple-clss N* ⟷ *M* ∈ *simple-clss N*›
by (*metis pNeg-simple-clss-iff pNeg-def*)

lemma *pNeg-mono*: ‹*C* ⊆# *C'* ⟹ *pNeg C* ⊆# *pNeg C'*›
by (*auto simp*: *image-mset-subseteq-mono pNeg-def*)

end
theory *Partial-And-Total-Herbrand-Interpretation*
imports *Partial-Herbrand-Interpretation*
Ordered-Resolution-Prover.Herbrand-Interpretation
begin

1.3 Bridging of total and partial Herbrand interpretation

This theory has mostly be written as a sanity check between the two entailment notion.

definition *partial-model-of* :: ‹*'a interp* ⇒ *'a partial-interp*› **where**
‹*partial-model-of I* = *Pos* ‘ *I* ∪ *Neg* ‘ {*x*. *x* ∉ *I*}›

definition *total-model-of* :: ‹*'a partial-interp* ⇒ *'a interp*› **where**
‹*total-model-of I* = {*x*. *Pos x* ∈ *I*}›

lemma *total-over-set-partial-model-of*:
‹*total-over-set* (*partial-model-of I*) *UNIV*›
unfolding *total-over-set-def*
by (*auto simp*: *partial-model-of-def*)

lemma *consistent-interp-partial-model-of*:
‹*consistent-interp* (*partial-model-of I*)›
unfolding *consistent-interp-def*
by (*auto simp*: *partial-model-of-def*)

lemma *consistent-interp-alt-def*:
‹*consistent-interp I* ⟷ (∀ *L*. ¬(*Pos L* ∈ *I* ∧ *Neg L* ∈ *I*))›
unfolding *consistent-interp-def*
by (*metis literal.exhaust uminus-Neg uminus-of-uminus-id*)

context
fixes *I* :: ‹*'a partial-interp*›
assumes *cons*: ‹*consistent-interp I*›
begin

lemma *partial-implies-total-true-cls-total-model-of*:
assumes ‹*Partial-Herbrand-Interpretation.true-cls I C*›
shows ‹*Herbrand-Interpretation.true-cls* (*total-model-of I*) *C*›
using *assms cons* **apply** −
unfolding *total-model-of-def Partial-Herbrand-Interpretation.true-cls-def*
Herbrand-Interpretation.true-cls-def consistent-interp-alt-def
by (*rule bexE*, *assumption*)
(*case-tac x*; *auto*)

lemma *total-implies-partial-true-cls-total-model-of*:
assumes ‹*Herbrand-Interpretation.true-cls* (*total-model-of I*) *C*› **and**

‹*total-over-set I* (*atms-of C*)›
shows ‹*Partial-Herbrand-Interpretation.true-cls I C*›
using *assms cons*
unfolding *total-model-of-def Partial-Herbrand-Interpretation.true-cls-def*
Herbrand-Interpretation.true-cls-def consistent-interp-alt-def
total-over-m-def total-over-set-def
by (*auto simp*: *atms-of-def dest*: *multi-member-split*)

lemma *partial-implies-total-true-clss-total-model-of*:
assumes ‹*Partial-Herbrand-Interpretation.true-clss I C*›
shows ‹*Herbrand-Interpretation.true-clss* (*total-model-of I*) *C*›
using *assms cons*
unfolding *Partial-Herbrand-Interpretation.true-clss-def*
Herbrand-Interpretation.true-clss-def
by (*auto simp*: *partial-implies-total-true-cls-total-model-of*)

lemma *total-implies-partial-true-clss-total-model-of*:
assumes ‹*Herbrand-Interpretation.true-clss* (*total-model-of I*) *C*› **and**
‹*total-over-m I C*›
shows ‹*Partial-Herbrand-Interpretation.true-clss I C*›
using *assms cons mk-disjoint-insert*
unfolding *Partial-Herbrand-Interpretation.true-clss-def*
Herbrand-Interpretation.true-clss-def
total-over-set-def
by (*fastforce simp*: *total-implies-partial-true-cls-total-model-of*
dest: *multi-member-split*)

end

lemma *total-implies-partial-true-cls-partial-model-of*:
assumes ‹*Herbrand-Interpretation.true-cls I C*›
shows ‹*Partial-Herbrand-Interpretation.true-cls* (*partial-model-of I*) *C*›
using *assms* **apply** −
unfolding *partial-model-of-def Partial-Herbrand-Interpretation.true-cls-def*
Herbrand-Interpretation.true-cls-def consistent-interp-alt-def
by (*rule bexE*, *assumption*)
(*case-tac x*; *auto*)

lemma *total-implies-partial-true-clss-partial-model-of*:
assumes ‹*Herbrand-Interpretation.true-clss I C*›
shows ‹*Partial-Herbrand-Interpretation.true-clss* (*partial-model-of I*) *C*›
using *assms*
unfolding *Partial-Herbrand-Interpretation.true-clss-def*
Herbrand-Interpretation.true-clss-def
by (*auto simp*: *total-implies-partial-true-cls-partial-model-of*)

lemma *partial-total-satisfiable-iff*:
‹*Partial-Herbrand-Interpretation.satisfiable N* $\longleftrightarrow$ *Herbrand-Interpretation.satisfiable N*›
by (*meson consistent-interp-partial-model-of partial-implies-total-true-clss-total-model-of*
satisfiable-carac total-implies-partial-true-clss-partial-model-of)

end
theory *Prop-Logic*
imports *Main*

begin

Chapter 2

Normalisation

We define here the normalisation from formula towards conjunctive and disjunctive normal form, including normalisation towards multiset of multisets to represent CNF.

2.1 Logics

In this section we define the syntax of the formula and an abstraction over it to have simpler proofs. After that we define some properties like subformula and rewriting.

2.1.1 Definition and Abstraction

The propositional logic is defined inductively. The type parameter is the type of the variables.

datatype *'v propo* =
  FT | *FF* | *FVar 'v* | *FNot 'v propo* | *FAnd 'v propo 'v propo* | *FOr 'v propo 'v propo*
  | *FImp 'v propo 'v propo* | *FEq 'v propo 'v propo*

We do not define any notation for the formula, to distinguish properly between the formulas and Isabelle's logic.

To ease the proofs, we will write the the formula on a homogeneous manner, namely a connecting argument and a list of arguments.

datatype *'v connective* = *CT* | *CF* | *CVar 'v* | *CNot* | *CAnd* | *COr* | *CImp* | *CEq*

abbreviation *nullary-connective* ≡ {*CF*} ∪ {*CT*} ∪ {*CVar x* | *x. True*}
definition *binary-connectives* ≡ {*CAnd*, *COr*, *CImp*, *CEq*}

We define our own induction principal: instead of distinguishing every constructor, we group them by arity.

lemma *propo-induct-arity*[*case-names nullary unary binary*]:
  fixes φ ψ :: *'v propo*
  assumes *nullary*: $\bigwedge\varphi\ x.\ \varphi = FF \vee \varphi = FT \vee \varphi = FVar\ x \Longrightarrow P\ \varphi$
  and *unary*: $\bigwedge\psi.\ P\ \psi \Longrightarrow P\ (FNot\ \psi)$
  and *binary*: $\bigwedge\varphi\ \psi 1\ \psi 2.\ P\ \psi 1 \Longrightarrow P\ \psi 2 \Longrightarrow \varphi = FAnd\ \psi 1\ \psi 2 \vee \varphi = FOr\ \psi 1\ \psi 2 \vee \varphi = FImp\ \psi 1$
$\psi 2$
    $\vee\ \varphi = FEq\ \psi 1\ \psi 2 \Longrightarrow P\ \varphi$
  shows $P\ \psi$
  apply (*induct rule*: *propo.induct*)
  using *assms* **by** *metis+*

The function *conn* is the interpretation of our representation (connective and list of arguments). We define any thing that has no sense to be false

fun *conn* :: *'v connective* ⇒ *'v propo list* ⇒ *'v propo* **where**
conn CT [] = *FT* |
conn CF [] = *FF* |
conn (*CVar v*) [] = *FVar v* |
conn CNot [φ] = *FNot* φ |
conn CAnd (φ # [ψ]) = *FAnd* φ ψ |
conn COr (φ # [ψ]) = *FOr* φ ψ |
conn CImp (φ # [ψ]) = *FImp* φ ψ |
conn CEq (φ # [ψ]) = *FEq* φ ψ |
conn - - = *FF*

We will often use case distinction, based on the arity of the *'v connective*, thus we define our own splitting principle.

lemma *connective-cases-arity*[*case-names nullary binary unary*]:
  assumes *nullary*: $\bigwedge x.\ c = CT \vee c = CF \vee c = CVar\ x \Longrightarrow P$
  and *binary*: $c \in$ *binary-connectives* $\Longrightarrow P$
  and *unary*: $c = CNot \Longrightarrow P$
  shows *P*
  using *assms* **by** (*cases c*) (*auto simp*: *binary-connectives-def*)

lemma *connective-cases-arity-2*[*case-names nullary unary binary*]:
  assumes *nullary*: $c \in$ *nullary-connective* $\Longrightarrow P$
  and *unary*: $c = CNot \Longrightarrow P$
  and *binary*: $c \in$ *binary-connectives* $\Longrightarrow P$
  shows *P*
  using *assms* **by** (*cases c*, *auto simp add*: *binary-connectives-def*)

Our previous definition is not necessary correct (connective and list of arguments), so we define an inductive predicate.

inductive *wf-conn* :: *'v connective* ⇒ *'v propo list* ⇒ *bool* **for** *c* :: *'v connective* **where**
wf-conn-nullary[*simp*]: ($c = CT \vee c = CF \vee c = CVar\ v$) $\Longrightarrow$ *wf-conn c* [] |
wf-conn-unary[*simp*]: $c = CNot \Longrightarrow$ *wf-conn c* [ψ] |
wf-conn-binary[*simp*]: $c \in$ *binary-connectives* $\Longrightarrow$ *wf-conn c* (ψ # ψ' # [])
thm *wf-conn.induct*
lemma *wf-conn-induct*[*consumes 1*, *case-names CT CF CVar CNot COr CAnd CImp CEq*]:
  assumes *wf-conn c x* **and**
    $\bigwedge v.\ c = CT \Longrightarrow P$ [] **and**
    $\bigwedge v.\ c = CF \Longrightarrow P$ [] **and**
    $\bigwedge v.\ c = CVar\ v \Longrightarrow P$ [] **and**
    $\bigwedge \psi.\ c = CNot \Longrightarrow P\ [\psi]$ **and**
    $\bigwedge \psi\ \psi'.\ c = COr \Longrightarrow P\ [\psi, \psi']$ **and**
    $\bigwedge \psi\ \psi'.\ c = CAnd \Longrightarrow P\ [\psi, \psi']$ **and**
    $\bigwedge \psi\ \psi'.\ c = CImp \Longrightarrow P\ [\psi, \psi']$ **and**
    $\bigwedge \psi\ \psi'.\ c = CEq \Longrightarrow P\ [\psi, \psi']$
  shows *P x*
  using *assms* **by** *induction* (*auto simp*: *binary-connectives-def*)

2.1.2 Properties of the Abstraction

First we can define simplification rules.

lemma *wf-conn-conn*[*simp*]:

wf-conn CT l ⟹ *conn CT l* = *FT*
wf-conn CF l ⟹ *conn CF l* = *FF*
wf-conn (*CVar x*) *l* ⟹ *conn* (*CVar x*) *l* = *FVar x*
apply (*simp-all add*: *wf-conn.simps*)
unfolding *binary-connectives-def* **by** *simp-all*

lemma *wf-conn-list-decomp*[*simp*]:
wf-conn CT l ⟷ *l* = []
wf-conn CF l ⟷ *l* = []
wf-conn (*CVar x*) *l* ⟷ *l* = []
wf-conn CNot (ξ @ φ # ξ') ⟷ ξ = [] ∧ ξ' = []
apply (*simp-all add*: *wf-conn.simps*)
unfolding *binary-connectives-def* **apply** *simp-all*
by (*metis append-Nil append-is-Nil-conv list.distinct*(*1*) *list.sel*(*3*) *tl-append2*)

lemma *wf-conn-list*:
wf-conn c l ⟹ *conn c l* = *FT* ⟷ (*c* = *CT* ∧ *l* = [])
wf-conn c l ⟹ *conn c l* = *FF* ⟷ (*c* = *CF* ∧ *l* = [])
wf-conn c l ⟹ *conn c l* = *FVar x* ⟷ (*c* = *CVar x* ∧ *l* = [])
wf-conn c l ⟹ *conn c l* = *FAnd a b* ⟷ (*c* = *CAnd* ∧ *l* = *a* # *b* # [])
wf-conn c l ⟹ *conn c l* = *FOr a b* ⟷ (*c* = *COr* ∧ *l* = *a* # *b* # [])
wf-conn c l ⟹ *conn c l* = *FEq a b* ⟷ (*c* = *CEq* ∧ *l* = *a* # *b* # [])
wf-conn c l ⟹ *conn c l* = *FImp a b* ⟷ (*c* = *CImp* ∧ *l* = *a* # *b* # [])
wf-conn c l ⟹ *conn c l* = *FNot a* ⟷ (*c* = *CNot* ∧ *l* = *a* # [])
apply (*induct l rule*: *wf-conn.induct*)
unfolding *binary-connectives-def* **by** *auto*

In the binary connective cases, we will often decompose the list of arguments (of length 2) into two elements.

lemma *list-length2-decomp*: *length l* = *2* ⟹ (∃ *a b*. *l* = *a* # *b* # [])
apply (*induct l*, *auto*)
by (*rename-tac l*, *case-tac l*, *auto*)

wf-conn for binary operators means that there are two arguments.

lemma *wf-conn-bin-list-length*:
fixes *l* :: *'v propo list*
assumes *conn*: *c* ∈ *binary-connectives*
shows *length l* = *2* ⟷ *wf-conn c l*
proof
assume *length l* = *2*
then show *wf-conn c l* **using** *wf-conn-binary list-length2-decomp* **using** *conn* **by** *metis*
next
assume *wf-conn c l*
then show *length l* = *2* (**is** *?P l*)
proof (*cases rule*: *wf-conn.induct*)
case *wf-conn-nullary*
then show *?P* [] **using** *conn binary-connectives-def*
using *connective.distinct*(*11*) *connective.distinct*(*13*) *connective.distinct*(*9*) **by** *blast*
next
fix ψ :: *'v propo*
case *wf-conn-unary*
then show *?P* [ψ] **using** *conn binary-connectives-def*
using *connective.distinct* **by** *blast*

next
fix ψ ψ':: *'v propo*
show *?P* $[\psi, \psi']$ **by** *auto*
qed
qed

lemma *wf-conn-not-list-length*[*iff*]:
fixes *l* :: *'v propo list*
shows *wf-conn CNot l* $\longleftrightarrow$ *length l* = *1*
apply *auto*
apply (*metis append-Nil connective.distinct*(*5*,*17*,*27*) *length-Cons list.size*(*3*) *wf-conn.simps wf-conn-list-decomp*(*4*))
by (*simp add*: *length-Suc-conv wf-conn.simps*)

Decomposing the Not into an element is moreover very useful.

lemma *wf-conn-Not-decomp*:
fixes *l* :: *'v propo list* **and** *a* :: *'v*
assumes *corr*: *wf-conn CNot l*
shows $\exists$ *a*. *l* = [*a*]
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-length-conv corr length-0-conv wf-conn-not-list-length*)

The *wf-conn* remains correct if the length of list does not change. This lemma is very useful when we do one rewriting step

lemma *wf-conn-no-arity-change*:
length l = *length l'* $\Longrightarrow$ *wf-conn c l* $\longleftrightarrow$ *wf-conn c l'*
proof −
{
fix *l l'*
have *length l* = *length l'* $\Longrightarrow$ *wf-conn c l* $\Longrightarrow$ *wf-conn c l'*
apply (*cases c l rule*: *wf-conn.induct*, *auto*)
by (*metis wf-conn-bin-list-length*)
}
then show *length l* = *length l'* $\Longrightarrow$ *wf-conn c l* = *wf-conn c l'* **by** *metis*
qed

lemma *wf-conn-no-arity-change-helper*:
length (ξ @ φ # ξ') = *length* (ξ @ φ' # ξ')
by *auto*

The injectivity of *conn* is useful to prove equality of the connectives and the lists.

lemma *conn-inj-not*:
assumes *correct*: *wf-conn c l*
and *conn*: *conn c l* = *FNot* ψ
shows *c* = *CNot* **and** *l* = $[\psi]$
apply (*cases c l rule*: *wf-conn.cases*)
using *correct conn* **unfolding** *binary-connectives-def* **apply** *auto*
apply (*cases c l rule*: *wf-conn.cases*)
using *correct conn* **unfolding** *binary-connectives-def* **by** *auto*

lemma *conn-inj*:
fixes *c ca* :: *'v connective* **and** *l* ψs :: *'v propo list*
assumes *corr*: *wf-conn ca l*
and *corr'*: *wf-conn c* ψs

and *eq*: *conn ca l* = *conn c ψs*
shows *ca* = *c* ∧ *ψs* = *l*
using *corr*
proof (*cases ca l rule*: *wf-conn.cases*)
case (*wf-conn-nullary v*)
then show *ca* = *c* ∧ *ψs* = *l* **using** *assms*
by (*metis conn.simps*(*1*) *conn.simps*(*2*) *conn.simps*(*3*) *wf-conn-list*(*1*−*3*))
next
case (*wf-conn-unary ψ′*)
then have ∗: *FNot ψ′* = *conn c ψs* **using** *conn-inj-not eq assms* **by** *auto*
then have *c* = *ca* **by** (*metis conn-inj-not*(*1*) *corr′ wf-conn-unary*(*2*))
moreover have *ψs* = *l* **using** ∗ *conn-inj-not*(*2*) *corr′ wf-conn-unary*(*1*) **by** *force*
ultimately show *ca* = *c* ∧ *ψs* = *l* **by** *auto*
next
case (*wf-conn-binary ψ′ ψ′′*)
then show *ca* = *c* ∧ *ψs* = *l*
using *eq corr′* **unfolding** *binary-connectives-def* **apply** (*cases ca*, *auto simp add*: *wf-conn-list*)
using *wf-conn-list*(*4*−*7*) *corr′* **by** *metis*+
qed

2.1.3 Subformulas and Properties

A characterization using sub-formulas is interesting for rewriting: we will define our relation on the sub-term level, and then lift the rewriting on the term-level. So the rewriting takes place on a subformula.

inductive *subformula* :: ′*v propo* ⇒ ′*v propo* ⇒ *bool* (**infix** ⪯ *45*) **for** *φ* **where**
subformula-refl[*simp*]: *φ* ⪯ *φ* |
subformula-into-subformula: *ψ* ∈ *set l* ⟹ *wf-conn c l* ⟹ *φ* ⪯ *ψ* ⟹ *φ* ⪯ *conn c l*

On the *subformula-into-subformula*, we can see why we use our *conn* representation: one case is enough to express the subformulas property instead of listing all the cases.

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:
shows *b*: *FNot φ* ⪯ *ψ* ⟹ *φ* ⪯ *ψ*
apply (*induct rule*: *subformula.induct*)
using *subformula-into-subformula wf-conn-unary subformula-refl list.set-intros*(*1*) *subformula-refl*
by (*fastforce intro*: *subformula-into-subformula*)+

lemma *subformula-in-binary-conn*:
assumes *conn*: *c* ∈ *binary-connectives*
shows *f* ⪯ *conn c* [*f*, *g*]
and *g* ⪯ *conn c* [*f*, *g*]
proof −
have *a*: *wf-conn c* (*f*# [*g*]) **using** *conn wf-conn-binary binary-connectives-def* **by** *auto*
moreover have *b*: *f* ⪯ *f* **using** *subformula-refl* **by** *auto*
ultimately show *f* ⪯ *conn c* [*f*, *g*]
by (*metis append-Nil in-set-conv-decomp subformula-into-subformula*)
next
have *a*: *wf-conn c* ([*f*] @ [*g*]) **using** *conn wf-conn-binary binary-connectives-def* **by** *auto*
moreover have *b*: *g* ⪯ *g* **using** *subformula-refl* **by** *auto*
ultimately show *g* ⪯ *conn c* [*f*, *g*] **using** *subformula-into-subformula* **by** *force*
qed

lemma *subformula-trans*:

$\psi \preceq \psi' \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi'$
  apply (*induct* ψ' *rule*: *subformula.inducts*)
  by (*auto simp*: *subformula-into-subformula*)

lemma *subformula-leaf*:
  fixes φ ψ :: *'v propo*
  assumes *incl*: $\varphi \preceq \psi$
  and *simple*: $\psi = FT \vee \psi = FF \vee \psi = FVar\ x$
  shows $\varphi = \psi$
  using *incl simple*
  by (*induct rule*: *subformula.induct*, *auto simp*: *wf-conn-list*)

lemma *subfurmula-not-incl-eq*:
  assumes $\varphi \preceq$ *conn c l*
  and *wf-conn c l*
  and $\forall \psi.\ \psi \in$ *set l* $\longrightarrow \neg\ \varphi \preceq \psi$
  shows $\varphi =$ *conn c l*
  using *assms* **apply** (*induction conn c l rule*: *subformula.induct*, *auto*)
  using *conn-inj* **by** *blast*

lemma *wf-subformula-conn-cases*:
  wf-conn c l $\Longrightarrow \varphi \preceq$ *conn c l* $\longleftrightarrow (\varphi =$ *conn c l* $\vee (\exists \psi.\ \psi \in$ *set l* $\wedge\ \varphi \preceq \psi))$
  apply *standard*
    using *subfurmula-not-incl-eq* **apply** *metis*
  by (*auto simp add*: *subformula-into-subformula*)

lemma *subformula-decomp-explicit*[*simp*]:
  $\varphi \preceq$ *FAnd* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FAnd* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$ (**is** *?P FAnd*)
  $\varphi \preceq$ *FOr* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FOr* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
  $\varphi \preceq$ *FEq* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FEq* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
  $\varphi \preceq$ *FImp* $\psi\ \psi' \longleftrightarrow (\varphi =$ *FImp* $\psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
proof −
  have *wf-conn CAnd* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
  then have $\varphi \preceq$ *conn CAnd* [ψ, ψ'] $\longleftrightarrow$
    $(\varphi =$ *conn CAnd* [ψ, ψ'] $\vee (\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\wedge\ \varphi \preceq \psi''))$
    using *wf-subformula-conn-cases* **by** *metis*
  then show *?P FAnd* **by** *auto*
next
  have *wf-conn COr* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
  then have $\varphi \preceq$ *conn COr* [ψ, ψ'] $\longleftrightarrow$
    $(\varphi =$ *conn COr* [ψ, ψ'] $\vee (\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\wedge\ \varphi \preceq \psi''))$
    using *wf-subformula-conn-cases* **by** *metis*
  then show *?P FOr* **by** *auto*
next
  have *wf-conn CEq* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
  then have $\varphi \preceq$ *conn CEq* [ψ, ψ'] $\longleftrightarrow$
    $(\varphi =$ *conn CEq* [ψ, ψ'] $\vee (\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\wedge\ \varphi \preceq \psi''))$
    using *wf-subformula-conn-cases* **by** *metis*
  then show *?P FEq* **by** *auto*
next
  have *wf-conn CImp* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
  then have $\varphi \preceq$ *conn CImp* [ψ, ψ'] $\longleftrightarrow$
    $(\varphi =$ *conn CImp* [ψ, ψ'] $\vee (\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\wedge\ \varphi \preceq \psi''))$
    using *wf-subformula-conn-cases* **by** *metis*
  then show *?P FImp* **by** *auto*
qed

lemma *wf-conn-helper-facts*[*iff*]:
  wf-conn CNot [φ]
  wf-conn CT []
  wf-conn CF []
  wf-conn (*CVar x*) []
  wf-conn CAnd [φ, ψ]
  wf-conn COr [φ, ψ]
  wf-conn CImp [φ, ψ]
  wf-conn CEq [φ, ψ]
  using *wf-conn.intros* **unfolding** *binary-connectives-def* **by** *fastforce+*

lemma *exists-c-conn*: $\exists$ *c l*. φ = *conn c l* $\wedge$ *wf-conn c l*
  by (*cases* φ) *force+*

lemma *subformula-conn-decomp*[*simp*]:
  assumes *wf*: *wf-conn c l*
  shows $\varphi \preceq$ *conn c l* $\longleftrightarrow$ (φ = *conn c l* $\vee$ ($\exists$ $\psi\in$ *set l*. $\varphi \preceq \psi$)) (**is** *?A* $\longleftrightarrow$ *?B*)
proof (*rule iffI*)
  {
    fix ξ
    have $\varphi \preceq \xi \Longrightarrow \xi$ = *conn c l* $\Longrightarrow$ *wf-conn c l* $\Longrightarrow \forall$ *x*::'*a propo*$\in$*set l*. $\neg$ $\varphi \preceq$ *x* $\Longrightarrow \varphi$ = *conn c l*
      apply (*induct rule*: *subformula.induct*)
        apply *simp*
      using *conn-inj* **by** *blast*
  }
  moreover assume *?A*
  ultimately show *?B* **using** *wf* **by** *metis*
next
  assume *?B*
  then show $\varphi \preceq$ *conn c l* **using** *wf wf-subformula-conn-cases* **by** *blast*
qed

lemma *subformula-leaf-explicit*[*simp*]:
  $\varphi \preceq$ *FT* $\longleftrightarrow \varphi$ = *FT*
  $\varphi \preceq$ *FF* $\longleftrightarrow \varphi$ = *FF*
  $\varphi \preceq$ *FVar x* $\longleftrightarrow \varphi$ = *FVar x*
  apply *auto*
  using *subformula-leaf* **by** *metis* +

The variables inside the formula gives precisely the variables that are needed for the formula.

primrec *vars-of-prop*:: '*v propo* $\Rightarrow$ '*v set* **where**
vars-of-prop FT = {} |
vars-of-prop FF = {} |
vars-of-prop (*FVar x*) = {*x*} |
vars-of-prop (*FNot* φ) = *vars-of-prop* φ |
vars-of-prop (*FAnd* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ |
vars-of-prop (*FOr* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ |
vars-of-prop (*FImp* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ |
vars-of-prop (*FEq* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ

lemma *vars-of-prop-incl-conn*:
  fixes ξ ξ' :: '*v propo list* **and** ψ :: '*v propo* **and** *c* :: '*v connective*
  assumes *corr*: *wf-conn c l* **and** *incl*: $\psi \in$ *set l*
  shows *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
proof (*cases c rule*: *connective-cases-arity-2*)

```
  case nullary
  then have False using corr incl by auto
  then show vars-of-prop ψ ⊆ vars-of-prop (conn c l) by blast
next
  case binary note c = this
  then obtain a b where ab: l = [a, b]
    using wf-conn-bin-list-length list-length2-decomp corr by metis
  then have ψ = a ∨ ψ = b using incl by auto
  then show vars-of-prop ψ ⊆ vars-of-prop (conn c l)
    using ab c unfolding binary-connectives-def by auto
next
  case unary note c = this
  fix φ :: 'v propo
  have l = [ψ] using corr c incl split-list by force
  then show vars-of-prop ψ ⊆ vars-of-prop (conn c l) using c by auto
qed
```

The set of variables is compatible with the subformula order.

```
lemma subformula-vars-of-prop:
  φ ⪯ ψ ⟹ vars-of-prop φ ⊆ vars-of-prop ψ
  apply (induct rule: subformula.induct)
  apply simp
  using vars-of-prop-incl-conn by blast
```

2.1.4 Positions

Instead of 1 or 2 we use L or R

```
datatype sign = L | R
```

We use *nil* instead of ε.

```
fun pos :: 'v propo ⇒ sign list set where
pos FF = {[]} |
pos FT = {[]} |
pos (FVar x) = {[]} |
pos (FAnd φ ψ) = {[]} ∪ { L # p | p. p∈ pos φ} ∪ { R # p | p. p∈ pos ψ} |
pos (FOr φ ψ) = {[]} ∪ { L # p | p. p∈ pos φ} ∪ { R # p | p. p∈ pos ψ} |
pos (FEq φ ψ) = {[]} ∪ { L # p | p. p∈ pos φ} ∪ { R # p | p. p∈ pos ψ} |
pos (FImp φ ψ) = {[]} ∪ { L # p | p. p∈ pos φ} ∪ { R # p | p. p∈ pos ψ} |
pos (FNot φ) = {[]} ∪ { L # p | p. p∈ pos φ}

lemma finite-pos: finite (pos φ)
  by (induct φ, auto)

lemma finite-inj-comp-set:
  fixes s :: 'v set
  assumes finite: finite s
  and inj: inj f
  shows card ({f p |p. p ∈ s}) = card s
  using finite
proof (induct s rule: finite-induct)
  show card {f p |p. p ∈ {}} = card {} by auto
next
  fix x :: 'v and s:: 'v set
  assume f: finite s and notin: x ∉ s
  and IH: card {f p |p. p ∈ s} = card s
```

have *f'*: *finite* {*f p* |*p*. *p* ∈ *insert x s*} **using** *f* **by** *auto*
have *notin'*: *f x* ∉ {*f p* |*p*. *p* ∈ *s*} **using** *notin inj injD* **by** *fastforce*
have {*f p* |*p*. *p* ∈ *insert x s*} = *insert* (*f x*) {*f p* |*p*. *p*∈ *s*} **by** *auto*
then have *card* {*f p* |*p*. *p* ∈ *insert x s*} = *1* + *card* {*f p* |*p*. *p* ∈ *s*}
using *finite card-insert-disjoint f' notin'* **by** *auto*
moreover have ... = *card* (*insert x s*) **using** *notin f IH* **by** *auto*
finally show *card* {*f p* |*p*. *p* ∈ *insert x s*} = *card* (*insert x s*) **.**
qed

lemma *cons-inject*:
inj ((#) *s*)
by (*meson injI list.inject*)

lemma *finite-insert-nil-cons*:
finite s ⟹ *card* (*insert* [] {*L* # *p* |*p*. *p* ∈ *s*}) = *1* + *card* {*L* # *p* |*p*. *p* ∈ *s*}
using *card-insert-disjoint* **by** *auto*

lemma *cord-not*[*simp*]:
card (*pos* (*FNot* φ)) = *1* + *card* (*pos* φ)
by (*simp add*: *cons-inject finite-inj-comp-set finite-pos*)

lemma *card-seperate*:
assumes *finite s1* **and** *finite s2*
shows *card* ({*L* # *p* |*p*. *p* ∈ *s1*} ∪ {*R* # *p* |*p*. *p* ∈ *s2*}) = *card* ({*L* # *p* |*p*. *p* ∈ *s1*})
+ *card*({*R* # *p* |*p*. *p* ∈ *s2*}) (**is** *card* (*?L*∪*?R*) = *card ?L* + *card ?R*)
proof −
have *finite ?L* **using** *assms* **by** *auto*
moreover have *finite ?R* **using** *assms* **by** *auto*
moreover have *?L* ∩ *?R* = {} **by** *blast*
ultimately show *?thesis* **using** *assms card-Un-disjoint* **by** *blast*
qed

definition *prop-size* **where** *prop-size* φ = *card* (*pos* φ)

lemma *prop-size-vars-of-prop*:
fixes φ :: *'v propo*
shows *card* (*vars-of-prop* φ) ≤ *prop-size* φ

unfolding *prop-size-def* **apply** (*induct* φ, *auto simp add*: *cons-inject finite-inj-comp-set finite-pos*)
proof −
fix *φ1 φ2* :: *'v propo*
assume *IH1*: *card* (*vars-of-prop φ1*) ≤ *card* (*pos φ1*)
and *IH2*: *card* (*vars-of-prop φ2*) ≤ *card* (*pos φ2*)
let *?L* = {*L* # *p* |*p*. *p* ∈ *pos φ1*}
let *?R* = {*R* # *p* |*p*. *p* ∈ *pos φ2*}
have *card* (*?L* ∪ *?R*) = *card ?L* + *card ?R*
using *card-seperate finite-pos* **by** *blast*
moreover have ... = *card* (*pos φ1*) + *card* (*pos φ2*)
by (*simp add*: *cons-inject finite-inj-comp-set finite-pos*)
moreover have ... ≥ *card* (*vars-of-prop φ1*) + *card* (*vars-of-prop φ2*) **using** *IH1 IH2* **by** *arith*
then have ... ≥ *card* (*vars-of-prop φ1* ∪ *vars-of-prop φ2*) **using** *card-Un-le le-trans* **by** *blast*
ultimately
show *card* (*vars-of-prop φ1* ∪ *vars-of-prop φ2*) ≤ *Suc* (*card* (*?L* ∪ *?R*))
card (*vars-of-prop φ1* ∪ *vars-of-prop φ2*) ≤ *Suc* (*card* (*?L* ∪ *?R*))
card (*vars-of-prop φ1* ∪ *vars-of-prop φ2*) ≤ *Suc* (*card* (*?L* ∪ *?R*))

card (*vars-of-prop* φ*1* ∪ *vars-of-prop* φ*2*) ≤ *Suc* (*card* (*?L* ∪ *?R*))
by *auto*
qed

value *pos* (*FImp* (*FAnd* (*FVar P*) (*FVar Q*)) (*FOr* (*FVar P*) (*FVar Q*)))

inductive *path-to* :: *sign list* ⇒ *'v propo* ⇒ *'v propo* ⇒ *bool* **where**
path-to-refl[*intro*]: *path-to* [] φ φ |
path-to-l: *c*∈*binary-connectives* ∨ *c* = *CNot* ⟹ *wf-conn c* (φ#*l*) ⟹ *path-to p* φ φ'⟹
path-to (*L*#*p*) (*conn c* (φ#*l*)) φ' |
path-to-r: *c*∈*binary-connectives* ⟹ *wf-conn c* (ψ#φ#[]) ⟹ *path-to p* φ φ' ⟹
path-to (*R*#*p*) (*conn c* (ψ#φ#[])) φ'

There is a deep link between subformulas and pathes: a (correct) path leads to a subformula and a subformula is associated to a given path.

lemma *path-to-subformula*:
path-to p φ φ' ⟹ φ' ≼ φ
apply (*induct rule*: *path-to.induct*)
apply *simp*
apply (*metis list.set-intros*(*1*) *subformula-into-subformula*)
using *subformula-trans subformula-in-binary-conn*(*2*) **by** *metis*

lemma *subformula-path-exists*:
fixes φ φ':: *'v propo*
shows φ' ≼ φ ⟹ ∃ *p*. *path-to p* φ φ'
proof (*induct rule*: *subformula.induct*)
case *subformula-refl*
have *path-to* [] φ' φ' **by** *auto*
then show ∃ *p*. *path-to p* φ' φ' **by** *metis*
next
case (*subformula-into-subformula* ψ *l c*)
note *wf* = *this*(*2*) **and** *IH* = *this*(*4*) **and** ψ = *this*(*1*)
then obtain *p* **where** *p*: *path-to p* ψ φ' **by** *metis*
{
fix *x* :: *'v*
assume *c* = *CT* ∨ *c* = *CF* ∨ *c* = *CVar x*
then have *False* **using** *subformula-into-subformula* **by** *auto*
then have ∃ *p*. *path-to p* (*conn c l*) φ' **by** *blast*
}
moreover {
assume *c*: *c* = *CNot*
then have *l* = [ψ] **using** *wf* ψ *wf-conn-Not-decomp* **by** *fastforce*
then have *path-to* (*L* # *p*) (*conn c l*) φ' **by** (*metis c wf-conn-unary p path-to-l*)
then have ∃ *p*. *path-to p* (*conn c l*) φ' **by** *blast*
}
moreover {
assume *c*: *c*∈ *binary-connectives*
obtain *a b* **where** *ab*: [*a*, *b*] = *l* **using** *subformula-into-subformula c wf-conn-bin-list-length list-length2-decomp* **by** *metis*
then have *a* = ψ ∨ *b* = ψ **using** ψ **by** *auto*
then have *path-to* (*L* # *p*) (*conn c l*) φ' ∨ *path-to* (*R* # *p*) (*conn c l*) φ' **using** *c path-to-l path-to-r p ab* **by** (*metis wf-conn-binary*)
then have ∃ *p*. *path-to p* (*conn c l*) φ' **by** *blast*
}
ultimately show ∃ *p*. *path-to p* (*conn c l*) φ' **using** *connective-cases-arity* **by** *metis*
qed

```
fun replace-at :: sign list ⇒ 'v propo ⇒ 'v propo ⇒ 'v propo where
replace-at [] - ψ = ψ |
replace-at (L # l) (FAnd φ φ') ψ = FAnd (replace-at l φ ψ) φ'|
replace-at (R # l) (FAnd φ φ') ψ = FAnd φ (replace-at l φ' ψ) |
replace-at (L # l) (FOr φ φ') ψ = FOr (replace-at l φ ψ) φ' |
replace-at (R # l) (FOr φ φ') ψ = FOr φ (replace-at l φ' ψ) |
replace-at (L # l) (FEq φ φ') ψ = FEq (replace-at l φ ψ) φ'|
replace-at (R # l) (FEq φ φ') ψ = FEq φ (replace-at l φ' ψ) |
replace-at (L # l) (FImp φ φ') ψ = FImp (replace-at l φ ψ) φ'|
replace-at (R # l) (FImp φ φ') ψ = FImp φ (replace-at l φ' ψ) |
replace-at (L # l) (FNot φ) ψ = FNot (replace-at l φ ψ)
```

2.2 Semantics over the Syntax

Given the syntax defined above, we define a semantics, by defining an evaluation function *eval*. This function is the bridge between the logic as we define it here and the built-in logic of Isabelle.

```
fun eval :: ('v ⇒ bool) ⇒ 'v propo ⇒ bool (infix ⊨ 50) where
𝒜 ⊨ FT = True |
𝒜 ⊨ FF = False |
𝒜 ⊨ FVar v = (𝒜 v) |
𝒜 ⊨ FNot φ = (¬(𝒜⊨ φ)) |
𝒜 ⊨ FAnd φ1 φ2 = (𝒜⊨φ1 ∧ 𝒜⊨φ2) |
𝒜 ⊨ FOr φ1 φ2 = (𝒜⊨φ1 ∨ 𝒜⊨φ2) |
𝒜 ⊨ FImp φ1 φ2 = (𝒜⊨φ1 ⟶ 𝒜⊨φ2) |
𝒜 ⊨ FEq φ1 φ2 = (𝒜⊨φ1 ⟷ 𝒜 ⊨φ2)
```

```
definition evalf (infix ⊨f 50) where
evalf φ ψ = (∀ A. A ⊨ φ ⟶ A ⊨ ψ)
```

The deduction rule is in the book. And the proof looks like to the one of the book.

```
theorem deduction-theorem:
  φ ⊨f ψ ⟷ (∀ A. A ⊨ FImp φ ψ)
proof
  assume H: φ ⊨f ψ
  {
    fix A
    have A ⊨ FImp φ ψ
      proof (cases A ⊨ φ)
        case True
        then have A ⊨ ψ using H unfolding evalf-def by metis
        then show A ⊨ FImp φ ψ by auto
      next
        case False
        then show A ⊨ FImp φ ψ by auto
      qed
  }
  then show ∀ A. A ⊨ FImp φ ψ by blast
next
  assume A: ∀ A. A ⊨ FImp φ ψ
  show φ ⊨f ψ
    proof (rule ccontr)
      assume ¬ φ ⊨f ψ
      then obtain A where A ⊨ φ and ¬ A ⊨ ψ using evalf-def by metis
```

then have $\neg$ $A \models$ *FImp* φ ψ **by** *auto*
then show *False* **using** *A* **by** *blast*
qed
qed

A shorter proof:

lemma $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models$ *FImp* $\varphi\ \psi)$
by (*simp add*: *evalf-def*)

definition *same-over-set*:: $('v \Rightarrow bool) \Rightarrow ('v \Rightarrow bool) \Rightarrow 'v\ set \Rightarrow bool$ **where**
same-over-set $A\ B\ S = (\forall c \in S.\ A\ c = B\ c)$

If two mapping A and B have the same value over the variables, then the same formula are satisfiable.

lemma *same-over-set-eval*:
assumes *same-over-set* $A\ B$ (*vars-of-prop* φ)
shows $A \models \varphi \longleftrightarrow B \models \varphi$
using *assms* **unfolding** *same-over-set-def* **by** (*induct* φ, *auto*)

end