Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

April 26, 2016

Contents

Chapter 1

More Standard Theorems

This chapter contains additional lemmas built on top of HOL.

end

theory *Multiset-More*
imports ~~/*src*/*HOL*/*Library*/*Multiset-Order*
begin

1.1 More about Multisets

Isabelle's theory of finite multisets is not as developed as other areas, such as lists and sets. The present theory introduces some missing concepts and lemmas. Some of it is expected to move to Isabelle's library.

1.1.1 Basic Setup

declare
diff-single-trivial [*simp*]
in-image-mset [*iff*]
image-mset.compositionality [*simp*]

mset-leD[*dest*, *intro?*]

Multiset.in-multiset-in-set[*simp*]

lemma *image-mset-cong2*[*cong*]:
$(\bigwedge x.\ x \in\#\ M \Longrightarrow f\ x = g\ x) \Longrightarrow M = N \Longrightarrow \mathit{image\text{-}mset}\ f\ M = \mathit{image\text{-}mset}\ g\ N$
by (*hypsubst*, *rule image-mset-cong*)

lemma *subset-msetE* [*elim*!]:
$[|A \subset\#\ B;\ [|A \subseteq\#\ B;\ \sim\ (B \subseteq\# A)|] ==> R|] ==> R$
unfolding *subseteq-mset-def subset-mset-def* **by** (*meson mset-less-eqI subset-mset.eq-iff*)

1.1.2 Lemmas about intersections

lemma *mset-inter-single*:

$x \in\# \Sigma \Longrightarrow \Sigma \;\#\cap\; \{\#x\#\} = \{\#x\#\}$
$x \notin\# \Sigma \Longrightarrow \Sigma \;\#\cap\; \{\#x\#\} = \{\#\}$
apply (*simp add*: *mset-le-single subset-mset.inf-absorb2*)
by (*simp add*: *multiset-inter-def*)

1.1.3 Lemmas about size

This sections adds various lemmas about size. Most lemmas have a finite set equivalent.

lemma *size-mset-SucE*: $size\ A = Suc\ n \Longrightarrow (\bigwedge a\ B.\ A = \{\#a\#\} + B \Longrightarrow size\ B = n \Longrightarrow P) \Longrightarrow P$
by (*cases A*) (*auto simp add*: *ac-simps*)

lemma *size-Suc-Diff1*:
$x \in\# \Sigma \Longrightarrow Suc\ (size\ (\Sigma - \{\#x\#\})) = size\ \Sigma$
using *arg-cong*[*OF insert-DiffM*, *of - - size*] **by** *simp*

lemma *size-Diff-singleton*: $x \in\# \Sigma \Longrightarrow size\ (\Sigma - \{\#x\#\}) = size\ \Sigma - 1$
by (*simp add*: *size-Suc-Diff1* [*symmetric*])

lemma *size-Diff-singleton-if*: $size\ (A - \{\#x\#\}) = (if\ x \in\# A\ then\ size\ A - 1\ else\ size\ A)$
by (*simp add*: *size-Diff-singleton*)

lemma *size-Un-Int*:
$size\ A + size\ B = size\ (A \;\#\cup\; B) + size\ (A \;\#\cap\; B)$
proof −
have $*$: $A + B = B + (A - B + (A - (A - B)))$
by (*simp add*: *subset-mset.add-diff-inverse union-commute*)
have $size\ B + size\ (A - B) = size\ A + size\ (B - A)$
unfolding *size-union*[*symmetric*] *subset-mset.sup-commute sup-subset-mset-def*[*symmetric*]
by *blast*
then show *?thesis* **unfolding** *multiset-inter-def size-union*[*symmetric*] $*$
by (*auto simp add*: *sup-subset-mset-def*)
qed

lemma *size-Un-disjoint*:
assumes $A \;\#\cap\; B = \{\#\}$
shows $size\ (A \;\#\cup\; B) = size\ A + size\ B$
using *assms size-Un-Int* [*of A B*] **by** *simp*

lemma *size-Diff-subset-Int*:
shows $size\ (\Sigma - \Sigma') = size\ \Sigma - size\ (\Sigma \;\#\cap\; \Sigma')$
proof −
have $*$: $\Sigma - \Sigma' = \Sigma - \Sigma \;\#\cap\; \Sigma'$ **by** (*auto simp add*: *multiset-eq-iff*)
show *?thesis* **unfolding** $*$ **using** *size-Diff-submset subset-mset.inf.cobounded1* **by** *blast*
qed

lemma *diff-size-le-size-Diff*: $size\ (\Sigma{::}\ \text{-}\ multiset) - size\ \Sigma' \le size\ (\Sigma - \Sigma')$
proof−
have $size\ \Sigma - size\ \Sigma' \le size\ \Sigma - size\ (\Sigma \;\#\cap\; \Sigma')$
using *size-mset-mono diff-le-mono2 subset-mset.inf-le2* **by** *blast*
also have $\ldots = size(\Sigma{-}\Sigma')$ **using** *assms* **by**(*simp add*: *size-Diff-subset-Int*)
finally show *?thesis* **.**
qed

lemma *size-Diff1-less*: $x{\in}\# \Sigma \Longrightarrow size\ (\Sigma - \{\#x\#\}) < size\ \Sigma$
apply (*rule Suc-less-SucD*)

by (*simp add*: *size-Suc-Diff1*)

lemma *size-Diff2-less*: $x \in\# \Sigma \Longrightarrow y \in\# \Sigma \Longrightarrow$ *size* $(\Sigma - \{\#x\#\} - \{\#y\#\}) <$ *size* Σ
using *nonempty-has-size* **by** (*fastforce intro*!: *diff-Suc-less simp add*: *size-Diff1-less size-Diff-subset-Int mset-inter-single*)

lemma *size-Diff1-le*: *size* $(\Sigma - \{\#x\#\}) \leq$ *size* Σ
by (*cases* $x \in\# \Sigma$) (*simp-all add*: *size-Diff1-less less-imp-le*)

lemma *size-psubset*: $(\Sigma$:: - *multiset*$) \leq\# \Sigma' \Longrightarrow$ *size* $\Sigma <$ *size* $\Sigma' \Longrightarrow \Sigma <\# \Sigma'$
using *less-irrefl subset-mset-def* **by** *blast*

1.1.4 Multiset Extension of Multiset Ordering

The *op* $\#\subset\#\#$ and *op* $\#\subseteq\#\#$ operators are introduced as the multiset extension of the multiset orderings of *op* $\#\subset\#$ and *op* $\#\subseteq\#$.

definition *less-mset-mset* :: ($'a$ :: *order*) *multiset multiset* $\Rightarrow$ $'a$ *multiset multiset* $\Rightarrow$ *bool*
(**infix** $\#<\#\#$ *50*)
where
$M' \#<\#\# M \longleftrightarrow (M', M) \in$ *mult* $\{(x', x).\ x' \#<\# x\}$

definition *le-mset-mset* :: ($'a$ :: *order*) *multiset multiset* $\Rightarrow$ $'a$ *multiset multiset* $\Rightarrow$ *bool*
(**infix** $\#<=\#\#$ *50*)
where
$M' \#<=\#\# M \longleftrightarrow M' \#<\#\# M \vee M' = M$

notation *less-mset-mset* (**infix** $\#\subset\#\#$ *50*)
notation *le-mset-mset* (**infix** $\#\subseteq\#\#$ *50*)

lemmas *less-mset-mset*$_{DM}$ = *order.mult*$_{DM}$[*OF order-multiset*, *folded less-mset-mset-def*]
lemmas *less-mset-mset*$_{HO}$ = *order.mult*$_{HO}$[*OF order-multiset*, *folded less-mset-mset-def*]

interpretation *multiset-multiset-order*: *order*
le-mset-mset :: ($'a$ :: *linorder*) *multiset multiset* $\Rightarrow$ ($'a$ :: *linorder*) *multiset multiset* $\Rightarrow$ *bool*
less-mset-mset :: ($'a$ :: *linorder*) *multiset multiset* $\Rightarrow$ ($'a$::*linorder*) *multiset multiset* $\Rightarrow$ *bool*
unfolding *less-mset-mset-def*[*abs-def*] *le-mset-mset-def*[*abs-def*] *less-multiset-def*[*abs-def*]
by (*rule order.order-mult*)+ *standard*

interpretation *multiset-multiset-linorder*: *linorder*
le-mset-mset :: ($'a$ :: *linorder*) *multiset multiset* $\Rightarrow$ ($'a$ :: *linorder*) *multiset multiset* $\Rightarrow$ *bool*
less-mset-mset :: ($'a$ :: *linorder*) *multiset multiset* $\Rightarrow$ ($'a$::*linorder*) *multiset multiset* $\Rightarrow$ *bool*
unfolding *less-mset-mset-def*[*abs-def*] *le-mset-mset-def*[*abs-def*]
by (*rule linorder.linorder-mult*[*OF linorder-multiset*])

lemma *wf-less-mset-mset*: *wf* $\{(\Sigma$ :: ($'a$ :: *wellorder*) *multiset multiset*, $T).\ \Sigma \#\subset\#\# T\}$
unfolding *less-mset-mset-def* **by** (*auto intro*: *wf-mult wf-less-multiset*)

interpretation *multiset-multiset-wellorder*: *wellorder*
le-mset-mset :: ($'a$::*wellorder*) *multiset multiset* $\Rightarrow$ ($'a$::*wellorder*) *multiset multiset* $\Rightarrow$ *bool*
less-mset-mset :: ($'a$::*wellorder*) *multiset multiset* $\Rightarrow$ ($'a$::*wellorder*) *multiset multiset* $\Rightarrow$ *bool*
by *unfold-locales* (*blast intro*: *wf-less-mset-mset*[*unfolded wf-def*, *rule-format*])

lemma *union-less-mset-mset-mono2*: $B \#\subset\#\# D \Longrightarrow C + B \#\subset\#\# C + (D$::$'a$::*order multiset multiset*)
apply (*unfold less-mset-mset-def mult-def*)

apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union*)
apply (*blast intro*: *mult1-union trancl-trans*)
done

lemma *union-less-mset-mset-diff-plus*:
$U \leq\# \Sigma \Longrightarrow T \#\subset\#\# U \Longrightarrow \Sigma - U + T \#\subset\#\# \Sigma$
apply (*drule subset-mset.diff-add*[*symmetric*])
using *union-less-mset-mset-mono2*[*of* T U $\Sigma - U$] **by** *simp*

lemma *ex-gt-imp-less-mset-mset*:
$(\exists y :: 'a :: linorder\ multiset \in\# T. (\forall x. x \in\# \Sigma \longrightarrow x \#\subset\# y)) \Longrightarrow \Sigma \#\subset\#\# T$
using *less-mset-mset*$_{HO}$ **by** (*metis count-greater-zero-iff count-inI less-nat-zero-code multiset-linorder.not-less-iff-gr-or-eq*)

1.1.5 Remove

lemma *set-mset-minus-replicate-mset*[*simp*]:
$n \geq count\ A\ a \Longrightarrow set\text{-}mset\ (A - replicate\text{-}mset\ n\ a) = set\text{-}mset\ A - \{a\}$
$n < count\ A\ a \Longrightarrow set\text{-}mset\ (A - replicate\text{-}mset\ n\ a) = set\text{-}mset\ A$
unfolding *set-mset-def* **by** (*auto split*: *if-split simp*: *not-in-iff*)

abbreviation *removeAll-mset* :: $'a \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **where**
$removeAll\text{-}mset\ C\ M \equiv M - replicate\text{-}mset\ (count\ M\ C)\ C$

lemma *mset-removeAll*[*simp*, *code*]:
$removeAll\text{-}mset\ C\ (mset\ L) = mset\ (removeAll\ C\ L)$
by (*induction L*) (*auto simp*: *ac-simps multiset-eq-iff split*: *if-split-asm*)

lemma *removeAll-mset-filter-mset*:
$removeAll\text{-}mset\ C\ M = filter\text{-}mset\ (op \neq C)\ M$
by (*induction M*) (*auto simp*: *ac-simps multiset-eq-iff*)

abbreviation *remove1-mset* :: $'a \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **where**
$remove1\text{-}mset\ C\ M \equiv M - \{\#C\#\}$

lemma *remove1-mset-remove1*[*code*]:
$remove1\text{-}mset\ C\ (mset\ L) = mset\ (remove1\ C\ L)$
by *auto*

lemma *in-remove1-mset-neq*:
assumes *ab*: $a \neq b$
shows $a \in\# remove1\text{-}mset\ b\ C \longleftrightarrow a \in\# C$
proof −
have $count\ \{\#b\#\}\ a = 0$
using *ab* **by** *simp*
then show *?thesis*
by (*metis* (*no-types*) *count-diff diff-zero mem-Collect-eq set-mset-def*)
qed

lemma *size-mset-removeAll-mset-le-iff*:
$size\ (removeAll\text{-}mset\ x\ M) < size\ M \longleftrightarrow x \in\# M$
apply (*rule iffI*)
apply (*force intro*: *count-inI*)
apply (*rule mset-less-size*)
apply (*auto simp*: *subset-mset-def multiset-eq-iff*)

done

lemma *size-mset-remove1-mset-le-iff*:
size (*remove1-mset x M*) < *size M* ⟷ *x* ∈# *M*
apply (*rule iffI*)
using *less-irrefl* **apply** *fastforce*
apply (*rule mset-less-size*)
by (*auto elim*: *in-countE simp*: *subset-mset-def multiset-eq-iff*)

lemma *set-mset-remove1-mset*[*simp*]:
set-mset (*remove1-mset L* (*mset W*)) = *set* (*remove1 L W*)
by (*metis mset-remove1 set-mset-mset*)

1.1.6 Replicate

lemma *replicate-mset-plus*: *replicate-mset* (*a* + *b*) *C* = *replicate-mset a C* + *replicate-mset b C*
by (*induct a*) (*auto simp*: *ac-simps*)

lemma *mset-replicate-replicate-mset*:
mset (*replicate n L*) = *replicate-mset n L*
by (*induction n*) *auto*

lemma *set-mset-single-iff-replicate-mset*:
set-mset U = {*a*} ⟷ (∃ *n*>*0*. *U* = *replicate-mset n a*) (**is** *?S* ⟷ *?R*)
proof
assume *?R*
then show *?S* **by** *auto*
next
assume *?S*
show *?R*
proof (*rule ccontr*)
assume ¬ *?R*
have ∀ *n*. *U* ≠ *replicate-mset n a*
using ‹*?S*› ‹¬ *?R*› **by** (*metis gr-zeroI insert-not-empty set-mset-replicate-mset-subset*)
then obtain *b* **where** *b* ∈# *U* **and** *b* ≠ *a*
by (*metis count-replicate-mset mem-Collect-eq multiset-eqI neq0-conv set-mset-def*)
then show *False*
using ‹*?S*› **by** *auto*
qed
qed

1.1.7 Multiset and set conversion

lemma *count-mset-set-if*:
count (*mset-set A*) *a* = (*if a* ∈ *A* ∧ *finite A then 1 else 0*)
by *auto*

lemma *mset-set-set-mset-empty-mempty*[*iff*]:
mset-set (*set-mset D*) = {#} ⟷ *D* = {#}
by (*auto dest*: *arg-cong*[*of - - set-mset*])

lemma *size-mset-set-card*:
finite S ⟹ *size* (*mset-set S*) = *card S*
by (*induction S rule*: *finite-induct*) *auto*

lemma *count-mset-set-le-one*: *count* (*mset-set A*) *x* ≤ *1*

by (*metis count-mset-set*(*1*) *count-mset-set*(*2*) *count-mset-set*(*3*) *eq-iff le-numeral-extra*(*1*))

lemma *mset-set-subseteq-mset-set*[*iff*]:
assumes *finite A finite B*
shows *mset-set A* $\subseteq\#$ *mset-set B* $\longleftrightarrow$ *A* $\subseteq$ *B*
by (*metis assms contra-subsetD count-mset-set*(*1*,*3*) *count-mset-set-le-one finite-set-mset-mset-set less-eq-nat.simps*(*1*) *mset-less-eqI set-mset-mono*)

lemma *mset-set-set-mset-subseteq*[*simp*]: *mset-set* (*set-mset A*) $\subseteq\#$ *A*
by (*metis count-mset-set*(*1*,*3*) *finite-set-mset less-eq-nat.simps*(*1*) *less-one mem-Collect-eq mset-less-eqI not-less set-mset-def*)

lemma *mset-sorted-list-of-set*[*simp*]:
mset (*sorted-list-of-set A*) = *mset-set A*
by (*metis mset-sorted-list-of-multiset sorted-list-of-mset-set*)

lemma *mset-take-subseteq*: *mset* (*take n xs*) $\subseteq\#$ *mset xs*
apply (*induct xs arbitrary*: *n*)
apply *simp*
by (*case-tac n*) *simp-all*

1.1.8 Removing duplicates

definition *remdups-mset* :: *'v multiset* $\Rightarrow$ *'v multiset* **where**
remdups-mset S = *mset-set* (*set-mset S*)

lemma *remdups-mset-in*[*iff*]: *a* $\in\#$ *remdups-mset A* $\longleftrightarrow$ *a* $\in\#$ *A*
unfolding *remdups-mset-def* **by** *auto*

lemma *count-remdups-mset-eq-1*: *a* $\in\#$ *remdups-mset A* $\longleftrightarrow$ *count* (*remdups-mset A*) *a* = *1*
unfolding *remdups-mset-def* **by** (*auto simp*: *count-eq-zero-iff intro*: *count-inI*)

lemma *remdups-mset-empty*[*simp*]:
remdups-mset {#} = {#}
unfolding *remdups-mset-def* **by** *auto*

lemma *remdups-mset-singleton*[*simp*]:
remdups-mset {#*a*#} = {#*a*#}
unfolding *remdups-mset-def* **by** *auto*

lemma *set-mset-remdups*[*simp*]: *set-mset* (*remdups-mset C*) = *set-mset C*
by *auto*

lemma *remdups-mset-eq-empty*[*iff*]:
remdups-mset D = {#} $\longleftrightarrow$ *D* = {#}
unfolding *remdups-mset-def* **by** *blast*

lemma *remdups-mset-singleton-sum*[*simp*]:
remdups-mset ({#*a*#} + *A*) = (*if a* $\in\#$ *A then remdups-mset A else* {#*a*#} + *remdups-mset A*)
remdups-mset (*A*+{#*a*#}) = (*if a* $\in\#$ *A then remdups-mset A else* {#*a*#} + *remdups-mset A*)
unfolding *remdups-mset-def* **by** (*simp-all add*: *insert-absorb*)

lemma *mset-remdups-remdups-mset*[*simp*]:
mset (*remdups D*) = *remdups-mset* (*mset D*)
by (*induction D*) (*auto simp add*: *ac-simps*)

definition *distinct-mset* :: *'a multiset* ⇒ *bool* **where**
distinct-mset $S \longleftrightarrow (\forall a.\ a \in\# S \longrightarrow count\ S\ a = 1)$

lemma *distinct-mset-empty*[*simp*]: *distinct-mset* {#}
  unfolding *distinct-mset-def* **by** *auto*

lemma *distinct-mset-singleton*[*simp*]: *distinct-mset* {#*a*#}
  unfolding *distinct-mset-def* **by** *auto*

definition *distinct-mset-set* :: *'a multiset set* ⇒ *bool* **where**
distinct-mset-set $\Sigma \longleftrightarrow (\forall S \in\Sigma.\ distinct\text{-}mset\ S)$

lemma *distinct-mset-set-empty*[*simp*]:
  distinct-mset-set {}
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-singleton*[*iff*]:
  distinct-mset-set $\{A\} \longleftrightarrow$ *distinct-mset* A
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-insert*[*iff*]:
  distinct-mset-set (*insert* S Σ) $\longleftrightarrow$ (*distinct-mset* S $\wedge$ *distinct-mset-set* Σ)
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-union*[*iff*]:
  distinct-mset-set $(\Sigma \cup \Sigma')$ $\longleftrightarrow$ (*distinct-mset-set* Σ $\wedge$ *distinct-mset-set* Σ')
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-union*:
  assumes *dist*: *distinct-mset* $(A + B)$
  shows *distinct-mset* A
proof −
  obtain *aa* :: *'a multiset* ⇒ *'a* **where**
    f2: $\forall m.\ \neg$ *distinct-mset* $m \vee (\forall a.\ (a::'a) \notin\# m \vee count\ m\ a = 1)$
      $\forall m.\ aa\ m \in\# m \wedge count\ m\ (aa\ m) \neq 1 \vee$ *distinct-mset* m
    by (*metis* (*full-types*) *distinct-mset-def*)
  then have *count* $(A + B)\ (aa\ A) = 1 \vee$ *distinct-mset* A
    using *dist* **by** (*meson mset-leD mset-le-add-left*)
  then show *?thesis*
    using *f2* **by** (*metis* (*no-types*) *One-nat-def add-is-1 count-union mem-Collect-eq order-less-irrefl*
      set-mset-def)
qed

lemma *distinct-mset-minus*[*simp*]:
  distinct-mset $A \Longrightarrow$ *distinct-mset* $(A - B)$
  by (*metis Multiset.diff-le-self mset-le-exists-conv distinct-mset-union*)

lemma *in-distinct-mset-set-distinct-mset*:
  $a \in \Sigma \Longrightarrow$ *distinct-mset-set* $\Sigma \Longrightarrow$ *distinct-mset* a
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-remdups-mset*[*simp*]: *distinct-mset* (*remdups-mset* S)
  using *count-remdups-mset-eq-1* **unfolding** *distinct-mset-def* **by** *metis*

lemma *distinct-mset-distinct*[*simp*]:
  distinct-mset (*mset* x) = *distinct* x

unfolding *distinct-mset-def* **by** (*auto simp*: *distinct-count-atmost-1 not-in-iff*[*symmetric*])

lemma *distinct-mset-mset-set*:
distinct-mset (*mset-set A*)
unfolding *distinct-mset-def count-mset-set-if* **by** (*auto simp*: *not-in-iff*)

lemma *distinct-mset-rempdups-union-mset*:
assumes *distinct-mset A* **and** *distinct-mset B*
shows $A \mathbin{\#\cup} B =$ *remdups-mset* $(A + B)$
using *assms nat-le-linear* **unfolding** *remdups-mset-def*
by (*force simp add*: *multiset-eq-iff max-def count-mset-set-if distinct-mset-def not-in-iff*)

lemma *distinct-mset-set-distinct*:
distinct-mset-set (*mset* ‘ *set Cs*) $\longleftrightarrow$ ($\forall$ *c*$\in$ *set Cs. distinct c*)
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-add-single*:
distinct-mset ($\{\#a\#\} + L$) $\longleftrightarrow$ *distinct-mset* $L \wedge a \notin\# L$
unfolding *distinct-mset-def*
apply (*rule iffI*)
 prefer *2* **apply** (*auto simp*: *not-in-iff*)[]
apply *standard*
 apply (*intro allI*)
 apply (*rename-tac aa*, *case-tac a* = *aa*)
 by (*auto split*: *if-split-asm*)

lemma *distinct-mset-single-add*:
distinct-mset ($L + \{\#a\#\}$) $\longleftrightarrow$ *distinct-mset* $L \wedge a \notin\# L$
unfolding *add.commute*[*of L* $\{\#a\#\}$] *distinct-mset-add-single* **by** *fast*

lemma *distinct-mset-size-eq-card*:
distinct-mset $C \Longrightarrow$ *size* C = *card* (*set-mset* C)
by (*induction C*) (*auto simp*: *distinct-mset-single-add*)

Another characterisation of *distinct-mset*

lemma *distinct-mset-count-less-1*:
distinct-mset $S \longleftrightarrow$ ($\forall$ *a. count S a* $\leq$ *1*)
using *eq-iff nat-le-linear* **unfolding** *distinct-mset-def* **by** *fastforce*

lemma *distinct-mset-add*:
distinct-mset $(L + L')$ $\longleftrightarrow$ *distinct-mset* $L \wedge$ *distinct-mset* $L' \wedge L \mathbin{\#\cap} L' = \{\#\}$ (**is** *?A* $\longleftrightarrow$ *?B*)
proof (*rule iffI*)
assume *?A*
have *L*: *distinct-mset L*
 using ‹*distinct-mset* $(L + L')$› *distinct-mset-union* **by** *blast*
moreover have L': *distinct-mset* L'
 using ‹*distinct-mset* $(L + L')$› *distinct-mset-union* **unfolding** *add.commute*[*of L L'*] **by** *blast*
moreover have $L \mathbin{\#\cap} L' = \{\#\}$
 using L L' ‹*?A*› **unfolding** *multiset-inter-def multiset-eq-iff distinct-mset-count-less-1*
 by (*metis Nat.diff-le-self add-diff-cancel-left' count-diff count-empty diff-is-0-eq eq-iff le-neq-implies-less less-one*)
ultimately show *?B* **by** *fast*
next
assume *?B*
show *?A*
 unfolding *distinct-mset-count-less-1*

proof (*intro allI*)
  fix *a*
  have *count* $(L + L')$ *a* $\leq$ *count L a* + *count* L' *a*
    by *auto*
  moreover have *count L a* + *count* L' *a* $\leq$ *1*
    using ‹*?B*› **by** (*metis One-nat-def add.commute add-decreasing2 count-diff diff-add-zero distinct-mset-count-less-1 le-SucE multiset-inter-count plus-multiset.rep-eq subset-mset.inf.idem*)
  ultimately show *count* $(L + L')$ *a* $\leq$ *1*
    by *arith*
qed
qed

lemma *distinct-mset-set-mset-ident*[*simp*]: *distinct-mset M* $\Longrightarrow$ *mset-set* (*set-mset M*) = *M*
  apply (*auto simp*: *multiset-eq-iff*)
  apply (*rename-tac x*)
  apply (*case-tac count M x* = *0*)
   apply (*simp add*: *not-in-iff*[*symmetric*])
  apply (*case-tac count M x* = *1*)
   apply (*simp add*: *count-inI*)
  unfolding *distinct-mset-count-less-1* **by** (*meson le-neq-implies-less less-one*)

lemma *distinct-finite-set-mset-subseteq-iff*[*iff*]:
  assumes *dist*: *distinct-mset M* **and** *fin*: *finite N*
  shows *set-mset M* $\subseteq$ *N* $\longleftrightarrow$ *M* $\subseteq$# *mset-set N*
proof
  assume *set-mset M* $\subseteq$ *N*
  then show *M* $\subseteq$# *mset-set N*
    by (*metis dist distinct-mset-set-mset-ident fin finite-subset mset-set-subseteq-mset-set*)
next
  assume *M* $\subseteq$# *mset-set N*
  then show *set-mset M* $\subseteq$ *N*
    by (*metis contra-subsetD empty-iff finite-set-mset-mset-set infinite-set-mset-mset-set set-mset-mono subsetI*)
qed

lemma *distinct-mem-diff-mset*:
  assumes *dist*: *distinct-mset M* **and** *mem*: *x* $\in$ *set-mset* (*M* − *N*)
  shows *x* $\notin$ *set-mset N*
proof −
  have *count M x* = *1*
    using *dist mem* **by** (*meson distinct-mset-def in-diffD*)
  then show *?thesis*
    using *mem* **by** (*metis count-greater-eq-one-iff in-diff-count not-less*)
qed

lemma *distinct-set-mset-eq*:
  assumes
    dist-m: *distinct-mset M* **and**
    dist-n: *distinct-mset N* **and**
    set-eq: *set-mset M* = *set-mset N*
  shows *M* = *N*
proof −
  have *mset-set* (*set-mset M*) = *mset-set* (*set-mset N*)
    using *set-eq* **by** *simp*
  thus *?thesis*

using *dist-m dist-n* **by** *auto*
qed

lemma *distinct-mset-union-mset*:
assumes
distinct-mset D **and**
distinct-mset C
shows *distinct-mset* $(D \ \#\cup \ C)$
using *assms* **unfolding** *distinct-mset-count-less-1* **by** *force*

lemma *distinct-mset-inter-mset*:
assumes
distinct-mset D **and**
distinct-mset C
shows *distinct-mset* $(D \ \#\cap \ C)$
using *assms* **unfolding** *distinct-mset-count-less-1*
by (*meson dual-order.trans subset-mset.inf-le2 subseteq-mset-def*)

lemma *distinct-mset-remove1-All*:
distinct-mset C $\Longrightarrow$ *remove1-mset L C* = *removeAll-mset L C*
by (*auto simp*: *multiset-eq-iff distinct-mset-count-less-1*)

lemma *distinct-mset-size-2*: *distinct-mset* $\{\#a, \ b\#\} \longleftrightarrow a \neq b$
unfolding *distinct-mset-def* **by** *auto*

1.1.9 Filter

lemma *mset-filter-compl*: *mset* (*filter p xs*) + *mset* (*filter* (*Not* $\circ$ *p*) *xs*) = *mset xs*
apply (*induct xs*)
by *simp*
(*metis* (*no-types*) *add-diff-cancel-left′ comp-apply filter.simps*(*2*) *mset.simps*(*2*)
mset-compl-union)

lemma *image-mset-subseteq-mono*: $A \subseteq\# \ B \Longrightarrow$ *image-mset f A* $\subseteq\#$ *image-mset f B*
by (*metis image-mset-union subset-mset.le-iff-add*)

lemma *image-filter-ne-mset*[*simp*]:
image-mset f $\{\#x \in\# \ M. \ f \ x \neq y\#\}$ = *removeAll-mset y* (*image-mset f M*)
by (*induct M*, *auto*, *meson count-le-replicate-mset-le order-refl subset-mset.add-diff-assoc2*)

lemma *comprehension-mset-False*[*simp*]:
$\{\# \ L \in\# \ A. \ False\#\} = \{\#\}$
by (*auto simp*: *multiset-eq-iff*)

Near duplicate of *filter-eq-replicate-mset*: $\{\# \ y \in\# \ ?D. \ y = \ ?x\#\}$ = *replicate-mset* (*count ?D ?x*) *?x*.

lemma *filter-mset-eq*:
filter-mset (*op* = *L*) *A* = *replicate-mset* (*count A L*) *L*
by (*auto simp*: *multiset-eq-iff*)

lemma *filter-mset-union-mset*:
filter-mset P $(A \ \#\cup \ B)$ = *filter-mset P A* $\#\cup$ *filter-mset P B*
by (*auto simp*: *multiset-eq-iff*)

lemma *filter-mset-mset-set*:
finite A $\Longrightarrow$ *filter-mset P* (*mset-set A*) = *mset-set* $\{a \in A. \ P \ a\}$

by (*auto simp*: *multiset-eq-iff count-mset-set-if*)

See *filter-cong* for the set version. Mark as [*fundef-cong*] too?

lemma *filter-mset-cong*:
assumes [*simp*]: $M = M'$ **and** [*simp*]: $\bigwedge a.\ a \in\# M \Longrightarrow P\ a = Q\ a$
shows *filter-mset* $P\ M$ = *filter-mset* $Q\ M$
proof −
have M − *filter-mset* $Q\ M$ = *filter-mset* $(\lambda a.\ \neg Q\ a)\ M$
by (*subst multiset-partition*[*of - Q*]) *simp*
then show *?thesis*
by (*auto simp*: *filter-mset-eq-conv*)
qed

1.1.10 Sums

lemma *msetsum-distrib*[*simp*]:
fixes $C\ D :: {'a} \Rightarrow {'b}::\{comm\text{-}monoid\text{-}add\}$
shows $(\sum x{\in}\#A.\ C\ x + D\ x) = (\sum x{\in}\#A.\ C\ x) + (\sum x{\in}\#A.\ D\ x)$
by (*induction A*) (*auto simp*: *ac-simps*)

lemma *msetsum-union-disjoint*:
assumes $A \#\cap B = \{\#\}$
shows $(\sum La{\in}\#A \#\cup B.\ f\ La) = (\sum La{\in}\#A.\ f\ La) + (\sum La{\in}\#B.\ f\ La)$
by (*metis assms diff-zero empty-sup image-mset-union msetsum.union multiset-inter-commute multiset-union-diff-commute sup-subset-mset-def zero-diff*)

1.1.11 Order

Instantiating multiset order as a linear order.

TODO: remove when multiset is of sort ord again

instantiation *multiset* :: (*linorder*) *linorder*
begin

definition *less-multiset* :: *'a::linorder multiset* ⇒ *'a multiset* ⇒ *bool* **where**
$M' < M \longleftrightarrow M' \#{\subset}\# M$

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒*bool* **where**
$(M'::'a\ multiset) \le M \longleftrightarrow M' \#{\subseteq}\# M$

instance
by *standard* (*auto simp add*: *less-eq-multiset-def less-multiset-def multiset-order.less-le-not-le add.commute multiset-order.add-right-mono*)
end
end

1.2 Transitions

This theory contains some facts about closure, the definition of full transformations, and well-foundedness.

theory *Wellfounded-More*
imports *Main*

begin

1.2.1 More theorems about Closures

This is the equivalent of the theorem *rtranclp-mono* for *tranclp*

lemma *tranclp-mono-explicit*:
 $r^{++}\ a\ b \Longrightarrow r \leq s \Longrightarrow s^{++}\ a\ b$
  using *rtranclp-mono* **by** (*auto dest*!: *tranclpD intro*: *rtranclp-into-tranclp2*)

lemma *tranclp-mono*:
 assumes *mono*: $r \leq s$
 shows $r^{++} \leq s^{++}$
  using *rtranclp-mono*[*OF mono*] *mono* **by** (*auto dest*!: *tranclpD intro*: *rtranclp-into-tranclp2*)

lemma *tranclp-idemp-rel*:
 $R^{++++}\ a\ b \longleftrightarrow R^{++}\ a\ b$
 apply (*rule iffI*)
  prefer *2* **apply** *blast*
 by (*induction rule*: *tranclp-induct*) *auto*

Equivalent of the theorem *rtranclp-idemp*

lemma *trancl-idemp*: $(r^+)^+ = r^+$
 by *simp*

lemmas *tranclp-idemp*[*simp*] = *trancl-idemp*[*to-pred*]

This theorem already exists as theroem *Nitpick.rtranclp-unfold* (and sledgehammer uses it), but it makes sense to duplicate it, because it is unclear how stable the lemmas in the `~~/src/HOL/Nitpick.thy` theory are.

lemma *rtranclp-unfold*: *rtranclp r a b* $\longleftrightarrow$ ($a = b \vee$ *tranclp r a b*)
 by (*meson rtranclp.simps rtranclpD tranclp-into-rtranclp*)

lemma *tranclp-unfold-end*: *tranclp r a b* $\longleftrightarrow$ ($\exists\, a'$. *rtranclp r a* $a' \wedge r\ a'\ b$)
 by (*metis rtranclp.rtrancl-refl rtranclp-into-tranclp1 tranclp.cases tranclp-into-rtranclp*)

Near duplicate of theorem *tranclpD*:

lemma *tranclp-unfold-begin*: *tranclp r a b* $\longleftrightarrow$ ($\exists\, a'$. $r\ a\ a' \wedge$ *rtranclp r* $a'\ b$)
 by (*meson rtranclp-into-tranclp2 tranclpD*)

lemma *trancl-set-tranclp*: $(a, b) \in \{(b,a).\ P\ a\ b\}^+ \longleftrightarrow P^{++}\ b\ a$
 apply (*rule iffI*)
  apply (*induction rule*: *trancl-induct*; *simp*)
 apply (*induction rule*: *tranclp-induct*; *auto simp*: *trancl-into-trancl2*)
 done

lemma *tranclp-rtranclp-rtranclp-rel*: $R^{++**}\ a\ b \longleftrightarrow R^{**}\ a\ b$
 by (*simp add*: *rtranclp-unfold*)

lemma *tranclp-rtranclp-rtranclp*[*simp*]: $R^{++**} = R^{**}$
 by (*fastforce simp*: *rtranclp-unfold*)

lemma *rtranclp-exists-last-with-prop*:
 assumes $R\ x\ z$ **and** $R^{**}\ z\ z'$ **and** $P\ x\ z$
 shows $\exists\, y\ y'.\ R^{**}\ x\ y \wedge R\ y\ y' \wedge P\ y\ y' \wedge (\lambda a\ b.\ R\ a\ b \wedge \neg P\ a\ b)^{**}\ y'\ z'$
 using *assms*(*2*,*1*,*3*)
proof (*induction*)

case *base*
then show *?case* **by** *auto*
next
case (*step* z' z'') **note** $z = this(2)$ **and** $IH = this(3)[OF\ this(4-5)]$
show *?case*
apply (*cases* P z' z'')
apply (*rule exI*[*of* - z'], *rule exI*[*of* - z''])
using *z assms*(*1*) *step.hyps*(*1*) *step.prems*(*2*) **apply** *auto*[*1*]
using *IH z rtranclp.rtrancl-into-rtrancl* **by** *fastforce*
qed

lemma *rtranclp-and-rtranclp-left*: $(\lambda\ a\ b.\ P\ a\ b \wedge Q\ a\ b)^{**}\ S\ T \Longrightarrow P^{**}\ S\ T$
by (*induction rule*: *rtranclp-induct*) *auto*

1.2.2 Full Transitions

We define here properties to define properties after all possible transitions.

abbreviation *no-step step* $S \equiv (\forall S'.\ \neg step\ S\ S')$

definition *full1* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
full1 transf $= (\lambda S\ S'.\ tranclp\ transf\ S\ S' \wedge (\forall S''.\ \neg\ transf\ S'\ S''))$

definition *full*:: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
full transf $= (\lambda S\ S'.\ rtranclp\ transf\ S\ S' \wedge (\forall S''.\ \neg\ transf\ S'\ S''))$

We define output notations only for printing:

notation (**output**) *full1* ($-^{+\downarrow}$)
notation (**output**) *full* ($-^{\downarrow}$)

lemma *rtranclp-full1I*:
$R^{**}\ a\ b \Longrightarrow full1\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
unfolding *full1-def* **by** *auto*

lemma *tranclp-full1I*:
$R^{++}\ a\ b \Longrightarrow full1\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
unfolding *full1-def* **by** *auto*

lemma *rtranclp-fullI*:
$R^{**}\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full\ R\ a\ c$
unfolding *full-def* **by** *auto*

lemma *tranclp-full-full1I*:
$R^{++}\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
unfolding *full-def full1-def* **by** *auto*

lemma *full-fullI*:
$R\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
unfolding *full-def full1-def* **by** *auto*

lemma *full-unfold*:
$full\ r\ S\ S' \longleftrightarrow ((S = S' \wedge no\text{-}step\ r\ S') \vee full1\ r\ S\ S')$
unfolding *full-def full1-def* **by** (*auto simp add*: *rtranclp-unfold*)

lemma *full1-is-full*[*intro*]: $full1\ R\ S\ T \Longrightarrow full\ R\ S\ T$
by (*simp add*: *full-unfold*)

lemma *not-full1-rtranclp-relation*: $\neg$*full1* R^{**} *a b*
by (*meson full1-def rtranclp.rtrancl-refl*)

lemma *not-full-rtranclp-relation*: $\neg$*full* R^{**} *a b*
by (*meson full-fullI not-full1-rtranclp-relation rtranclp.rtrancl-refl*)

lemma *full1-tranclp-relation-full*:
full1 R^{++} *a b* $\longleftrightarrow$ *full1 R a b*
by (*metis converse-tranclpE full1-def reflclp-tranclp rtranclpD rtranclp-idemp rtranclp-reflclp tranclp.r-into-trancl tranclp-into-rtranclp*)

lemma *full-tranclp-relation-full*:
full R^{++} *a b* $\longleftrightarrow$ *full R a b*
by (*metis full-unfold full1-tranclp-relation-full tranclp.r-into-trancl tranclpD*)

lemma *rtranclp-full1-eq-or-full1*:
$(full1\ R)^{**}\ a\ b \longleftrightarrow (a = b \lor full1\ R\ a\ b)$
proof $-$
have $\forall p\ a\ aa.\ \neg\ p^{**}\ (a::'a)\ aa \lor a = aa \lor (\exists ab.\ p^{**}\ a\ ab \land p\ ab\ aa)$
by (*metis rtranclp.cases*)
then obtain $aa :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
f1: $\forall p\ a\ ab.\ \neg\ p^{**}\ a\ ab \lor a = ab \lor p^{**}\ a\ (aa\ p\ a\ ab) \land p\ (aa\ p\ a\ ab)\ ab$
by *moura*
{ assume $a \neq b$
{ assume $\neg$ *full1 R a b* $\land\ a \neq b$
then have $a \neq b \land a \neq b \land \neg\ full1\ R\ (aa\ (full1\ R)\ a\ b)\ b \lor \neg\ (full1\ R)^{**}\ a\ b \land a \neq b$
using *f1* **by** (*metis* (*no-types*) *full1-def full1-tranclp-relation-full*)
then have *?thesis*
using *f1* **by** *blast* **}**
then have *?thesis*
by *auto* **}**
then show *?thesis*
by *fastforce*
qed

lemma *tranclp-full1-full1*:
$(full1\ R)^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b$
by (*metis full1-def rtranclp-full1-eq-or-full1 tranclp-unfold-begin*)

1.2.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:
assumes *wf*:*wf* $\{(x, y).\ R\ y\ x\}$
shows $\exists b.\ R^{**}\ a\ b \land$ *no-step R b*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then have *H*: $\bigwedge b.\ \neg\ R^{**}\ a\ b \lor \neg$*no-step R b*
by *blast*
def $F \equiv$ *rec-nat a* ($\lambda i\ b.$ *SOME c. R b c*)
have [*simp*]: $F\ 0 = a$
unfolding *F-def* **by** *auto*
have [*simp*]: $\bigwedge i.\ F\ (Suc\ i) = (SOME\ b.\ R\ (F\ i)\ b)$
using *F-def* **by** *simp*
{ fix *i*
have $\forall j<i.\ R\ (F\ j)\ (F\ (Suc\ j))$

```
    proof (induction i)
      case 0
      then show ?case by auto
    next
      case (Suc i)
      then have R** a (F i)
        by (induction i) auto
      then have R (F i) (SOME b. R (F i) b)
        using H by (simp add: someI-ex)
      then have ∀ j < Suc i. R (F j) (F (Suc j))
        using H Suc by (simp add: less-Suc-eq)
      then show ?case by fast
    qed
  }
  then have ∀ j. R (F j) (F (Suc j)) by blast
  then show False
    using wf unfolding wfP-def wf-iff-no-infinite-down-chain by blast
qed

lemma wf-exists-normal-form-full:
  assumes wf: wf {(x, y). R y x}
  shows ∃ b. full R a b
  using wf-exists-normal-form[OF assms] unfolding full-def by blast
```

1.2.4 More Well-Foundedness

A little list of theorems that could be useful, but are hidden:

- link between *wf* and infinite chains: theorems *wf-iff-no-infinite-down-chain* and *wf-no-infinite-down-chainI*

```
lemma wf-if-measure-in-wf:
  wf R ⟹ (⋀a b. (a, b) ∈ S ⟹ (ν a, ν b)∈R) ⟹ wf S
  by (metis in-inv-image wfE-min wfI-min wf-inv-image)

lemma wfP-if-measure: fixes f :: 'a ⇒ nat
  shows (⋀x y. P x ⟹ g x y ⟹ f y < f x) ⟹ wf {(y,x). P x ∧ g x y}
  apply(insert wf-measure[of f])
  apply(simp only: measure-def inv-image-def less-than-def less-eq)
  apply(erule wf-subset)
  apply auto
  done

lemma wf-if-measure-f:
  assumes wf r
  shows wf {(b, a). (f b, f a) ∈ r}
  using assms by (metis inv-image-def wf-inv-image)

lemma wf-wf-if-measure':
  assumes wf r and H: ⋀x y. P x ⟹ g x y ⟹ (f y, f x) ∈ r
  shows wf {(y,x). P x ∧ g x y}
proof –
  have wf {(b, a). (f b, f a) ∈ r} using assms(1) wf-if-measure-f by auto
  then have wf {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r}
    using wf-subset[of - {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r}] by auto
  moreover have {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r} ⊆ {(b, a). (f b, f a) ∈ r} by auto
```

moreover have $\{(b, a).\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ a) \in r\} = \{(b, a).\ P\ a \wedge g\ a\ b\}$ **using** *H* **by** *auto*
ultimately show *?thesis* **using** *wf-subset* **by** *simp*
qed

lemma *wf-lex-less*: $wf\ (lex\ \{(a, b).\ (a{::}nat) < b\})$
proof −
have *m*: $\{(a, b).\ a < b\} = measure\ id$ **by** *auto*
show *?thesis* **apply** (*rule wf-lex*) **unfolding** *m* **by** *auto*
qed

lemma *wfP-if-measure2*: **fixes** $f :: 'a \Rightarrow nat$
shows $(\bigwedge x\ y.\ P\ x\ y \Longrightarrow g\ x\ y \Longrightarrow f\ x < f\ y) \Longrightarrow wf\ \{(x,y).\ P\ x\ y \wedge g\ x\ y\}$
apply(*insert wf-measure*[*of f*])
apply(*simp only*: *measure-def inv-image-def less-than-def less-eq*)
apply(*erule wf-subset*)
apply *auto*
done

lemma *lexord-on-finite-set-is-wf*:
assumes
P-finite: $\bigwedge U.\ P\ U \longrightarrow U \in A$ **and**
finite: *finite A* **and**
wf: *wf R* **and**
trans: *trans R*
shows $wf\ \{(T, S).\ (P\ S \wedge P\ T) \wedge (T, S) \in lexord\ R\}$
proof (*rule wfP-if-measure2*)
fix *T S*
assume *P*: $P\ S \wedge P\ T$ **and**
s-le-t: $(T, S) \in lexord\ R$
let $?f = \lambda S.\ \{U.\ (U, S) \in lexord\ R \wedge P\ U \wedge P\ S\}$
have $?f\ T \subseteq ?f\ S$
using *s-le-t P lexord-trans trans* **by** *auto*
moreover have $T \in ?f\ S$
using *s-le-t P* **by** *auto*
moreover have $T \notin ?f\ T$
using *s-le-t* **by** (*auto simp add*: *lexord-irreflexive local.wf*)
ultimately have $\{U.\ (U, T) \in lexord\ R \wedge P\ U \wedge P\ T\} \subset \{U.\ (U, S) \in lexord\ R \wedge P\ U \wedge P\ S\}$
by *auto*
moreover have $finite\ \{U.\ (U, S) \in lexord\ R \wedge P\ U \wedge P\ S\}$
using *finite* **by** (*metis* (*no-types*, *lifting*) *P-finite finite-subset mem-Collect-eq subsetI*)
ultimately show $card\ (?f\ T) < card\ (?f\ S)$ **by** (*simp add*: *psubset-card-mono*)
qed

lemma *wf-fst-wf-pair*:
assumes $wf\ \{(M', M).\ R\ M'\ M\}$
shows $wf\ \{((M', N'), (M, N)).\ R\ M'\ M\}$
proof −
have $wf\ (\{(M', M).\ R\ M'\ M\}\ {<}{*}lex{*}{>}\ \{\})$
using *assms* **by** *auto*
then show *?thesis*
by (*rule wf-subset*) *auto*
qed

lemma *wf-snd-wf-pair*:
assumes $wf\ \{(M', M).\ R\ M'\ M\}$

shows *wf* $\{((M', N'), (M, N)).\ R\ N'\ N\}$
proof −
have *wf*: *wf* $\{((M', N'), (M, N)).\ R\ M'\ M\}$
using *assms wf-fst-wf-pair* **by** *auto*
then have *wf*: $\bigwedge P.\ (\forall x.\ (\forall y.\ (y, x) \in \{((M', N'), M, N).\ R\ M'\ M\} \longrightarrow P\ y) \longrightarrow P\ x) \Longrightarrow All\ P$
unfolding *wf-def* **by** *auto*
show *?thesis*
unfolding *wf-def*
proof (*intro allI impI*)
fix $P :: 'c \times 'a \Rightarrow bool$ **and** $x :: 'c \times 'a$
assume H: $\forall x.\ (\forall y.\ (y, x) \in \{((M', N'), M, y).\ R\ N'\ y\} \longrightarrow P\ y) \longrightarrow P\ x$
obtain $a\ b$ **where** x: $x = (a, b)$ **by** (*cases x*)
have P: $P\ x = (P \circ (\lambda(a, b).\ (b, a)))\ (b, a)$
unfolding x **by** *auto*
show $P\ x$
using *wf*[*of P o* $(\lambda(a, b).\ (b, a))$] **apply** *rule*
using H **apply** *simp*
unfolding P **by** *blast*
qed
qed

lemma *wf-if-measure-f-notation2*:
assumes *wf r*
shows *wf* $\{(b, h\ a)|b\ a.\ (f\ b, f\ (h\ a)) \in r\}$
apply (*rule wf-subset*)
using *wf-if-measure-f*[*OF assms*, *of f*] **by** *auto*

lemma *wf-wf-if-measure'-notation2*:
assumes *wf r* **and** H: $\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y, f\ (h\ x)) \in r$
shows *wf* $\{(y,h\ x)|\ y\ x.\ P\ x \wedge g\ x\ y\}$
proof −
have *wf* $\{(b, h\ a)|b\ a.\ (f\ b, f\ (h\ a)) \in r\}$ **using** *assms*(*1*) *wf-if-measure-f-notation2* **by** *auto*
then have *wf* $\{(b, h\ a)|b\ a.\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ (h\ a)) \in r\}$
using *wf-subset*[*of* - $\{(b, h\ a)|\ b\ a.\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ (h\ a)) \in r\}$] **by** *auto*
moreover have $\{(b, h\ a)|b\ a.\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ (h\ a)) \in r\}$
$\subseteq \{(b, h\ a)|b\ a.\ (f\ b, f\ (h\ a)) \in r\}$ **by** *auto*
moreover have $\{(b, h\ a)|b\ a.\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ (h\ a)) \in r\} = \{(b, h\ a)|b\ a.\ P\ a \wedge g\ a\ b\}$
using H **by** *auto*
ultimately show *?thesis* **using** *wf-subset* **by** *simp*
qed

end
theory *List-More*
imports *Main ../lib/Multiset-More*
begin

Sledgehammer parameters

sledgehammer-params[*debug*]

1.3 Various Lemmas

Close to the theorem *nat-less-induct* $((\bigwedge n.\ \forall m<n.\ ?P\ m \Longrightarrow ?P\ n) \Longrightarrow ?P\ ?n)$, but with a separation between the zero and non-zero case.

thm *nat-less-induct*

lemma *nat-less-induct-case*[*case-names 0 Suc*]:
  assumes
    P 0 **and**
    $\bigwedge n.$ $(\forall m < Suc\ n.\ P\ m) \Longrightarrow P\ (Suc\ n)$
  shows *P n*
  apply (*induction rule*: *nat-less-induct*)
  by (*rename-tac n*, *case-tac n*) (*auto intro*: *assms*)

This is only proved in simple cases by auto. In assumptions, nothing happens, and the theorem *if-split-asm* can blow up goals (because of other if-expressions either in the context or as simplification rules).

lemma *if-0-1-ge-0*[*simp*]:
  $0 < (\mathit{if}\ P\ \mathit{then}\ a\ \mathit{else}\ (0{::}nat)) \longleftrightarrow P \wedge 0 < a$
  by *auto*

Bounded function have not yet been defined in Isabelle.

definition *bounded* **where**
$bounded\ f \longleftrightarrow (\exists b.\ \forall n.\ f\ n \le b)$

abbreviation *unbounded* :: $('a \Rightarrow 'b{::}ord) \Rightarrow bool$ **where**
$unbounded\ f \equiv \neg\ bounded\ f$

lemma *not-bounded-nat-exists-larger*:
  fixes $f :: nat \Rightarrow nat$
  assumes *unbound*: *unbounded f*
  shows $\exists n.\ f\ n > m \wedge n > n_0$
proof (*rule ccontr*)
  assume *H*: ¬ *?thesis*
  have *finite* $\{f\ n | n.\ n \le n_0\}$
    by *auto*
  have $\bigwedge n.\ f\ n \le Max\ (\{f\ n|n.\ n \le n_0\} \cup \{m\})$
    apply ($case\text{-}tac\ n \le n_0$)
    apply (*metis* (*mono-tags*, *lifting*) *Max-ge Un-insert-right* ‹*finite* $\{f\ n\ |n.\ n \le n_0\}$›
      finite-insert insertCI mem-Collect-eq sup-bot.right-neutral)
    by (*metis* (*no-types*, *lifting*) *H Max-less-iff Un-insert-right* ‹*finite* $\{f\ n\ |n.\ n \le n_0\}$›
      finite-insert insertI1 insert-not-empty leI sup-bot.right-neutral)
  then show *False*
    using *unbound* **unfolding** *bounded-def* **by** *auto*
qed

A function is bounded iff its product with a non-zero constant is bounded. The non-zero condition is needed only for the reverse implication (see for example $k = 0$ and $f = (\lambda i.\ i)$ for a counter-example).

lemma *bounded-const-product*:
  fixes $k :: nat$ **and** $f :: nat \Rightarrow nat$
  assumes $k > 0$
  shows $bounded\ f \longleftrightarrow bounded\ (\lambda i.\ k * f\ i)$
  unfolding *bounded-def* **apply** (*rule iffI*)
   using *mult-le-mono2* **apply** *blast*
  by (*meson assms le-less-trans less-or-eq-imp-le nat-mult-less-cancel-disj split-div-lemma*)

This lemma is not used, but here to show that property that can be expected from *bounded* holds.

lemma *bounded-finite-linorder*:

fixes $f :: 'a \Rightarrow 'a :: \{finite,\ linorder\}$
shows *bounded f*
proof −
have $\bigwedge x.\ f\ x \leq Max\ \{f\ x|x.\ True\}$
by (*metis* (*mono-tags*) *Max-ge finite mem-Collect-eq*)
then show *?thesis*
unfolding *bounded-def* **by** *blast*
qed

1.4 More List

1.4.1 *upt*

The simplification rules are not very handy, because theorem *upt.simps* (*2*) (i.e. $[?i..<Suc\ ?j] = (if\ ?i \leq ?j\ then\ [?i..<?j]\ @\ [?j]\ else\ [])$) leads to a case distinction, that we do not want if the condition is not in the context.

lemma *upt-Suc-le-append*: $\neg i \leq j \Longrightarrow [i..<Suc\ j] = []$
by *auto*

lemmas *upt-simps*[*simp*] = *upt-Suc-append upt-Suc-le-append*

declare *upt.simps*(*2*)[*simp del*]

The counterpart for this lemma when $n - m < i$ is theorem *take-all*. It is close to theorem $?i + ?m \leq ?n \Longrightarrow take\ ?m\ [?i..<?n] = [?i..<?i + ?m]$, but seems more general.

lemma *take-upt-bound-minus*[*simp*]:
assumes $i \leq n - m$
shows $take\ i\ [m..<n] = [m\ ..<m+i]$
using *assms* **by** (*induction i*) *auto*

lemma *append-cons-eq-upt*:
assumes $A\ @\ B = [m..<n]$
shows $A = [m\ ..<m+length\ A]$ **and** $B = [m + length\ A..<n]$
proof −
have $take\ (length\ A)\ (A\ @\ B) = A$ **by** *auto*
moreover
have $length\ A \leq n - m$ **using** *assms linear calculation* **by** *fastforce*
then have $take\ (length\ A)\ [m..<n] = [m\ ..<m+length\ A]$ **by** *auto*
ultimately show $A = [m\ ..<m+length\ A]$ **using** *assms* **by** *auto*
show $B = [m + length\ A..<n]$ **using** *assms* **by** (*metis append-eq-conv-conj drop-upt*)
qed

The converse of theorem *append-cons-eq-upt* does not hold, for example if @ term "B:: nat list" is empty and A is $[0::'a]$:

lemma $A\ @\ B = [m..<\ n] \longleftrightarrow A = [m\ ..<m+length\ A] \wedge B = [m + length\ A..<n]$

oops

A more restrictive version holds:

lemma $B \neq [] \Longrightarrow A\ @\ B = [m..<\ n] \longleftrightarrow A = [m\ ..<m+length\ A] \wedge B = [m + length\ A..<n]$
(**is** $?P \Longrightarrow ?A = ?B$)
proof
assume *?A* **then show** *?B* **by** (*auto simp add*: *append-cons-eq-upt*)

next
  assume *?P* **and** *?B*
  then show *?A* **using** *append-eq-conv-conj* **by** *fastforce*
qed

lemma *append-cons-eq-upt-length-i*:
  assumes $A @ i \# B = [m..<n]$
  shows $A = [m ..<i]$
proof −
  have $A = [m ..< m + length\ A]$ **using** *assms append-cons-eq-upt* **by** *auto*
  have $(A @ i \# B) ! (length\ A) = i$ **by** *auto*
  moreover have $n - m = length\ (A @ i \# B)$
    using *assms length-upt* **by** *presburger*
  then have $[m..<n] ! (length\ A) = m + length\ A$ **by** *simp*
  ultimately have $i = m + length\ A$ **using** *assms* **by** *auto*
  then show *?thesis* **using** ‹$A = [m ..< m + length\ A]$› **by** *auto*
qed

lemma *append-cons-eq-upt-length*:
  assumes $A @ i \# B = [m..<n]$
  shows $length\ A = i - m$
  using *assms*
proof (*induction A arbitrary*: *m*)
  case *Nil*
  then show *?case* **by** (*metis append-Nil diff-is-0-eq list.size*(*3*) *order-refl upt-eq-Cons-conv*)
next
  case (*Cons a A*)
  then have *A*: $A @ i \# B = [m + 1..<n]$ **by** (*metis append-Cons upt-eq-Cons-conv*)
  then have $m < i$ **by** (*metis Cons.prems append-cons-eq-upt-length-i upt-eq-Cons-conv*)
  with *Cons.IH*[*OF A*] **show** *?case* **by** *auto*
qed

lemma *append-cons-eq-upt-length-i-end*:
  assumes $A @ i \# B = [m..<n]$
  shows $B = [Suc\ i ..<n]$
proof −
  have $B = [Suc\ m + length\ A..<n]$ **using** *assms append-cons-eq-upt*[*of* $A @ [i]\ B\ m\ n$] **by** *auto*
  have $(A @ i \# B) ! (length\ A) = i$ **by** *auto*
  moreover have $n - m = length\ (A @ i \# B)$
    using *assms length-upt* **by** *auto*
  then have $[m..<n]! (length\ A) = m + length\ A$ **by** *simp*
  ultimately have $i = m + length\ A$ **using** *assms* **by** *auto*
  then show *?thesis* **using** ‹$B = [Suc\ m + length\ A..<n]$› **by** *auto*
qed

lemma *Max-n-upt*: $Max\ (insert\ 0\ \{Suc\ 0..<n\}) = n - Suc\ 0$
proof (*induct n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*) **note** $IH = this$
  have *i*: $insert\ 0\ \{Suc\ 0..<Suc\ n\} = insert\ 0\ \{Suc\ 0..< n\} \cup \{n\}$ **by** *auto*
  show *?case* **using** *IH* **unfolding** *i* **by** *auto*
qed

lemma *upt-decomp-lt*:

assumes *H*: *xs* @ *i* # *ys* @ *j* # *zs* = [*m* ..< *n*]
shows $i < j$
proof −
have *xs*: *xs* = [*m* ..< *i*] **and** *ys*: *ys* = [*Suc i* ..< *j*] **and** *zs*: *zs* = [*Suc j* ..< *n*]
using *H* **by** (*auto dest*: *append-cons-eq-upt-length-i append-cons-eq-upt-length-i-end*)
show *?thesis*
by (*metis append-cons-eq-upt-length-i-end assms lessI less-trans self-append-conv2 upt-eq-Cons-conv upt-rec ys*)
qed

The following two lemmas are useful as simp rules for case-distinction. The case *length* $l = 0$ is already simplified by default.

lemma *length-list-Suc-0*:
length W = *Suc 0* ⟷ (∃ *L*. *W* = [*L*])
apply (*cases W*)
apply *simp*
apply (*rename-tac a W′*, *case-tac W′*)
apply *auto*
done

lemma *length-list-2*: *length S* = *2* ⟷ (∃ *a b*. *S* = [*a*, *b*])
apply (*cases S*)
apply *simp*
apply (*rename-tac a S′*)
apply (*case-tac S′*)
by *simp-all*

lemma *finite-bounded-list*:
fixes *b* :: *nat*
shows *finite* {*xs*. *length xs* < *s* ∧ (∀ *i*< *length xs*. *xs* ! *i* < *b*)} (**is** *finite* (*?S s*))
proof (*induction s*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc s*) **note** *IH* = *this*(*1*)
have *H*: *?S* (*Suc s*) ⊆ *?S s* ∪ {*x* # *xs*| *x xs*. *x* < *b* ∧ *length xs* < *s* ∧ (∀ *i*< *length xs*. *xs* ! *i* < *b*)} ∪ {[]}
(**is** - ⊆ - ∪ *?C* ∪ -)
proof
fix *xs*
assume *xs* ∈ *?S* (*Suc s*)
then have *B*: ∀ *i*<*length xs*. *xs* ! *i* < *b* **and** *len*: *length xs* < *Suc s*
by *auto*
consider
(*st*) *length xs* < *s* |
(*s*) *length xs* = *0* **and** *s* = *0* |
(*s′*) *s′* **where** *length xs* = *Suc s′*
using *len* **by** (*cases s*) (*auto simp add*: *Nat.less-Suc-eq*)
then show *xs* ∈ *?S s* ∪ *?C* ∪ {[]}
proof *cases*
case *st*
then show *?thesis* **using** *B* **by** *auto*
next
case *s*
then show *?thesis* **using** *B* **by** *auto*
next

case *s'* **note** *len-xs* = *this*(*1*)
then obtain *x xs'* **where** *xs*: *xs* = *x* # *xs'* **by** (*cases xs*) *auto*
then show *?thesis* **using** *len-xs B len s'* **unfolding** *xs* **by** *auto*
qed
qed
have *?C* ⊆ (*case-prod Cons*) ‘ ({*x*. *x* < *b*} × *?S s*)
by *auto*
moreover have *finite* ({*x*. *x* < *b*} × *?S s*)
using *IH* **by** (*auto simp*: *finite-cartesian-product-iff*)
ultimately have *finite ?C* **by** (*simp add*: *finite-surj*)
then have *finite* (*?S s* ∪ *?C* ∪ {[]})
using *IH* **by** *auto*
then show *?case* **using** *H* **by** (*auto intro*: *finite-subset*)
qed

1.4.2 Lexicographic Ordering

lemma *lexn-Suc*:
(*x* # *xs*, *y* # *ys*) ∈ *lexn r* (*Suc n*) ⟷
(*length xs* = *n* ∧ *length ys* = *n*) ∧ ((*x*, *y*) ∈ *r* ∨ (*x* = *y* ∧ (*xs*, *ys*) ∈ *lexn r n*))
by (*auto simp*: *map-prod-def image-iff lex-prod-def*)

lemma *lexn-n*:
n > *0* ⟹ (*x* # *xs*, *y* # *ys*) ∈ *lexn r n* ⟷
(*length xs* = *n*−*1* ∧ *length ys* = *n*−*1*) ∧ ((*x*, *y*) ∈ *r* ∨ (*x* = *y* ∧ (*xs*, *ys*) ∈ *lexn r* (*n* − *1*)))
apply (*cases n*)
apply *simp*
by (*auto simp*: *map-prod-def image-iff lex-prod-def*)

There is some subtle point in the proof here. *1* is converted to *Suc 0*, but *2* is not: meaning that *1* is automatically simplified by default using the default simplification rule *lexn.simps*. However, the latter needs additional simplification rule (see the proof of the theorem above).

lemma *lexn2-conv*:
([*a*, *b*], [*c*, *d*]) ∈ *lexn r 2* ⟷ (*a*, *c*) ∈ *r* ∨ (*a* = *c* ∧ (*b*, *d*) ∈*r*)
by (*auto simp*: *lexn-n simp del*: *lexn.simps*(*2*))

lemma *lexn3-conv*:
([*a*, *b*, *c*], [*a'*, *b'*, *c'*]) ∈ *lexn r 3* ⟷
(*a*, *a'*) ∈ *r* ∨ (*a* = *a'* ∧ (*b*, *b'*) ∈ *r*) ∨ (*a* = *a'* ∧ *b* = *b'* ∧ (*c*, *c'*) ∈ *r*)
by (*auto simp*: *lexn-n simp del*: *lexn.simps*(*2*))

1.4.3 Remove

More lemmas about remove

lemma *remove1-Nil*:
remove1 (− *L*) *W* = [] ⟷ (*W* = [] ∨ *W* = [−*L*])
by (*cases W*) *auto*

lemma *remove1-mset-single-add*:
a ≠ *b* ⟹ *remove1-mset a* ({#*b*#} + *C*) = {#*b*#} + *remove1-mset a C*
remove1-mset a ({#*a*#} + *C*) = *C*
by (*auto simp*: *multiset-eq-iff*)

Remove under condition

This function removes the first element such that the condition f holds. It generalises *remove1*.

fun *remove1-cond* **where**
remove1-cond f $[] = []$ |
remove1-cond f $(C' \# L) = ($*if f* C' *then* L *else* $C' \#$ *remove1-cond f L*$)$

lemma *remove1 x xs = remove1-cond ((op =) x) xs*
  by (*induction xs*) *auto*

lemma *mset-map-mset-remove1-cond*:
  mset (*map mset* (*remove1-cond* (λL. *mset L = mset a*) *C*)) =
    remove1-mset (*mset a*) (*mset* (*map mset C*))
  by (*induction C*) (*auto simp*: *ac-simps remove1-mset-single-add*)

We can also generalise *removeAll*, which is close to *filter*:

fun *removeAll-cond* **where**
removeAll-cond f $[] = []$ |
removeAll-cond f $(C' \# L) =$
  (*if f* C' *then removeAll-cond f L else* $C' \#$ *removeAll-cond f L*)

lemma *removeAll x xs = removeAll-cond ((op =) x) xs*
  by (*induction xs*) *auto*

lemma *removeAll-cond P xs = filter* ($\lambda x. \neg P\ x$) *xs*
  by (*induction xs*) *auto*

lemma *mset-map-mset-removeAll-cond*:
  mset (*map mset* (*removeAll-cond* (λb. *mset b = mset a*) *C*))
  = *removeAll-mset* (*mset a*) (*mset* (*map mset C*))
  by (*induction C*) (*auto simp*: *ac-simps mset-less-eqI multiset-diff-union-assoc*)

The definition and the correctness theorem are from the multiset theory `~~/src/HOL/Library/Multiset.thy`, but a name is necessary to refer to them:

abbreviation *union-mset-list* **where**
union-mset-list xs ys $\equiv$ *case-prod append* (*fold* (λx (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))

lemma *union-mset-list*:
  mset xs #∪ *mset ys* = *mset* (*union-mset-list xs ys*)
proof −
  have $\bigwedge$*zs*. *mset* (*case-prod append* (*fold* (λx (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, *zs*))) =
      (*mset xs* #∪ *mset ys*) + *mset zs*
    by (*induct xs arbitrary*: *ys*) (*simp-all add*: *multiset-eq-iff*)
  then show *?thesis* **by** *simp*
qed

Filter

lemma *distinct-filter-eq-if*:
  distinct C $\Longrightarrow$ *length* (*filter* (*op* = *L*) *C*) = (*if* $L \in$ *set C then 1 else 0*)
  by (*induction C*) *auto*

end

Chapter 2

Definition of Entailment

This chapter defines various form of entailment.

end

2.1 Clausal Logic

theory *Clausal-Logic*
imports *../lib/Multiset-More*
begin

Resolution operates of clauses, which are disjunctions of literals. The material formalized here corresponds roughly to Sections 2.1 ("Formulas and Clauses") of Bachmair and Ganzinger, excluding the formula and term syntax.

2.1.1 Literals

Literals consist of a polarity (positive or negative) and an atom, of type *'a*.

datatype *'a literal* =
  is-pos: *Pos* (*atm-of*: *'a*)
| *Neg* (*atm-of*: *'a*)

abbreviation *is-neg* :: *'a literal* ⇒ *bool* **where** *is-neg L* ≡ ¬ *is-pos L*

lemma *Pos-atm-of-iff*[*simp*]: *Pos* (*atm-of L*) = *L* ⟷ *is-pos L*
  by *auto* (*metis literal.disc*(*1*))

lemma *Neg-atm-of-iff*[*simp*]: *Neg* (*atm-of L*) = *L* ⟷ *is-neg L*
  by *auto* (*metis literal.disc*(*2*))

lemma *ex-lit-cases*: (∃ *L*. *P L*) ⟷ (∃ *A*. *P* (*Pos A*) ∨ *P* (*Neg A*))
  by (*metis literal.exhaust*)

instantiation *literal* :: (*type*) *uminus*
begin

definition *uminus-literal* :: *'a literal* ⇒ *'a literal* **where**
  uminus L = (*if is-pos L then Neg else Pos*) (*atm-of L*)

instance ..

end

lemma
  uminus-Pos[*simp*]: $-$ *Pos* A $=$ *Neg* A **and**
  uminus-Neg[*simp*]: $-$ *Neg* A $=$ *Pos* A
  unfolding *uminus-literal-def* **by** *simp-all*

lemma *atm-of-uminus*[*simp*]:
  atm-of $(-L)$ $=$ *atm-of* L
  by (*case-tac* L, *auto*)

lemma *uminus-of-uminus-id*[*simp*]:
  $-$ $(-$ $(x$:: *'v literal*$))$ $=$ x
  by (*simp add*: *uminus-literal-def*)

lemma *uminus-not-id*[*simp*]:
  $x \neq -$ $(x$:: *'v literal*$)$
  by (*case-tac* x, *auto*)

lemma *uminus-not-id'*[*simp*]:
  $-$ $x \neq$ $(x$:: *'v literal*$)$
  by (*case-tac* x, *auto*)

lemma *uminus-eq-inj*[*iff*]:
  $-(a$::*'v literal*$)$ $=$ $-b \longleftrightarrow a = b$
  by (*case-tac* a; *case-tac* b) *auto+*

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

instantiation *literal* :: (*preorder*) *preorder*
begin

definition *less-literal* :: *'a literal* $\Rightarrow$ *'a literal* $\Rightarrow$ *bool* **where**
  less-literal L M $\longleftrightarrow$ *atm-of* L $<$ *atm-of* M $\vee$ *atm-of* $L \leq$ *atm-of* M $\wedge$ *is-neg* L $<$ *is-neg* M

definition *less-eq-literal* :: *'a literal* $\Rightarrow$ *'a literal* $\Rightarrow$ *bool* **where**
  less-eq-literal L M $\longleftrightarrow$ *atm-of* L $<$ *atm-of* M $\vee$ *atm-of* $L \leq$ *atm-of* M $\wedge$ *is-neg* $L \leq$ *is-neg* M

instance
  apply *intro-classes*
  unfolding *less-literal-def less-eq-literal-def* **by** (*auto intro*: *order-trans simp*: *less-le-not-le*)

end

instantiation *literal* :: (*order*) *order*
begin

instance
  apply *intro-classes*
  unfolding *less-eq-literal-def* **by** (*auto intro*: *literal.expand*)

end

lemma *pos-less-neg*[*simp*]: *Pos* A $<$ *Neg* A

unfolding *less-literal-def* **by** *simp*

lemma *pos-less-pos-iff* [*simp*]: *Pos* A < *Pos* $B \longleftrightarrow A < B$
unfolding *less-literal-def* **by** *simp*

lemma *pos-less-neg-iff* [*simp*]: *Pos* A < *Neg* $B \longleftrightarrow A \leq B$
unfolding *less-literal-def* **by** (*auto simp*: *less-le-not-le*)

lemma *neg-less-pos-iff* [*simp*]: *Neg* A < *Pos* $B \longleftrightarrow A < B$
unfolding *less-literal-def* **by** *simp*

lemma *neg-less-neg-iff* [*simp*]: *Neg* A < *Neg* $B \longleftrightarrow A < B$
unfolding *less-literal-def* **by** *simp*

lemma *pos-le-neg*[*simp*]: *Pos* $A \leq$ *Neg* A
unfolding *less-eq-literal-def* **by** *simp*

lemma *pos-le-pos-iff* [*simp*]: *Pos* $A \leq$ *Pos* $B \longleftrightarrow A \leq B$
unfolding *less-eq-literal-def* **by** (*auto simp*: *less-le-not-le*)

lemma *pos-le-neg-iff* [*simp*]: *Pos* $A \leq$ *Neg* $B \longleftrightarrow A \leq B$
unfolding *less-eq-literal-def* **by** (*auto simp*: *less-imp-le*)

lemma *neg-le-pos-iff* [*simp*]: *Neg* $A \leq$ *Pos* $B \longleftrightarrow A < B$
unfolding *less-eq-literal-def* **by** *simp*

lemma *neg-le-neg-iff* [*simp*]: *Neg* $A \leq$ *Neg* $B \longleftrightarrow A \leq B$
unfolding *less-eq-literal-def* **by** (*auto simp*: *less-imp-le*)

lemma *leq-imp-less-eq-atm-of*: $L \leq M \Longrightarrow$ *atm-of* $L \leq$ *atm-of* M
by (*metis less-eq-literal-def less-le-not-le*)

instantiation *literal* :: (*linorder*) *linorder*
begin

instance
apply *intro-classes*
unfolding *less-eq-literal-def less-literal-def* **by** *auto*

end

instantiation *literal* :: (*wellorder*) *wellorder*
begin

instance
proof *intro-classes*
fix P :: *'a literal* $\Rightarrow$ *bool* **and** L :: *'a literal*
assume *ih*: $\bigwedge L. (\bigwedge M. M < L \Longrightarrow P\ M) \Longrightarrow P\ L$
have $\bigwedge x. (\bigwedge y. y < x \Longrightarrow P\ (Pos\ y) \wedge P\ (Neg\ y)) \Longrightarrow P\ (Pos\ x) \wedge P\ (Neg\ x)$
by (*rule conjI*[*OF ih ih*])
(*auto simp*: *less-literal-def atm-of-def split*: *literal.splits intro*: *ih*)
hence $\bigwedge A. P\ (Pos\ A) \wedge P\ (Neg\ A)$
by (*rule less-induct*) *blast*
thus $P\ L$
by (*cases* L) *simp+*
qed

end

2.1.2 Clauses

Clauses are (finite) multisets of literals.

type-synonym *'a clause* = *'a literal multiset*

abbreviation *poss* :: *'a multiset* ⇒ *'a clause* **where** *poss AA* ≡ {#*Pos A*. *A* ∈# *AA*#}
abbreviation *negs* :: *'a multiset* ⇒ *'a clause* **where** *negs AA* ≡ {#*Neg A*. *A* ∈# *AA*#}

lemma *image-replicate-mset*[*simp*]: {#*f A*. *A* ∈# *replicate-mset n A*#} = *replicate-mset n* (*f A*)
  by (*induct n*) (*simp*, *subst replicate-mset-Suc*, *simp*)

lemma *Max-in-lits*: *C* ≠ {#} ⟹ *Max* (*set-mset C*) ∈# *C*
  by (*rule Max-in*[*OF finite-set-mset*, *unfolded set-mset-eq-empty-iff*])

lemma *Max-atm-of-set-mset-commute*: *C* ≠ {#} ⟹ *Max* (*atm-of* ‘ *set-mset C*) = *atm-of* (*Max* (*set-mset C*))
  by (*rule mono-Max-commute*[*symmetric*])
    (*auto simp*: *mono-def atm-of-def less-eq-literal-def less-literal-def*)

lemma *Max-pos-neg-less-multiset*:
  assumes *max*: *Max* (*set-mset C*) = *Pos A* **and** *neg*: *Neg A* ∈# *D*
  shows *C* #⊂# *D*
proof −
  have *Max* (*set-mset C*) < *Neg A*
    using *max* **by** *simp*
  thus *?thesis*
    using *neg* **by** (*metis* (*no-types*) *ex-gt-imp-less-multiset Max-less-iff*[*OF finite-set-mset*]
      all-not-in-conv)
qed

lemma *pos-Max-imp-neg-notin*: *Max* (*set-mset C*) = *Pos A* ⟹ *Neg A* ∉# *C*
  using *Max-pos-neg-less-multiset*[*unfolded multiset-linorder.not-le*[*symmetric*]] **by** *blast*

lemma *less-eq-Max-lit*: *C* ≠ {#} ⟹ *C* #⊆# *D* ⟹ *Max* (*set-mset C*) ≤ *Max* (*set-mset D*)
proof (*unfold le-multiset*$_{HO}$)
  assume *ne*: *C* ≠ {#} **and** *ex-gt*: ∀ *x*. *count D x* < *count C x* ⟶ (∃ *y* > *x*. *count C y* < *count D y*)
  from *ne* **have** *Max* (*set-mset C*) ∈# *C*
    by (*fast intro*: *Max-in-lits*)
  hence ∃ *l*. *l* ∈# *D* ∧ ¬ *l* < *Max* (*set-mset C*)
    using *ex-gt* **by** (*metis count-greater-zero-iff count-inI less-not-sym*)
  hence ¬ *Max* (*set-mset D*) < *Max* (*set-mset C*)
    by (*metis Max.coboundedI*[*OF finite-set-mset*] *le-less-trans*)
  thus *?thesis*
    by *simp*
qed

definition *atms-of* :: *'a clause* ⇒ *'a set* **where**
  atms-of C = *atm-of* ‘ *set-mset C*

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

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

lemma *atms-of-union-mset*[*simp*]:
atms-of $(A \, \#\cup \, B) =$ *atms-of* $A \cup$ *atms-of* B
unfolding *atms-of-def* **by** (*auto simp*: *max-def split*: *if-split-asm*)

lemma *finite-atms-of*[*iff*]: *finite* (*atms-of* C)
unfolding *atms-of-def* **by** *simp*

lemma *atm-of-lit-in-atms-of*: $L \in\# \, C \Longrightarrow$ *atm-of* $L \in$ *atms-of* C
unfolding *atms-of-def* **by** *simp*

lemma *atms-of-plus*[*simp*]: *atms-of* $(C + D) =$ *atms-of* $C \cup$ *atms-of* D
unfolding *atms-of-def image-def* **by** *auto*

lemma *pos-lit-in-atms-of*: *Pos* $A \in\# \, C \Longrightarrow A \in$ *atms-of* C
unfolding *atms-of-def* **by** (*metis image-iff literal.sel*(*1*))

lemma *neg-lit-in-atms-of*: *Neg* $A \in\# \, C \Longrightarrow A \in$ *atms-of* C
unfolding *atms-of-def* **by** (*metis image-iff literal.sel*(*2*))

lemma *atm-imp-pos-or-neg-lit*: $A \in$ *atms-of* $C \Longrightarrow$ *Pos* $A \in\# \, C \lor$ *Neg* $A \in\# \, C$
unfolding *atms-of-def image-def mem-Collect-eq*
by (*metis Neg-atm-of-iff Pos-atm-of-iff*)

lemma *atm-iff-pos-or-neg-lit*: $A \in$ *atms-of* $L \longleftrightarrow$ *Pos* $A \in\# \, L \lor$ *Neg* $A \in\# \, L$
by (*auto intro*: *pos-lit-in-atms-of neg-lit-in-atms-of dest*: *atm-imp-pos-or-neg-lit*)

lemma *atm-of-eq-atm-of*:
atm-of $L =$ *atm-of* $L' \longleftrightarrow (L = L' \lor L = -L')$
by (*cases* L; *cases* L') *auto*

lemma *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*:
atm-of $L \in$ *atm-of* ‘ $I \longleftrightarrow (L \in I \lor -L \in I)$
by (*auto intro*: *rev-image-eqI simp*: *atm-of-eq-atm-of*)

lemma *lits-subseteq-imp-atms-subseteq*: *set-mset* $C \subseteq$ *set-mset* $D \Longrightarrow$ *atms-of* $C \subseteq$ *atms-of* D
unfolding *atms-of-def* **by** *blast*

lemma *atms-empty-iff-empty*[*iff*]: *atms-of* $C = \{\} \longleftrightarrow C = \{\#\}$
unfolding *atms-of-def image-def Collect-empty-eq*
by (*metis all-not-in-conv set-mset-eq-empty-iff*)

lemma
atms-of-poss[*simp*]: *atms-of* (*poss* AA) $=$ *set-mset* AA **and**
atms-of-negg[*simp*]: *atms-of* (*negs* AA) $=$ *set-mset* AA
unfolding *atms-of-def image-def* **by** *auto*

lemma *less-eq-Max-atms-of*: $C \neq \{\#\} \Longrightarrow C \, \#\subseteq\# \, D \Longrightarrow$ *Max* (*atms-of* C) $\leq$ *Max* (*atms-of* D)
unfolding *atms-of-def*
by (*metis Max-atm-of-set-mset-commute le-multiset-empty-right leq-imp-less-eq-atm-of less-eq-Max-lit*)

lemma *le-multiset-Max-in-imp-Max*:
Max (*atms-of* D) $= A \Longrightarrow C \, \#\subseteq\# \, D \Longrightarrow A \in$ *atms-of* $C \Longrightarrow$ *Max* (*atms-of* C) $= A$

by (*metis Max.coboundedI*[*OF finite-atms-of*] *atms-of-def empty-iff eq-iff image-subsetI less-eq-Max-atms-of set-mset-empty subset-Compl-self-eq*)

lemma *atm-of-Max-lit*[*simp*]: $C \neq \{\#\} \Longrightarrow$ *atm-of* (*Max* (*set-mset C*)) = *Max* (*atms-of C*)
unfolding *atms-of-def Max-atm-of-set-mset-commute* **..**

lemma *Max-lit-eq-pos-or-neg-Max-atm*:
$C \neq \{\#\} \Longrightarrow$ *Max* (*set-mset C*) = *Pos* (*Max* (*atms-of C*)) $\vee$ *Max* (*set-mset C*) = *Neg* (*Max* (*atms-of C*))
by (*metis Neg-atm-of-iff Pos-atm-of-iff atm-of-Max-lit*)

lemma *atms-less-imp-lit-less-pos*: $(\bigwedge B.\ B \in$ *atms-of* $C \Longrightarrow B < A) \Longrightarrow L \in\# C \Longrightarrow L <$ *Pos* A
unfolding *atms-of-def less-literal-def* **by** *force*

lemma *atms-less-eq-imp-lit-less-eq-neg*: $(\bigwedge B.\ B \in$ *atms-of* $C \Longrightarrow B \leq A) \Longrightarrow L \in\# C \Longrightarrow L \leq$ *Neg* A
unfolding *less-eq-literal-def* **by** (*simp add*: *atm-of-lit-in-atms-of*)

end

2.2 Clausal Logic

theory *Herbrand-Interpretation*
imports *Clausal-Logic*
begin

Resolution operates of clauses, which are disjunctions of literals. The material formalized here corresponds roughly to Sections 2.2 ("Herbrand Interpretations") of Bachmair and Ganzinger, excluding the formula and term syntax.

2.2.1 Herbrand Interpretations

A Herbrand interpretation is a set of ground atoms that are to be considered true.

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

definition *true-lit* :: *'a interp* ⇒ *'a literal* ⇒ *bool* (**infix** ⊨*l 50*) **where**
$I \models l\ L \longleftrightarrow$ (*if is-pos L then* ($\lambda P.\ P$) *else Not*) (*atm-of* $L \in I$)

lemma *true-lit-simps*[*simp*]:
$I \models l$ *Pos* $A \longleftrightarrow A \in I$
$I \models l$ *Neg* $A \longleftrightarrow A \notin I$
unfolding *true-lit-def* **by** *auto*

lemma *true-lit-iff*[*iff*]: $I \models l\ L \longleftrightarrow (\exists A.\ L =$ *Pos* $A \wedge A \in I \vee L =$ *Neg* $A \wedge A \notin I)$
by (*cases L*) *simp+*

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

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

lemma *true-cls-singleton*[*iff*]: $I \models \{\#L\#\} \longleftrightarrow I \models l\ L$
unfolding *true-cls-def* **by** *simp*

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

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

lemma
assumes $I \subseteq J$
shows
false-to-true-imp-ex-pos: $\neg I \models C \Longrightarrow J \models C \Longrightarrow \exists A \in J.$ *Pos* $A \in\# C$ **and**
true-to-false-imp-ex-neg: $I \models C \Longrightarrow \neg J \models C \Longrightarrow \exists A \in J.$ *Neg* $A \in\# C$
using *assms* **unfolding** *subset-iff true-cls-def*
by (*metis literal.collapse true-lit-simps*)+

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

lemma *pos-literal-in-imp-true-cls*[*intro*]: *Pos* $A \in\# C \Longrightarrow A \in I \Longrightarrow I \models C$
by (*metis true-cls-def true-lit-simps*(*1*))

lemma *neg-literal-notin-imp-true-cls*[*intro*]: *Neg* $A \in\# C \Longrightarrow A \notin I \Longrightarrow I \models C$
by (*metis true-cls-def true-lit-simps*(*2*))

lemma *pos-neg-in-imp-true*: *Pos* $A \in\# C \Longrightarrow$ *Neg* $A \in\# C \Longrightarrow I \models C$
unfolding *true-cls-def* **by** (*metis true-lit-simps*)

definition *true-clss* :: *'a 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*[*iff*]: $I \models s$ {}
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-union*[*iff*]: $I \models s\ CC \cup DD \longleftrightarrow I \models s\ CC \land I \models s\ DD$
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*

abbreviation *satisfiable* :: *'a clause set* $\Rightarrow$ *bool* **where**
satisfiable $CC \equiv \exists I.\ I \models s\ CC$

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

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

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** *auto*

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

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

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

end

2.3 Partial Clausal Logic

theory *Partial-Clausal-Logic*
imports *../lib/Clausal-Logic List-More*
begin

We define here entailment by a set of literals. This is *not* an Herbrand interpretation and 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.

2.3.1 Clauses

Clauses are (finite) multisets of literals.

type-synonym $'a\ clause = 'a\ literal\ multiset$
type-synonym $'v\ clauses = 'v\ clause\ set$

2.3.2 Partial Interpretations

type-synonym $'a\ interp = 'a\ literal\ set$

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

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

Consistency

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

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 *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*

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 literal multiset set* $\Rightarrow$ *'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* $\Rightarrow$ *'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-single-set-mset-atns-of*[*simp*]:
atms-of-ms (*single* ' *set-mset* B) = *atms-of* B
unfolding *atms-of-ms-def atms-of-def* **by** *auto*

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*[]
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 $\{\#A\#\} + D \in \psi s$
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* $\cup$ *Ib*) = *atms-of-s Ia* $\cup$ *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* $L\}$ $\cup$ *atms-of-s Ib*

unfolding *atms-of-s-def* **by** *auto*

lemma *in-atms-of-s-decomp*[*iff*]:
$P \in$ *atms-of-s* $I \longleftrightarrow (Pos\ P \in I \lor Neg\ P \in I)$ (**is** $?P \longleftrightarrow ?Q$)
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 $L' \in$ *atm-of* ‘ $B \Longrightarrow L' \in B \lor -L' \in B$
using *atms-of-s-def* **by** (*cases* L') *fastforce+*

Totality

definition *total-over-set* :: *'a interp* ⇒ *'a set* ⇒ *bool* **where**
total-over-set $I\ S = (\forall l{\in}S.\ Pos\ l \in I \lor Neg\ l \in I)$

definition *total-over-m* :: *'a literal set* ⇒ *'a clause set* ⇒ *bool* **where**
total-over-m $I\ \psi s$ = *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 $I\ \{L\} \longleftrightarrow (Pos\ L \in I \lor Neg\ L \in I)$
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-set-insert*[*iff*]:
total-over-set I (*insert* $L\ Ls$) $\longleftrightarrow ((Pos\ L \in I \lor Neg\ L \in I) \land$ *total-over-set* $I\ Ls$)
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-set-union*[*iff*]:
total-over-set $I\ (Ls \cup Ls') \longleftrightarrow$ (*total-over-set* $I\ Ls \land$ *total-over-set* $I\ Ls'$)
unfolding *total-over-set-def* **by** *auto*

lemma *total-over-m-subset*:
$A \subseteq B \Longrightarrow$ *total-over-m* $I\ B \Longrightarrow$ *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* $I\ \{C + D\} \longleftrightarrow$ (*total-over-m* $I\ \{C\} \land$ *total-over-m* $I\ \{D\}$)
using *assms* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*

lemma *total-over-m-union*[*iff*]:
total-over-m $I\ (A \cup B) \longleftrightarrow$ (*total-over-m* $I\ A \land$ *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*) $\longleftrightarrow$ (*total-over-set I* (*atms-of a*) $\wedge$ *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 clauses*
  assumes *total*: *total-over-m I A*
  shows $\exists I'$. *total-over-m* ($I \cup I'$) ($A \cup B$)
    $\wedge$ ($\forall x \in I'$. *atm-of x* $\in$ *atms-of-ms B* $\wedge$ *atm-of x* $\notin$ *atms-of-ms A*)
proof –
  let *?I'* = {*Pos v* |*v. v*$\in$ *atms-of-ms B* $\wedge$ *v* $\notin$ *atms-of-ms A*}
  have $\forall x \in$*?I'*. *atm-of x* $\in$ *atms-of-ms B* $\wedge$ *atm-of x* $\notin$ *atms-of-ms A* **by** *auto*
  moreover have *total-over-m* ($I \cup$ *?I'*) ($A \cup B$)
    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 clauses*
  assumes
    total: *total-over-m I A* **and**
    cons: *consistent-interp I*
  shows $\exists I'$. *total-over-m* ($I \cup I'$) ($A \cup B$)
    $\wedge$ ($\forall x \in I'$. *atm-of x* $\in$ *atms-of-ms B* $\wedge$ *atm-of x* $\notin$ *atms-of-ms A*) $\wedge$ *consistent-interp* ($I \cup I'$)
proof –
  let *?I'* = {*Pos v* |*v. v*$\in$ *atms-of-ms B* $\wedge$ *v* $\notin$ *atms-of-ms A* $\wedge$ *Pos v* $\notin$ *I* $\wedge$ *Neg v* $\notin$ *I*}
  have $\forall x \in$*?I'*. *atm-of x* $\in$ *atms-of-ms B* $\wedge$ *atm-of x* $\notin$ *atms-of-ms A* **by** *auto*
  moreover have *total-over-m* ($I \cup$ *?I'*) ($A \cup B$)
    using *total* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
  moreover have *consistent-interp* ($I \cup$ *?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 {#*A*#} + *D* $\in$ ψs
  and *total-over-set I* (*atms-of-ms* ψs)
  shows $A \in I \vee -A \in I$
  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* $\in I \vee$ *Neg l* $\in I \vee l =$ *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 ¬ *?thesis*
then have *atm-of* $L \in$ *atms-of* ψ **by** *auto*
then have *Pos* (*atm-of* L) $\in\#$ ψ $\vee$ *Neg* (*atm-of* L) $\in\#$ ψ
using *atm-imp-pos-or-neg-lit* **by** *metis*
then have $L \in\# \psi \vee - L \in\# \psi$ **by** (*cases* L) *auto*
then show *False* **using** L **by** *auto*
qed
ultimately show *Pos* $l \in I \vee$ *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*

Interpretations

definition *true-cls* :: *'a 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-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 interp* $\Rightarrow$ *'a clauses* $\Rightarrow$ *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*)

Satisfiability

definition *satisfiable* :: *'a clause set* $\Rightarrow$ *bool* **where**
satisfiable $CC \equiv \exists I.\ (I \models s\ CC \wedge$ *consistent-interp* $I \wedge$ *total-over-m* $I\ CC)$

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

abbreviation *unsatisfiable* :: *'a clause set* $\Rightarrow$ *bool* **where**
unsatisfiable $CC \equiv \neg$ *satisfiable* CC

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

lemma *satisfiable-def-min*:
satisfiable CC
$\longleftrightarrow (\exists I.\ I \models s\ CC \wedge$ *consistent-interp* $I \wedge$ *total-over-m* $I\ CC \wedge$ *atm-of*'$I =$ *atms-of-ms* $CC)$
(**is** *?sat* $\longleftrightarrow$ *?B*)
proof
assume *?B* **then show** *?sat* **by** (*auto simp add*: *satisfiable-def*)
next
assume *?sat*
then obtain I **where**
I-CC: $I \models s\ CC$ **and**
cons: *consistent-interp* I **and**
tot: *total-over-m* $I\ CC$
unfolding *satisfiable-def* **by** *auto*
let $?I = \{P.\ P \in I \wedge$ *atm-of* $P \in$ *atms-of-ms* $CC\}$

have *I-CC*: $?I \models s\ CC$
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*[*iff*]:
(∃ *I*. *consistent-interp I* ∧ *I* ⊨*s* φ) ⟷ *satisfiable* φ (**is** (∃ *I*. *?Q I*) ⟷ *?S*)
proof
assume *?S*
then show ∃ *I*. *?Q I* **unfolding** *satisfiable-def* **by** *auto*
next
assume ∃ *I*. *?Q I*
then obtain *I* **where** *cons*: *consistent-interp I* **and** *I*: *I* ⊨*s* φ **by** *metis*
let *?I'* = {*Pos v* |*v*. *v* ∉ *atms-of-s I* ∧ *v* ∈ *atms-of-ms* φ}
have *consistent-interp* (*I* ∪ *?I'*)
using *cons* **unfolding** *consistent-interp-def* **by** (*intro allI*) (*rename-tac L*, *case-tac L*, *auto*)
moreover have *total-over-m* (*I* ∪ *?I'*) φ
unfolding *total-over-m-def total-over-set-def* **by** *auto*
moreover have *I* ∪ *?I'* ⊨*s* φ
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* ⟹ *I* ⊨*s* φ ⟹ *satisfiable* φ
using *satisfiable-carac* **by** *metis*

Entailment for Multisets of Clauses

definition *true-cls-mset* :: *'a interp* ⇒ *'a clause multiset* ⇒ *bool* (**infix** ⊨*m 50*) **where**
I ⊨*m CC* ⟷ (∀ *C* ∈# *CC*. *I* ⊨ *C*)

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

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

lemma *true-cls-mset-union*[*iff*]: *I* ⊨*m CC* + *DD* ⟷ *I* ⊨*m CC* ∧ *I* ⊨*m DD*
unfolding *true-cls-mset-def* **by** *fastforce*

lemma *true-cls-mset-image-mset*[*iff*]: *I* ⊨*m image-mset f A* ⟷ (∀ *x* ∈# *A*. *I* ⊨ *f x*)
unfolding *true-cls-mset-def* **by** *fastforce*

lemma *true-cls-mset-mono*: *set-mset DD* ⊆ *set-mset CC* ⟹ *I* ⊨*m CC* ⟹ *I* ⊨*m DD*
unfolding *true-cls-mset-def subset-iff* **by** *auto*

lemma *true-clss-set-mset*[*iff*]: *I* ⊨*s set-mset CC* ⟷ *I* ⊨*m CC*
unfolding *true-clss-def true-cls-mset-def* **by** *auto*

lemma *true-cls-mset-increasing-r*[*simp*]:
I ⊨*m CC* ⟹ *I* ∪ *J* ⊨*m CC*
unfolding *true-cls-mset-def* **by** *auto*

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

theorem *true-clss-remove-unused*:
  assumes $I \models s\ \psi$
  shows $\{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of-ms}\ \psi\} \models s\ \psi$
  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 $\{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of}\ x\} \models x$
    by (*simp only*: *true-cls-remove-unused*[*of I*])
  moreover have $\{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of}\ x\} \subseteq \{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of-ms}\ \psi\}$
    using ‹$x \in \psi$› **by** (*auto simp add*: *atms-of-ms-def*)
  ultimately show $\{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of-ms}\ \psi\} \models x$
    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: $\forall v \in I'.\ \textit{atm-of}\ v \notin \textit{atms-of}\ \psi$
  shows $I \models \psi$
proof −
  let $?I = \{v \in I \cup I'.\ \textit{atm-of}\ v \in \textit{atms-of}\ \psi\}$
  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 $\exists A.\ x = \textit{Pos}\ A \lor x = \textit{Neg}\ A$
  using *assms literal.exhaust-sel* **by** *blast*

lemma *atms-of-ms-empty*:
  fixes $\psi :: 'v\ \textit{clauses}$
  assumes $\textit{atms-of-ms}\ \psi = \{\}$
  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*: $\textit{atms-of-s}\ A \cap \textit{atms-of-s}\ I = \{\}$
 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* $\{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of-ms}\ \psi\}$ 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*$'\models D$
and *atm-of a* $\notin$ *atms-of D*
shows $M' \models D$
using *assms* **by** (*auto simp add*: *atm-of-lit-in-atms-of true-cls-def*)

lemma *total-over-set-atm-of*:
fixes I :: *'v interp* **and** K :: *'v set*
shows *total-over-set I K* $\longleftrightarrow$ $(\forall l \in K.\ l \in (\textit{atm-of}\ `\ I))$
unfolding *total-over-set-def* **by** (*metis atms-of-s-def in-atms-of-s-decomp*)

Tautologies

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

definition *tautology* (ψ:: *'v clause*) $\equiv \forall I.$ *total-over-set I* (*atms-of* ψ) $\longrightarrow I \models \psi$

lemma *tautology-Pos-Neg*[*intro*]:
assumes *Pos p* $\in\#$ *A* **and** *Neg p* $\in\#$ *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 $L \in\# A$ **and** $-L \in\# A$
shows *tautology A*
by (*metis assms literal.exhaust tautology-Pos-Neg uminus-Neg uminus-Pos*)

lemma *tautology-exists-Pos-Neg*:
assumes *tautology* ψ
shows $\exists p.$ *Pos p* $\in\# \psi \wedge$ *Neg p* $\in\# \psi$
proof (*rule ccontr*)
assume *p*: $\neg$ ($\exists p.$ *Pos p* $\in\# \psi \wedge$ *Neg p* $\in\# \psi$)
let $?I = \{-L \mid L.\ L \in\# \psi\}$
have *total-over-set ?I* (*atms-of* ψ)
unfolding *total-over-set-def* **using** *atm-imp-pos-or-neg-lit* **by** *force*
moreover have $\neg\ ?I \models \psi$
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 $\psi \longleftrightarrow (\exists p.\ Pos\ p \in\#\ \psi \wedge Neg\ p \in\#\ \psi)$
using *tautology-exists-Pos-Neg* **by** *auto*

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

lemma *tautology-add-single*:
tautology $(\{\#a\#\} + L) \longleftrightarrow$ *tautology* $L \vee -a \in\#\ L$
unfolding *tautology-decomp* **by** (*cases a*) *auto*

lemma *minus-interp-tautology*:
assumes $\{-L \mid L.\ L{\in}\#\ \chi\} \models \chi$
shows *tautology* χ
proof −
obtain L **where** $L \in\#\ \chi \wedge -L \in\#\ \chi$
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 $I \cup \{Pos\ P\} \models \varphi$
and $I \cup \{Neg\ P\} \models \varphi$
shows $I \models \varphi \vee$ *tautology* φ
using *assms* **unfolding** *true-cls-def* **by** *auto*

lemma *tautology-imp-tautology*:
fixes $\chi\ \chi'$ :: *'v clause*
assumes $\forall I.$ *total-over-m* $I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$ **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 $?I' = \{Pos\ v\ |v.\ v{\in}$ *atms-of* $\chi \wedge v \notin$ *atms-of-s* $I\}$
have *totI'*: *total-over-m* $(I \cup ?I')\ \{\chi\}$ **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
then have χ: $I \cup ?I' \models \chi$ **using** *assms*(*2*) **unfolding** *total-over-m-def tautology-def* **by** *simp*
then have $I \cup (?I' - I) \models \chi'$ **using** *assms*(*1*) *totI'* **by** *auto*
moreover have $\bigwedge L.\ L \in\#\ \chi' \Longrightarrow L \notin ?I'$
using *totI* **unfolding** *total-over-set-def* **by** (*auto dest*: *pos-lit-in-atms-of*)
ultimately show $I \models \chi'$ **unfolding** *true-cls-def* **by** *auto*
qed

Entailment for clauses and propositions

We also need entailment of clauses by other clauses.

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

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

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

definition *true-clss-clss* :: *'a clauses* ⇒ *'a clauses* ⇒ *bool* (**infix** ⊨*ps* *49*) **where**
$N \models ps\ N' \longleftrightarrow (\forall I.$ *total-over-m* $I\ (N \cup N') \longrightarrow$ *consistent-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-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*)

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-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*]:
  $A \models ps\ C \cup D \longleftrightarrow (A \models ps\ C \land A \models ps\ D)$
proof
  {
    fix $A\ C\ D$ :: *'a clauses*
    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* $D \land$ *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 $A \models ps\ C \land A \models ps\ D$ **using** *H*[*of A*] *Un-commute*[*of C D*] **by** *metis*
next
  assume $A \models ps\ C \land A \models ps\ D$
  then show $A \models ps\ C \cup D$
    unfolding *true-clss-clss-def* **by** *auto*
qed

lemma *true-clss-clss-insert*[*iff*]:
  $A \models ps\ insert\ L\ Ls \longleftrightarrow (A \models p\ L \land A \models ps\ Ls)$
  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*)

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*: $A \cup C \models ps\ D$
assume $B \models ps\ C$
then have *f2*: $\bigwedge M.\ M \cup B \models ps\ C$
by (*meson true-clss-clss-union-l-r*)
have $\bigwedge M.\ C \cup (M \cup A) \models ps\ D$
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*: $N \models p\ D + \{\#- L\#\}$
and *C*: $N \models p\ C + \{\#L\#\}$
shows $N \models p\ D + C$
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**
$I \models s\ N$
{
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* ‹$I \models s\ N$› *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* ‹$I \models s\ N$› *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'* $\{D + \{\#- L\#\}\}$
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 $?I' \models s\ N$ **using** ‹$I \models s\ N$› **using** *true-clss-union-increase* **by** *blast*
ultimately have $?I' \models D + \{\#- L\#\}$
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-union-mset-true-clss-cls-or-not-true-clss-cls-or*:
assumes
D: $N \models p\ D + \{\#- L\#\}$ **and**
C: $N \models p\ C + \{\#L\#\}$
shows $N \models p\ D \#\cup C$
unfolding *true-clss-cls-def*
proof (*intro allI impI*)
fix *I*
assume
tot: *total-over-m I* $(N \cup \{D \#\cup C\})$ **and**
consistent-interp I **and**
$I \models s\ N$
{
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* ‹$I \models s\ N$› *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* ‹$I \models s\ N$› *tot* ‹*consistent-interp I*› **unfolding** *true-clss-cls-def* **by** *auto*
ultimately have $I \models D \#\cup C$ **using** ‹*consistent-interp I*› **unfolding** *consistent-interp-def*
by *auto*
}

moreover {
  assume *L*: $L \notin I \land -L \notin I$
  let $?I' = I \cup \{L\}$
  have *consistent-interp* $?I'$ **using** *L* ‹*consistent-interp I*› **by** *auto*
  moreover have *total-over-m* $?I'$ $\{D + \{\#- L\#\}\}$
    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 $?I' \models s\ N$ **using** ‹$I \models s\ N$› **using** *true-clss-union-increase* **by** *blast*
  ultimately have $?I' \models D + \{\#- L\#\}$
    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 $L \notin\# D \land -L \notin\# D$
      using *L* **unfolding** *total-over-set-def atms-of-def* **by** (*cases L*) *force+*
  ultimately have $I \models D \ \#\cup\ C$ **unfolding** *true-cls-def* **by** *auto*
}
ultimately show $I \models D \ \#\cup\ C$ **by** *blast*
qed

2.3.3 Subsumptions

lemma *subsumption-total-over-m*:
  assumes $A \subseteq\# B$
  shows *total-over-m I* $\{B\} \Longrightarrow$ *total-over-m I* $\{A\}$
  using *assms* **unfolding** *subset-mset-def total-over-m-def total-over-set-def*
  by (*auto simp add*: *mset-le-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 (*fastforce simp*: *atm-of-eq-atm-of atms-of-def*)

lemma *subsumption-chained*:
  assumes
    $\forall I.$ *total-over-m I* $\{D\} \longrightarrow I \models D \longrightarrow I \models \varphi$ **and**
    $C \subseteq\# D$
  shows ($\forall I.$ *total-over-m I* $\{C\} \longrightarrow I \models C \longrightarrow I \models \varphi$) $\lor$ *tautology* φ
  using *assms*
proof (*induct card* $\{$*Pos v* $\mid v.\ v \in$ *atms-of D* $\land\ v \notin$ *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* $\subseteq$ *atms-of C* **by** *auto*
  then have $\forall I.$ *total-over-m I* $\{C\} \longrightarrow$ *total-over-m I* $\{D\}$
    unfolding *total-over-m-def total-over-set-def* **by** *auto*
  moreover have $\forall I.\ I \models C \longrightarrow I \models D$ **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* $\mid v.\ v \in$ *atms-of D* $\land\ v \notin$ *atms-of C*$\}$
  have *finite ?atms* **by** *auto*
  then obtain *L* **where** *L*: $L \in$ *?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* $\{D\} \subseteq$ *atms-of-ms* $\{?D'\} \cup \{$*atm-of L*$\}$ **by** *auto*

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

  assume *IDL*: $I \models ?D'$
  then have $I \cup \{L\} \models D$ **unfolding** *true-cls-def* **by** *force*
  then have $I \cup \{L\} \models \varphi$ **using** *H tot* **by** *auto*

  moreover
    have *tot'*: *total-over-m* $(I \cup \{-L\})$ $\{D\}$
      using *tot* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
    have $I \cup \{-L\} \models D$ **using** *IDL* **unfolding** *true-cls-def* **by** *force*
    then have $I \cup \{-L\} \models \varphi$ **using** *H tot'* **by** *auto*
  ultimately have $I \models \varphi \lor$ *tautology* φ
    using *L remove-literal-in-model-tautology* **by** *force*
} **note** $H' = this$

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

2.3.4 Removing Duplicates

lemma *tautology-remdups-mset*[*iff*]:
tautology (*remdups-mset* C) $\longleftrightarrow$ *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*]: $I \models$ *remdups-mset* $C \longleftrightarrow I \models C$
unfolding *true-cls-def* **by** *auto*

lemma *true-clss-cls-remdups-mset*[*iff*]: $A \models p$ *remdups-mset* $C \longleftrightarrow A \models p\ C$
unfolding *true-clss-cls-def total-over-m-def* **by** *auto*

2.3.5 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*) =
    (*op* + {#*Pos l*#}) ‘ (*simple-clss atms*)
    ∪ (*op* + {#*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-single*)
      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** *auto*
    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 by (metis assms atm-of-lit-in-atms-of atms literal.sel(1,2) subset-eq
          tautology-add-single uminus-Neg uminus-Pos)
      moreover have distinct-mset C
        using dist C' L
        by (metis assms atm-of-lit-in-atms-of atms contra-subsetD distinct-mset-add-single
          literal.sel(1,2))
      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*:
    $\bigwedge C'$. {#*Pos l*#} + $C' \notin$ *simple-clss C*
    $\bigwedge C'$. {#*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** (*metis insert-noteq-member literal.distinct*(*1*) *union-commute*)
      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* = (*op* + {#*Pos l*#}) ' (*simple-clss C*)
  let *?N* = (*op* + {#*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 op* + {#*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 op* + {#*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*)

2.3.6 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*)

2.3.7 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 literal multiset 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*[]
    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*)

end
theory *Prop-Logic*
imports *Main*
begin

Chapter 3

Normalisation

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

3.1 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.

3.1.1 Definition and abstraction

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

datatype $'v$ *propo* $=$
 $FT \mid FF \mid FVar\ 'v \mid FNot\ 'v\ propo \mid FAnd\ \ 'v\ propo\ \ 'v\ propo \mid FOr\ \ 'v\ propo\ \ 'v\ propo$
 $\mid FImp\ \ 'v\ propo\ \ 'v\ propo \mid FEq\ \ 'v\ propo\ \ 'v\ propo$

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* $= CT \mid CF \mid CVar\ 'v \mid CNot \mid CAnd \mid COr \mid CImp \mid CEq$

abbreviation *nullary-connective* $\equiv \{CF\} \cup \{CT\} \cup \{CVar\ x \mid x.\ True\}$
definition *binary-connectives* $\equiv \{CAnd,\ COr,\ CImp,\ CEq\}$

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 $\varphi\ \psi :: 'v\ propo$
 assumes *nullary*: $\bigwedge\varphi\ x.\ \varphi = FF \lor \varphi = FT \lor \varphi = FVar\ x \Longrightarrow P\ \varphi$
 and *unary*: $\bigwedge\psi.\ P\ \psi \Longrightarrow P\ (FNot\ \psi)$
 and *binary*: $\bigwedge\varphi\ \psi 1\ \psi 2.\ P\ \psi 1 \Longrightarrow P\ \psi 2 \Longrightarrow \varphi = FAnd\ \psi 1\ \psi 2 \lor \varphi = FOr\ \psi 1\ \psi 2 \lor \varphi = FImp\ \psi 1$
$\psi 2$
 $\lor\ \varphi = FEq\ \psi 1\ \psi 2 \Longrightarrow P\ \varphi$
 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* $\Rightarrow$ $'v$ *propo list* $\Rightarrow$ $'v$ *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* $\Rightarrow$ $'v$ *propo list* $\Rightarrow$ *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*)

3.1.2 properties of the abstraction

First we can define simplification rules.

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

$wf\text{-}conn\ CT\ l \Longrightarrow conn\ CT\ l = FT$
$wf\text{-}conn\ CF\ l \Longrightarrow conn\ CF\ l = FF$
$wf\text{-}conn\ (CVar\ x)\ l \Longrightarrow conn\ (CVar\ x)\ l = FVar\ x$
apply (*simp-all add*: *wf-conn.simps*)
unfolding *binary-connectives-def* **by** *simp-all*

lemma *wf-conn-list-decomp*[*simp*]:
$wf\text{-}conn\ CT\ l \longleftrightarrow l = []$
$wf\text{-}conn\ CF\ l \longleftrightarrow l = []$
$wf\text{-}conn\ (CVar\ x)\ l \longleftrightarrow l = []$
$wf\text{-}conn\ CNot\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow \xi = [] \land \xi' = []$
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*:
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FT \longleftrightarrow (c = CT \land l = [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FF \longleftrightarrow (c = CF \land l = [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FVar\ x \longleftrightarrow (c = CVar\ x \land l = [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FAnd\ a\ b \longleftrightarrow (c = CAnd \land l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FOr\ a\ b \longleftrightarrow (c = COr \land l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FEq\ a\ b \longleftrightarrow (c = CEq \land l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FImp\ a\ b \longleftrightarrow (c = CImp \land l = a\ \#\ b\ \#\ [])$
$wf\text{-}conn\ c\ l \Longrightarrow conn\ c\ l = FNot\ a \longleftrightarrow (c = CNot \land l = a\ \#\ [])$
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*: $length\ l = 2 \Longrightarrow (\exists\ a\ b.\ l = a\ \#\ b\ \#\ [])$
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*: $c \in binary\text{-}connectives$
shows $length\ l = 2 \longleftrightarrow wf\text{-}conn\ c\ l$
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* = [ψ]
  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*: $conn\ ca\ l = conn\ c\ \psi s$
shows $ca = c \wedge \psi s = l$
using *corr*
proof (*cases ca l rule*: *wf-conn.cases*)
case (*wf-conn-nullary v*)
then show $ca = c \wedge \psi s = l$ **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 ∗: $FNot\ \psi' = conn\ c\ \psi s$ **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 $\psi s = l$ **using** ∗ *conn-inj-not*(*2*) *corr'* *wf-conn-unary*(*1*) **by** *force*
ultimately show $ca = c \wedge \psi s = l$ **by** *auto*
next
case (*wf-conn-binary* ψ' ψ'')
then show $ca = c \wedge \psi s = l$
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

3.1.3 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*]: $\varphi \preceq \varphi$ |
subformula-into-subformula: $\psi \in set\ l \Longrightarrow wf\text{-}conn\ c\ l \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq conn\ c\ l$

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*: $FNot\ \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi$
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*: $c \in binary\text{-}connectives$
shows $f \preceq conn\ c\ [f, g]$
and $g \preceq conn\ c\ [f, g]$
proof −
have *a*: $wf\text{-}conn\ c\ (f\#\ [g])$ **using** *conn wf-conn-binary binary-connectives-def* **by** *auto*
moreover have *b*: $f \preceq f$ **using** *subformula-refl* **by** *auto*
ultimately show $f \preceq conn\ c\ [f, g]$
by (*metis append-Nil in-set-conv-decomp subformula-into-subformula*)
next
have *a*: $wf\text{-}conn\ c\ ([f]\ @\ [g])$ **using** *conn wf-conn-binary binary-connectives-def* **by** *auto*
moreover have *b*: $g \preceq g$ **using** *subformula-refl* **by** *auto*
ultimately show $g \preceq conn\ c\ [f, g]$ **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*: $\psi = FT \lor \psi = FF \lor \psi = FVar\ x$
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* $\lor$ ($\exists \psi.\ \psi \in$ *set l* $\land\ \varphi \preceq \psi$))
apply *standard*
using *subfurmula-not-incl-eq* **apply** *metis*
by (*auto simp add*: *subformula-into-subformula*)

lemma *subformula-decomp-explicit*[*simp*]:
$\varphi \preceq$ *FAnd* ψ $\psi' \longleftrightarrow$ ($\varphi =$ *FAnd* ψ $\psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi'$) (**is** *?P FAnd*)
$\varphi \preceq$ *FOr* ψ $\psi' \longleftrightarrow$ ($\varphi =$ *FOr* ψ $\psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi'$)
$\varphi \preceq$ *FEq* ψ $\psi' \longleftrightarrow$ ($\varphi =$ *FEq* ψ $\psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi'$)
$\varphi \preceq$ *FImp* ψ $\psi' \longleftrightarrow$ ($\varphi =$ *FImp* ψ $\psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi'$)
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'] $\lor$ ($\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\land\ \varphi \preceq \psi''$))
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'] $\lor$ ($\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\land\ \varphi \preceq \psi''$))
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'] $\lor$ ($\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\land\ \varphi \preceq \psi''$))
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'] $\lor$ ($\exists \psi''.\ \psi'' \in$ *set* [ψ, ψ'] $\land\ \varphi \preceq \psi''$))
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 $vars\text{-}of\text{-}prop\ \psi \subseteq vars\text{-}of\text{-}prop\ (conn\ c\ l)$ **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 $\psi = a \lor \psi = b$ **using** *incl* **by** *auto*
then show $vars\text{-}of\text{-}prop\ \psi \subseteq vars\text{-}of\text{-}prop\ (conn\ c\ l)$
using *ab c* **unfolding** *binary-connectives-def* **by** *auto*
next
case *unary* **note** $c = this$
fix $\varphi :: 'v\ propo$
have $l = [\psi]$ **using** *corr c incl split-list* **by** *force*
then show $vars\text{-}of\text{-}prop\ \psi \subseteq vars\text{-}of\text{-}prop\ (conn\ c\ l)$ **using** *c* **by** *auto*
qed

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:
$\varphi \preceq \psi \Longrightarrow vars\text{-}of\text{-}prop\ \varphi \subseteq vars\text{-}of\text{-}prop\ \psi$
apply (*induct rule*: *subformula.induct*)
apply *simp*
using *vars-of-prop-incl-conn* **by** *blast*

3.1.4 Positions

Instead of 1 or 2 we use L or R

datatype $sign = L \mid R$

We use *nil* instead of ε.

fun $pos :: 'v\ propo \Rightarrow sign\ list\ set$ **where**
$pos\ FF = \{[]\}\ |$
$pos\ FT = \{[]\}\ |$
$pos\ (FVar\ x) = \{[]\}\ |$
$pos\ (FAnd\ \varphi\ \psi) = \{[]\} \cup \{\ L \# p \mid p.\ p\in pos\ \varphi\} \cup \{\ R \# p \mid p.\ p\in pos\ \psi\}\ |$
$pos\ (FOr\ \varphi\ \psi) = \{[]\} \cup \{\ L \# p \mid p.\ p\in pos\ \varphi\} \cup \{\ R \# p \mid p.\ p\in pos\ \psi\}\ |$
$pos\ (FEq\ \varphi\ \psi) = \{[]\} \cup \{\ L \# p \mid p.\ p\in pos\ \varphi\} \cup \{\ R \# p \mid p.\ p\in pos\ \psi\}\ |$
$pos\ (FImp\ \varphi\ \psi) = \{[]\} \cup \{\ L \# p \mid p.\ p\in pos\ \varphi\} \cup \{\ R \# p \mid p.\ p\in pos\ \psi\}\ |$
$pos\ (FNot\ \varphi) = \{[]\} \cup \{\ L \# p \mid p.\ p\in pos\ \varphi\}$

lemma *finite-pos*: *finite* $(pos\ \varphi)$
by (*induct* φ, *auto*)

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

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

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

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

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

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

definition *prop-size* **where** $prop\text{-}size\ \varphi = card\ (pos\ \varphi)$

lemma *prop-size-vars-of-prop*:
fixes φ :: *'v propo*
shows $card\ (vars\text{-}of\text{-}prop\ \varphi) \leq prop\text{-}size\ \varphi$

unfolding *prop-size-def* **apply** (*induct* φ, *auto simp add*: *cons-inject finite-inj-comp-set finite-pos*)
proof −
fix $\varphi 1\ \varphi 2$ :: *'v propo*
assume *IH1*: $card\ (vars\text{-}of\text{-}prop\ \varphi 1) \leq card\ (pos\ \varphi 1)$
and *IH2*: $card\ (vars\text{-}of\text{-}prop\ \varphi 2) \leq card\ (pos\ \varphi 2)$
let $?L = \{L\ \#\ p\ |p.\ p \in pos\ \varphi 1\}$
let $?R = \{R\ \#\ p\ |p.\ p \in pos\ \varphi 2\}$
have $card\ (?L \cup ?R) = card\ ?L + card\ ?R$
using *card-seperate finite-pos* **by** *blast*
moreover have $\ldots = card\ (pos\ \varphi 1) + card\ (pos\ \varphi 2)$
by (*simp add*: *cons-inject finite-inj-comp-set finite-pos*)
moreover have $\ldots \geq card\ (vars\text{-}of\text{-}prop\ \varphi 1) + card\ (vars\text{-}of\text{-}prop\ \varphi 2)$ **using** *IH1 IH2* **by** *arith*
then have $\ldots \geq card\ (vars\text{-}of\text{-}prop\ \varphi 1 \cup vars\text{-}of\text{-}prop\ \varphi 2)$ **using** *card-Un-le le-trans* **by** *blast*
ultimately
show $card\ (vars\text{-}of\text{-}prop\ \varphi 1 \cup vars\text{-}of\text{-}prop\ \varphi 2) \leq Suc\ (card\ (?L \cup ?R))$
$card\ (vars\text{-}of\text{-}prop\ \varphi 1 \cup vars\text{-}of\text{-}prop\ \varphi 2) \leq Suc\ (card\ (?L \cup ?R))$
$card\ (vars\text{-}of\text{-}prop\ \varphi 1 \cup vars\text{-}of\text{-}prop\ \varphi 2) \leq Suc\ (card\ (?L \cup ?R))$

$card\ (vars\text{-}of\text{-}prop\ \varphi 1 \cup vars\text{-}of\text{-}prop\ \varphi 2) \leq Suc\ (card\ (?L \cup ?R))$
  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* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* **where**
replace-at [] - $\psi = \psi$ |
replace-at $(L \# l)$ $(FAnd\ \varphi\ \varphi')$ $\psi = FAnd\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$|
replace-at $(R \# l)$ $(FAnd\ \varphi\ \varphi')$ $\psi = FAnd\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
replace-at $(L \# l)$ $(FOr\ \varphi\ \varphi')$ $\psi = FOr\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$ |
replace-at $(R \# l)$ $(FOr\ \varphi\ \varphi')$ $\psi = FOr\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
replace-at $(L \# l)$ $(FEq\ \varphi\ \varphi')$ $\psi = FEq\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$|
replace-at $(R \# l)$ $(FEq\ \varphi\ \varphi')$ $\psi = FEq\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
replace-at $(L \# l)$ $(FImp\ \varphi\ \varphi')$ $\psi = FImp\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$|
replace-at $(R \# l)$ $(FImp\ \varphi\ \varphi')$ $\psi = FImp\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
replace-at $(L \# l)$ $(FNot\ \varphi)$ $\psi = FNot\ (replace\text{-}at\ l\ \varphi\ \psi)$

3.2 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* :: $('v \Rightarrow bool) \Rightarrow {}'v\ propo \Rightarrow bool$ (**infix** $\models$ *50*) **where**
$\mathcal{A} \models FT = True$ |
$\mathcal{A} \models FF = False$ |
$\mathcal{A} \models FVar\ v = (\mathcal{A}\ v)$ |
$\mathcal{A} \models FNot\ \varphi = (\neg(\mathcal{A} \models \varphi))$ |
$\mathcal{A} \models FAnd\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \wedge \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FOr\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \vee \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FImp\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FEq\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \longleftrightarrow \mathcal{A} \models \varphi_2)$

definition *evalf* (**infix** $\models f$ *50*) **where**
evalf $\varphi\ \psi = (\forall A.\ A \models \varphi \longrightarrow A \models \psi)$

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

theorem *deduction-theorem*:
  $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
proof
  assume H: $\varphi \models f\ \psi$
  {
    fix A
    have $A \models FImp\ \varphi\ \psi$
      proof (*cases* $A \models \varphi$)
        case *True*
        then have $A \models \psi$ **using** H **unfolding** *evalf-def* **by** *metis*
        then show $A \models FImp\ \varphi\ \psi$ **by** *auto*
      next
        case *False*
        then show $A \models FImp\ \varphi\ \psi$ **by** *auto*
      qed
  }
  then show $\forall A.\ A \models FImp\ \varphi\ \psi$ **by** *blast*
next
  assume A: $\forall A.\ A \models FImp\ \varphi\ \psi$
  show $\varphi \models f\ \psi$
    proof (*rule ccontr*)
      assume $\neg\ \varphi \models f\ \psi$
      then obtain A **where** $A \models \varphi$ **and** $\neg\ A \models \psi$ **using** *evalf-def* **by** *metis*

then have $\neg$ $A \models FImp\ \varphi\ \psi$ **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
theory *Prop-Abstract-Transformation*
imports *Main Prop-Logic Wellfounded-More*

begin

This file is devoted to abstract properties of the transformations, like consistency preservation and lifting from terms to proposition.

3.3 Rewrite systems and properties

3.3.1 Lifting of rewrite rules

We can lift a rewrite relation r over a full1 formula: the relation r works on terms, while *propo-rew-step* works on formulas.

inductive *propo-rew-step* :: $('v\ propo \Rightarrow 'v\ propo \Rightarrow bool) \Rightarrow 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$
for $r :: 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$ **where**
global-rel: $r\ \varphi\ \psi \Longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \psi$ |
propo-rew-one-step-lift: $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow wf\text{-}conn\ c\ (\psi s\ @\ \varphi\ \#\ \psi s')$
$\Longrightarrow propo\text{-}rew\text{-}step\ r\ (conn\ c\ (\psi s\ @\ \varphi\ \#\ \psi s'))\ (conn\ c\ (\psi s\ @\ \varphi' \#\ \psi s'))$

Here is a more precise link between the lifting and the subformulas: if a rewriting takes place between φ and φ', then there are two subformulas ψ in φ and ψ' in φ', ψ' is the result of the rewriting of r on ψ.

This lemma is only a health condition:

lemma *propo-rew-step-subformula-imp*:
shows $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$
apply (*induct rule*: *propo-rew-step.induct*)
using *subformula.simps subformula-into-subformula* **apply** *blast*
using *wf-conn-no-arity-change subformula-into-subformula wf-conn-no-arity-change-helper in-set-conv-decomp* **by** *metis*

The converse is moreover true: if there is a ψ and ψ', then every formula φ containing ψ, can be rewritten into a formula φ', such that it contains φ'.

lemma *propo-rew-step-subformula-rec*:
  fixes $\psi\ \psi'\ \varphi$ :: *'v propo*
  shows $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists\varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \varphi\ \varphi')$
proof (*induct* φ *rule*: *subformula.induct*)
  case *subformula-refl*
  then have *propo-rew-step* $r\ \psi\ \psi'$ **using** *propo-rew-step.intros* **by** *auto*
  moreover have $\psi' \preceq \psi'$ **using** *Prop-Logic.subformula-refl* **by** *auto*
  ultimately show $\exists\varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \psi\ \varphi'$ **by** *fastforce*
next
  case (*subformula-into-subformula* ψ'' l c)
  note *IH* = *this*(*4*) **and** r = *this*(*5*) **and** ψ'' = *this*(*1*) **and** *wf* = *this*(*2*) **and** *incl* = *this*(*3*)
  then obtain φ' **where** $*$: $\psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \psi''\ \varphi'$ **by** *metis*
  moreover obtain $\xi\ \xi'$ :: *'v propo list* **where**
    l: $l = \xi\ @\ \psi''\ \#\ \xi'$ **using** *List.split-list* ψ'' **by** *metis*
  ultimately have *propo-rew-step* r (*conn* c l) (*conn* c $(\xi\ @\ \varphi'\ \#\ \xi')$)
    using *propo-rew-step.intros*(*2*) *wf* **by** *metis*
  moreover have $\psi' \preceq$ *conn* c $(\xi\ @\ \varphi'\ \#\ \xi')$
    using *wf* $*$ *wf-conn-no-arity-change Prop-Logic.subformula-into-subformula*
    by (*metis* (*no-types*) *in-set-conv-decomp* l *wf-conn-no-arity-change-helper*)
  ultimately show $\exists\varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* r (*conn* c l) φ' **by** *metis*
qed

lemma *propo-rew-step-subformula*:
  $(\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi') \longleftrightarrow (\exists\varphi'.$ *propo-rew-step* $r\ \varphi\ \varphi')$
  using *propo-rew-step-subformula-imp propo-rew-step-subformula-rec* **by** *metis+*

lemma *consistency-decompose-into-list*:
  assumes *wf*: *wf-conn* c l **and** *wf'*: *wf-conn* c l'
  and *same*: $\forall n.\ A \models l\ !\ n \longleftrightarrow (A \models l'\ !\ n)$
  shows $A \models$ *conn* c $l \longleftrightarrow A \models$ *conn* c l'
proof (*cases* c *rule*: *connective-cases-arity-2*)
  case *nullary*
  then show $(A \models$ *conn* c $l) \longleftrightarrow (A \models$ *conn* c $l')$ **using** *wf wf'* **by** *auto*
next
  case *unary* **note** c = *this*
  then obtain a **where** l: $l = [a]$ **using** *wf-conn-Not-decomp wf* **by** *metis*
  obtain a' **where** l': $l' = [a']$ **using** *wf-conn-Not-decomp wf'* c **by** *metis*
  have $A \models a \longleftrightarrow A \models a'$ **using** l l' **by** (*metis nth-Cons-0 same*)
  then show $A \models$ *conn* c $l \longleftrightarrow A \models$ *conn* c l' **using** l l' c **by** *auto*
next
  case *binary* **note** c = *this*
  then obtain a b **where** l: $l = [a, b]$
    using *wf-conn-bin-list-length list-length2-decomp wf* **by** *metis*
  obtain a' b' **where** l': $l' = [a', b']$
    using *wf-conn-bin-list-length list-length2-decomp wf'* c **by** *metis*

  have p: $A \models a \longleftrightarrow A \models a'$ $A \models b \longleftrightarrow A \models b'$
    using l l' *same* **by** (*metis diff-Suc-1 nth-Cons' nat.distinct*(*2*))+
  show $A \models$ *conn* c $l \longleftrightarrow A \models$ *conn* c l'
    using *wf* c p **unfolding** *binary-connectives-def* l l' **by** *auto*
qed

Relation between *propo-rew-step* and the rewriting we have seen before: *propo-rew-step* $r\ \varphi\ \varphi'$ means that we rewrite ψ inside φ (ie at a path p) into ψ'.

lemma *propo-rew-step-rewrite*:
  fixes $\varphi\ \varphi'$ :: *'v propo* **and** r :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool*

assumes *propo-rew-step r* φ φ'
shows $\exists\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* $p\ \varphi\ \psi \wedge$ *replace-at* $p\ \varphi\ \psi' = \varphi'$
using *assms*
proof (*induct rule*: *propo-rew-step.induct*)
case(*global-rel* φ ψ)
moreover have *path-to* [] φ φ **by** *auto*
moreover have *replace-at* [] $\varphi\ \psi = \psi$ **by** *auto*
ultimately show *?case* **by** *metis*
next
case (*propo-rew-one-step-lift* $\varphi\ \varphi'\ c\ \xi\ \xi'$) **note** *rel* = *this*(*1*) **and** *IH0* = *this*(*2*) **and** *corr* = *this*(*3*)
obtain $\psi\ \psi'\ p$ **where** *IH*: $r\ \psi\ \psi' \wedge$ *path-to* $p\ \varphi\ \psi \wedge$ *replace-at* $p\ \varphi\ \psi' = \varphi'$ **using** *IH0* **by** *metis*

{
fix $x :: {'v}$
assume $c = CT \vee c = CF \vee c = CVar\ x$
then have *False* **using** *corr* **by** *auto*
then have $\exists\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* p (*conn* c (ξ@ (φ # ξ'))) ψ
$\wedge$ *replace-at* p (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
by *fast*
}
moreover {
assume c: c = *CNot*
then have *empty*: ξ =[] ξ'=[] **using** *corr* **by** *auto*
have *path-to* (*L*#*p*) (*conn* c (ξ@ (φ # ξ'))) ψ
using *c empty IH wf-conn-unary path-to-l* **by** *fastforce*
moreover have *replace-at* (*L*#*p*) (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *c empty IH* **by** *auto*
ultimately have $\exists\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* p (*conn* c (ξ@ (φ # ξ'))) ψ
$\wedge$ *replace-at* p (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *IH* **by** *metis*
}
moreover {
assume c: $c \in$ *binary-connectives*
have *length* (ξ@ φ # ξ') = *2* **using** *wf-conn-bin-list-length corr c* **by** *metis*
then have *length* ξ + *length* ξ' = *1* **by** *auto*
then have *ld*: (*length* ξ = *1* $\wedge$ *length* ξ' = *0*) $\vee$ (*length* ξ = *0* $\wedge$ *length* ξ' = *1*) **by** *arith*
obtain *a b* **where** *ab*: (ξ=[] $\wedge$ ξ'=[*b*]) $\vee$ (ξ=[*a*] $\wedge$ ξ'=[])
using *ld* **by** (*case-tac* ξ, *case-tac* ξ', *auto*)
{
assume φ: ξ=[] $\wedge$ ξ'=[*b*]
have *path-to* (*L*#*p*) (*conn* c (ξ@ (φ # ξ'))) ψ
using φ *c IH ab corr* **by** (*simp add*: *path-to-l*)
moreover have *replace-at* (*L*#*p*) (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *c IH ab* φ **unfolding** *binary-connectives-def* **by** *auto*
ultimately have $\exists\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* p (*conn* c (ξ@ (φ # ξ'))) ψ
$\wedge$ *replace-at* p (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *IH* **by** *metis*
}
moreover {
assume φ: ξ=[*a*] ξ'=[]
then have *path-to* (*R*#*p*) (*conn* c (ξ@ (φ # ξ'))) ψ
using *c IH corr path-to-r corr* φ **by** (*simp add*: *path-to-r*)
moreover have *replace-at* (*R*#*p*) (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *c IH ab* φ **unfolding** *binary-connectives-def* **by** *auto*
ultimately have *?case* **using** *IH* **by** *metis*
}

ultimately have *?case* **using** *ab* **by** *blast*
}
ultimately show *?case* **using** *connective-cases-arity* **by** *blast*
qed

3.3.2 Consistency preservation

We define *preserves-un-sat*: it means that a relation preserves consistency.

definition *preserves-un-sat* **where**
$\textit{preserves-un-sat } r \longleftrightarrow (\forall \varphi\ \psi.\ r\ \varphi\ \psi \longrightarrow (\forall A.\ A \models \varphi \longleftrightarrow A \models \psi))$

lemma *propo-rew-step-preservers-val-explicit*:
$\textit{propo-rew-step } r\ \varphi\ \psi \Longrightarrow \textit{preserves-un-sat } r \Longrightarrow \textit{propo-rew-step } r\ \varphi\ \psi \Longrightarrow (\forall A.\ A \models \varphi \longleftrightarrow A \models \psi)$
unfolding *preserves-un-sat-def*
proof (*induction rule*: *propo-rew-step.induct*)
case *global-rel*
then show *?case* **by** *simp*
next
case ($\textit{propo-rew-one-step-lift}\ \varphi\ \varphi'\ c\ \xi\ \xi'$) **note** $\textit{rel} = \textit{this}(1)$ **and** $\textit{wf} = \textit{this}(2)$
and $\textit{IH} = \textit{this}(3)[\textit{OF this}(4)\ \textit{this}(1)]$ **and** $\textit{consistent} = \textit{this}(4)$
{
fix A
from *IH* **have** $\forall n.\ (A \models (\xi\ @\ \varphi\ \#\ \xi')\ !\ n) = (A \models (\xi\ @\ \varphi'\ \#\ \xi')\ !\ n)$
by (*metis* (*mono-tags*, *hide-lams*) *list-update-length nth-Cons-0 nth-append-length-plus nth-list-update-neq*)
then have $(A \models \textit{conn } c\ (\xi\ @\ \varphi\ \#\ \xi')) = (A \models \textit{conn } c\ (\xi\ @\ \varphi'\ \#\ \xi'))$
by (*meson consistency-decompose-into-list wf wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
}
then show $\forall A.\ A \models \textit{conn } c\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow A \models \textit{conn } c\ (\xi\ @\ \varphi'\ \#\ \xi')$ **by** *auto*
qed

lemma *propo-rew-step-preservers-val'*:
assumes *preserves-un-sat r*
shows *preserves-un-sat* (*propo-rew-step r*)
using *assms* **by** (*simp add*: *preserves-un-sat-def propo-rew-step-preservers-val-explicit*)

lemma *preserves-un-sat-OO*[*intro*]:
$\textit{preserves-un-sat } f \Longrightarrow \textit{preserves-un-sat } g \Longrightarrow \textit{preserves-un-sat } (f\ OO\ g)$
unfolding *preserves-un-sat-def* **by** *auto*

lemma *star-consistency-preservation-explicit*:
assumes $(\textit{propo-rew-step } r)^{\wedge**}\ \varphi\ \psi$ **and** *preserves-un-sat r*
shows $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
using *assms* **by** (*induct rule*: *rtranclp-induct*)
(*auto simp add*: *propo-rew-step-preservers-val-explicit*)

lemma *star-consistency-preservation*:
$\textit{preserves-un-sat } r \Longrightarrow \textit{preserves-un-sat } (\textit{propo-rew-step } r)^{\wedge**}$
by (*simp add*: *star-consistency-preservation-explicit preserves-un-sat-def*)

3.3.3 Full Lifting

In the previous a relation was lifted to a formula, now we define the relation such it is applied as long as possible. The definition is thus simply: it can be derived and nothing more can be derived.

lemma *full-ropo-rew-step-preservers-val*[*simp*]:
preserves-un-sat r $\Longrightarrow$ *preserves-un-sat* (*full* (*propo-rew-step r*))
by (*metis full-def preserves-un-sat-def star-consistency-preservation*)

lemma *full-propo-rew-step-subformula*:
full (*propo-rew-step r*) φ' φ $\Longrightarrow$ $\neg(\exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi')$
unfolding *full-def* **using** *propo-rew-step-subformula-rec* **by** *metis*

3.4 Transformation testing

3.4.1 Definition and first properties

To prove correctness of our transformation, we create a *all-subformula-st* predicate. It tests recursively all subformulas. At each step, the actual formula is tested. The aim of this *test-symb* function is to test locally some properties of the formulas (i.e. at the level of the connective or at first level). This allows a clause description between the rewrite relation and the *test-symb*

definition *all-subformula-st* :: ($'a$ *propo* $\Rightarrow$ *bool*) $\Rightarrow$ $'a$ *propo* $\Rightarrow$ *bool* **where**
all-subformula-st test-symb $\varphi \equiv \forall\psi.\ \psi \preceq \varphi \longrightarrow$ *test-symb* ψ

lemma *test-symb-imp-all-subformula-st*[*simp*]:
test-symb FT $\Longrightarrow$ *all-subformula-st test-symb FT*
test-symb FF $\Longrightarrow$ *all-subformula-st test-symb FF*
test-symb (*FVar x*) $\Longrightarrow$ *all-subformula-st test-symb* (*FVar x*)
unfolding *all-subformula-st-def* **using** *subformula-leaf* **by** *metis+*

lemma *all-subformula-st-test-symb-true-phi*:
all-subformula-st test-symb φ $\Longrightarrow$ *test-symb* φ
unfolding *all-subformula-st-def* **by** *auto*

lemma *all-subformula-st-decomp-imp*:
wf-conn c l $\Longrightarrow$ (*test-symb* (*conn c l*) $\wedge$ ($\forall\varphi\in$ *set l*. *all-subformula-st test-symb* φ))
$\Longrightarrow$ *all-subformula-st test-symb* (*conn c l*)
unfolding *all-subformula-st-def* **by** *auto*

To ease the finding of proofs, we give some explicit theorem about the decomposition.

lemma *all-subformula-st-decomp-rec*:
all-subformula-st test-symb (*conn c l*) $\Longrightarrow$ *wf-conn c l*
$\Longrightarrow$ (*test-symb* (*conn c l*) $\wedge$ ($\forall\varphi\in$ *set l*. *all-subformula-st test-symb* φ))
unfolding *all-subformula-st-def* **by** *auto*

lemma *all-subformula-st-decomp*:
fixes *c* :: $'v$ *connective* **and** *l* :: $'v$ *propo list*
assumes *wf-conn c l*
shows *all-subformula-st test-symb* (*conn c l*)
$\longleftrightarrow$ (*test-symb* (*conn c l*) $\wedge$ ($\forall\varphi\in$ *set l*. *all-subformula-st test-symb* φ))
using *assms all-subformula-st-decomp-rec all-subformula-st-decomp-imp* **by** *metis*

lemma *helper-fact*: $c \in$ *binary-connectives* $\longleftrightarrow$ (c = *COr* $\vee$ c = *CAnd* $\vee$ c = *CEq* $\vee$ c = *CImp*)
  unfolding *binary-connectives-def* **by** *auto*
lemma *all-subformula-st-decomp-explicit*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows *all-subformula-st test-symb* (*FAnd* φ ψ)
     $\longleftrightarrow$ (*test-symb* (*FAnd* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
  and *all-subformula-st test-symb* (*FOr* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FOr* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
  and *all-subformula-st test-symb* (*FNot* φ)
    $\longleftrightarrow$ (*test-symb* (*FNot* φ) $\wedge$ *all-subformula-st test-symb* φ)
  and *all-subformula-st test-symb* (*FEq* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FEq* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
  and *all-subformula-st test-symb* (*FImp* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FImp* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
proof −
  have *all-subformula-st test-symb* (*FAnd* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CAnd* [φ, ψ])
    by *auto*
  moreover have ... $\longleftrightarrow$*test-symb* (*conn CAnd* [φ, ψ])$\wedge$($\forall\xi\in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ)
    using *all-subformula-st-decomp wf-conn-helper-facts*(*5*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FAnd* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FAnd* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FOr* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn COr* [φ, ψ])
    by *auto*
  moreover have ... $\longleftrightarrow$
    (*test-symb* (*conn COr* [φ, ψ]) $\wedge$ ($\forall\xi\in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*6*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FOr* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FOr* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FEq* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CEq* [φ, ψ])
    by *auto*
  moreover have ...
    $\longleftrightarrow$ (*test-symb* (*conn CEq* [φ, ψ]) $\wedge$ ($\forall\xi\in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*8*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FEq* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FEq* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FImp* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CImp* [φ, ψ])
    by *auto*
  moreover have ...
    $\longleftrightarrow$(*test-symb* (*conn CImp* [φ, ψ]) $\wedge$ ($\forall\xi\in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*7*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FImp* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FImp* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FNot* φ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CNot* [φ])
    by *auto*
  moreover have ... = (*test-symb* (*conn CNot* [φ]) $\wedge$ ($\forall\xi\in$ *set* [φ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*1*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FNot* φ)

$\longleftrightarrow$ (*test-symb* (*FNot* φ) $\wedge$ *all-subformula-st test-symb* φ) **by** *simp*
qed

As *all-subformula-st* tests recursively, the function is true on every subformula.

lemma *subformula-all-subformula-st*:
$\psi \preceq \varphi \Longrightarrow$ *all-subformula-st test-symb* $\varphi \Longrightarrow$ *all-subformula-st test-symb* ψ
by (*induct rule*: *subformula.induct*, *auto simp add*: *all-subformula-st-decomp*)

The following theorem *no-test-symb-step-exists* shows the link between the *test-symb* function and the corresponding rewrite relation *r*: if we assume that if every time *test-symb* is true, then a *r* can be applied, finally as long as $\neg$ *all-subformula-st test-symb* φ, then something can be rewritten in φ.

lemma *no-test-symb-step-exists*:
fixes *r*:: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **and** *test-symb*:: *'v propo* $\Rightarrow$ *bool* **and** *x* :: *'v*
and φ :: *'v propo*
assumes
test-symb-false-nullary: $\forall x$. *test-symb FF* $\wedge$ *test-symb FT* $\wedge$ *test-symb* (*FVar x*) **and**
$\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg$*test-symb* $\varphi') \longrightarrow (\exists\ \psi.\ r\ \varphi'\ \psi)$ **and**
$\neg$ *all-subformula-st test-symb* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
using *assms*
proof (*induct* φ *rule*: *propo-induct-arity*)
case (*nullary* φ *x*)
then show $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
using *wf-conn-nullary test-symb-false-nullary* **by** *fastforce*
next
case (*unary* φ) **note** *IH* = *this*(*1*)[*OF this*(*2*)] **and** *r* = *this*(*2*) **and** *nst* = *this*(*3*) **and** *subf* = *this*(*4*)
from *r IH nst* **have** *H*: $\neg$ *all-subformula-st test-symb* $\varphi \Longrightarrow \exists \psi.\ \psi \preceq \varphi \wedge (\exists \psi'.\ r\ \psi\ \psi')$
by (*metis subformula-in-subformula-not subformula-refl subformula-trans*)
{
assume *n*: $\neg$*test-symb* (*FNot* φ)
obtain ψ **where** *r* (*FNot* φ) ψ **using** *subformula-refl r n nst* **by** *blast*
moreover have *FNot* $\varphi \preceq$ *FNot* φ **using** *subformula-refl* **by** *auto*
ultimately have $\exists \psi\ \psi'.\ \psi \preceq$ *FNot* $\varphi \wedge r\ \psi\ \psi'$ **by** *metis*
}
moreover {
assume *n*: *test-symb* (*FNot* φ)
then have $\neg$ *all-subformula-st test-symb* φ
using *all-subformula-st-decomp-explicit*(*3*) *nst subf* **by** *blast*
then have $\exists \psi\ \psi'.\ \psi \preceq$ *FNot* $\varphi \wedge r\ \psi\ \psi'$
using *H subformula-in-subformula-not subformula-refl subformula-trans* **by** *blast*
}
ultimately show $\exists \psi\ \psi'.\ \psi \preceq$ *FNot* $\varphi \wedge r\ \psi\ \psi'$ **by** *blast*
next
case (*binary* φ $\varphi 1$ $\varphi 2$)
note *IH*φ*1-0* = *this*(*1*)[*OF this*(*4*)] **and** *IH*φ*2-0* = *this*(*2*)[*OF this*(*4*)] **and** *r* = *this*(*4*)
and φ = *this*(*3*) **and** *le* = *this*(*5*) **and** *nst* = *this*(*6*)

obtain *c* :: *'v connective* **where**
c: (*c* = *CAnd* $\vee$ *c* = *COr* $\vee$ *c* = *CImp* $\vee$ *c* = *CEq*) $\wedge$ *conn c* [$\varphi 1$, $\varphi 2$] = φ
using φ **by** *fastforce*

then have *corr*: *wf-conn c* [$\varphi 1$, $\varphi 2$] **using** *wf-conn.simps* **unfolding** *binary-connectives-def* **by** *auto*
have *inc*: $\varphi 1 \preceq \varphi$ $\varphi 2 \preceq \varphi$ **using** *binary-connectives-def c subformula-in-binary-conn* **by** *blast+*

from *r IHφ1-0* **have** *IHφ1*: $\neg$ *all-subformula-st test-symb φ1* $\Longrightarrow \exists\psi\ \psi'.\ \psi \preceq \varphi 1 \wedge r\ \psi\ \psi'$
  using *inc*(*1*) *subformula-trans le* **by** *blast*
from *r IHφ2-0* **have** *IHφ2*: $\neg$ *all-subformula-st test-symb φ2* $\Longrightarrow \exists\psi.\ \psi \preceq \varphi 2 \wedge (\exists\psi'.\ r\ \psi\ \psi')$
  using *inc*(*2*) *subformula-trans le* **by** *blast*
have *cases*: $\neg$*test-symb φ* $\vee$ $\neg$*all-subformula-st test-symb φ1* $\vee$ $\neg$*all-subformula-st test-symb φ2*
  using *c nst* **by** *auto*
show $\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
  using *IHφ1 IHφ2 subformula-trans inc subformula-refl cases le* **by** *blast*
qed

3.4.2 Invariant conservation

If two rewrite relation are independant (or at least independant enough), then the property characterizing the first relation *all-subformula-st test-symb* remains true. The next show the same property, with changes in the assumptions.

The assumption $\forall\varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi' \longrightarrow$ *all-subformula-st test-symb* ψ means that rewriting with r does not mess up the property we want to preserve locally.

The previous assumption is not enough to go from r to *propo-rew-step* r: we have to add the assumption that rewriting inside does not mess up the term: $\forall c\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi' \longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi'\ \#\ \xi'))$

Invariant while lifting of the rewriting relation

The condition $\varphi \preceq \Phi$ (that will by used with $\Phi = \varphi$ most of the time) is here to ensure that the recursive conditions on Φ will moreover hold for the subterm we are rewriting. For example if there is no equivalence symbol in Φ, we do not have to care about equivalence symbols in the two previous assumptions.

lemma *propo-rew-step-inv-stay'*:
  fixes *r*:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
  and $\varphi\ \psi\ \Phi$:: *'v propo*
  assumes *H*: $\forall\varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* φ'
    $\longrightarrow$ *all-subformula-st test-symb* ψ
  and *H'*: $\forall$(*c*:: *'v connective*) $\xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi'$
    $\longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* φ'
    $\longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi'\ \#\ \xi'))$ **and**
    propo-rew-step $r\ \varphi\ \psi$ **and**
    $\varphi \preceq \Phi$ **and**
    all-subformula-st test-symb φ
  shows *all-subformula-st test-symb* ψ
  using *assms*(*3*−*5*)
proof (*induct rule*: *propo-rew-step.induct*)
  case *global-rel*
  then show *?case* **using** *H* **by** *simp*
next
  case (*propo-rew-one-step-lift* $\varphi\ \varphi'\ c\ \xi\ \xi'$)
  note *rel* = *this*(*1*) **and** φ = *this*(*2*) **and** *corr* = *this*(*3*) **and** Φ = *this*(*4*) **and** *nst* = *this*(*5*)
  have *sq*: $\varphi \preceq \Phi$
    using Φ *corr subformula-into-subformula subformula-refl subformula-trans*
    by (*metis in-set-conv-decomp*)
  from *corr* **have** $\forall\ \psi.\ \psi \in$ *set* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *all-subformula-st test-symb* ψ

using *all-subformula-st-decomp nst* **by** *blast*
then have $*$: $\forall\psi.\ \psi \in set\ (\xi\ @\ \varphi'\ \#\ \xi') \longrightarrow$ *all-subformula-st test-symb* ψ **using** φ *sq* **by** *fastforce*
then have *test-symb* φ' **using** *all-subformula-st-test-symb-true-phi* **by** *auto*
moreover from *corr nst* **have** *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$)
using *all-subformula-st-decomp* **by** *blast*
ultimately have *test-symb*: *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **using** H' *sq corr rel* **by** *blast*

have *wf-conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$
by (*metis wf-conn-no-arity-change-helper corr wf-conn-no-arity-change*)
then show *all-subformula-st test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$)
using $*$ *test-symb* **by** (*metis all-subformula-st-decomp*)
qed

The need for $\varphi \preceq \Phi$ is not always necessary, hence we moreover have a version without inclusion.

lemma *propo-rew-step-inv-stay*:
fixes *r*:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and $\varphi\ \psi$ :: $'v$ *propo*
assumes
H: $\forall\varphi'\ \psi.\ r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi' \longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall(c$:: $'v$ *connective*$)\ \xi\ \varphi\ \xi'\ \varphi'.$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$)
$\longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
propo-rew-step r $\varphi\ \psi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
using *propo-rew-step-inv-stay'*[*of* φ *r test-symb* $\varphi\ \psi$] *assms subformula-refl* **by** *metis*

The lemmas can be lifted to *propo-rew-step* $r^{\downarrow}$ instead of *propo-rew-step*

Invariant after all rewriting

lemma *full-propo-rew-step-inv-stay-with-inc*:
fixes *r*:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and $\varphi\ \psi$ :: $'v$ *propo*
assumes
H: $\forall\ \varphi\ \psi.$ *propo-rew-step r* $\varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall(c$:: $'v$ *connective*$)\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step r* $\varphi\ \varphi'$
$\longrightarrow$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
$\varphi \preceq \Phi$ **and**
full: *full* (*propo-rew-step r*) $\varphi\ \psi$ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
using *assms* **unfolding** *full-def*
proof $-$
have *rel*: (*propo-rew-step r*)** $\varphi\ \psi$
using *full* **unfolding** *full-def* **by** *auto*
then show *all-subformula-st test-symb* ψ
using *init*
proof (*induct rule*: *rtranclp-induct*)
case *base*
then show *all-subformula-st test-symb* φ **by** *blast*
next
case (*step b c*) **note** *star* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *one* = *this*(*2*) **and** *all* = *this*(*4*)
then have *all-subformula-st test-symb b* **by** *metis*
then show *all-subformula-st test-symb c* **using** *propo-rew-step-inv-stay'* $H\ H'$ *rel one* **by** *auto*

  qed
qed

lemma *full-propo-rew-step-inv-stay'*:
  fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
  and φ ψ :: $'v$ *propo*
  assumes
    H: $\forall$ φ ψ. *propo-rew-step* r φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ
      $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
    H': $\forall$ (c:: $'v$ *connective*) ξ φ ξ' φ'. *propo-rew-step* r φ φ' $\longrightarrow$ *wf-conn* c (ξ @ φ # ξ')
      $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ # ξ')) $\longrightarrow$ *test-symb* φ' $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ' # ξ')) **and**
    full: *full* (*propo-rew-step* r) φ ψ **and**
    init: *all-subformula-st test-symb* φ
  shows *all-subformula-st test-symb* ψ
  using *full-propo-rew-step-inv-stay-with-inc*[*of r test-symb* φ] *assms subformula-refl* **by** *metis*

lemma *full-propo-rew-step-inv-stay*:
  fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
  and φ ψ :: $'v$ *propo*
  assumes
    H: $\forall\varphi$ ψ. r φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
    H': $\forall$ (c:: $'v$ *connective*) ξ φ ξ' φ'. *wf-conn* c (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ # ξ'))
      $\longrightarrow$ *test-symb* φ' $\longrightarrow$ *test-symb* (*conn* c (ξ @ φ' # ξ')) **and**
    full: *full* (*propo-rew-step* r) φ ψ **and**
    init: *all-subformula-st test-symb* φ
  shows *all-subformula-st test-symb* ψ
  unfolding *full-def*
proof –
  have *rel*: (*propo-rew-step* r)^** φ ψ
    using *full* **unfolding** *full-def* **by** *auto*
  then show *all-subformula-st test-symb* ψ
    using *init*
    proof (*induct rule*: *rtranclp-induct*)
      case *base*
      then show *all-subformula-st test-symb* φ **by** *blast*
    next
      case (*step b c*)
      note *star* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *one* = *this*(*2*) **and** *all* = *this*(*4*)
      then have *all-subformula-st test-symb b* **by** *metis*
      then show *all-subformula-st test-symb c*
        using *propo-rew-step-inv-stay subformula-refl H H' rel one* **by** *auto*
    qed
qed

lemma *full-propo-rew-step-inv-stay-conn*:
  fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
  and φ ψ :: $'v$ *propo*
  assumes
    H: $\forall\varphi$ ψ. r φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
    H': $\forall$ (c:: $'v$ *connective*) l l'. *wf-conn* c l $\longrightarrow$ *wf-conn* c l'
      $\longrightarrow$ (*test-symb* (*conn* c l) $\longleftrightarrow$ *test-symb* (*conn* c l')) **and**
    full: *full* (*propo-rew-step* r) φ ψ **and**
    init: *all-subformula-st test-symb* φ
  shows *all-subformula-st test-symb* ψ
proof –

have $\bigwedge(c{::}\ 'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'.\ wf\text{-}conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')$
$\implies test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \implies test\text{-}symb\ \varphi' \implies test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$
using *H'* **by** (*metis wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
then show *all-subformula-st test-symb* ψ
using *H full init full-propo-rew-step-inv-stay* **by** *blast*
qed

end
theory *Prop-Normalisation*
imports *Main Prop-Logic Prop-Abstract-Transformation ../lib/Multiset-More*
begin

Given the previous definition about abstract rewriting and theorem about them, we now have the detailed rule making the transformation into CNF/DNF.

3.5 Rewrite Rules

The idea of Christoph Weidenbach's book is to remove gradually the operators: first equivalencies, then implication, after that the unused true/false and finally the reorganizing the or/and. We will prove each transformation seperately.

3.5.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

inductive *elim-equiv* :: *'v propo* ⇒ *'v propo* ⇒ *bool* **where**
elim-equiv[*simp*]: *elim-equiv* (*FEq* φ ψ) (*FAnd* (*FImp* φ ψ) (*FImp* ψ φ))

lemma *elim-equiv-transformation-consistent*:
$A \models FEq\ \varphi\ \psi \longleftrightarrow A \models FAnd\ (FImp\ \varphi\ \psi)\ (FImp\ \psi\ \varphi)$
by *auto*

lemma *elim-equiv-explicit*: $elim\text{-}equiv\ \varphi\ \psi \implies \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct rule*: *elim-equiv.induct*, *auto*)

lemma *elim-equiv-consistent*: *preserves-un-sat elim-equiv*
unfolding *preserves-un-sat-def* **by** (*simp add*: *elim-equiv-explicit*)

lemma *elimEquv-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step elim-equiv*))
by (*simp add*: *elim-equiv-consistent*)

This function ensures that there is no equivalencies left in the formula tested by *no-equiv-symb*.

fun *no-equiv-symb* :: *'v propo* ⇒ *bool* **where**
no-equiv-symb (*FEq* - -) = *False* |
no-equiv-symb - = *True*

Given the definition of *no-equiv-symb*, it does not depend on the formula, but only on the connective used.

lemma *no-equiv-symb-conn-characterization*[*simp*]:
fixes *c* :: *'v connective* **and** *l* :: *'v propo list*
assumes *wf*: *wf-conn c l*
shows *no-equiv-symb* (*conn c l*) $\longleftrightarrow c \neq CEq$

by (*metis connective.distinct*(*13*,*25*,*35*,*43*) *wf no-equiv-symb.elims*(*3*) *no-equiv-symb.simps*(*1*)
wf-conn.cases wf-conn-list(*6*))

definition *no-equiv* **where** *no-equiv* = *all-subformula-st no-equiv-symb*

lemma *no-equiv-eq*[*simp*]:
fixes φ ψ :: $'v$ *propo*
shows
$\neg$*no-equiv* (*FEq* φ ψ)
no-equiv FT
no-equiv FF
using *no-equiv-symb.simps*(*1*) *all-subformula-st-test-symb-true-phi* **unfolding** *no-equiv-def* **by** *auto*

The following lemma helps to reconstruct *no-equiv* expressions: this representation is easier to use than the set definition.

lemma *all-subformula-st-decomp-explicit-no-equiv*[*iff*]:
fixes φ ψ :: $'v$ *propo*
shows
no-equiv (*FNot* φ) $\longleftrightarrow$ *no-equiv* φ
no-equiv (*FAnd* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
no-equiv (*FOr* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
no-equiv (*FImp* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
by (*auto simp*: *no-equiv-def*)

A theorem to show the link between the rewrite relation *elim-equiv* and the function *no-equiv-symb*. This theorem is one of the assumption we need to characterize the transformation.

lemma *no-equiv-elim-equiv-step*:
fixes φ :: $'v$ *propo*
assumes *no-equiv*: $\neg$ *no-equiv* φ
shows $\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elim-equiv* $\psi\ \psi'$
proof −
have *test-symb-false-nullary*:
$\forall x$::$'v$. *no-equiv-symb FF* $\wedge$ *no-equiv-symb FT* $\wedge$ *no-equiv-symb* (*FVar* x)
unfolding *no-equiv-def* **by** *auto*
moreover {
fix c:: $'v$ *connective* **and** l :: $'v$ *propo list* **and** ψ :: $'v$ *propo*
assume *a1*: *elim-equiv* (*conn c l*) ψ
have $\bigwedge p\ pa.\ \neg$ *elim-equiv* (p::$'v$ *propo*) *pa* $\vee\ \neg$ *no-equiv-symb p*
using *elim-equiv.cases no-equiv-symb.simps*(*1*) **by** *blast*
then have *elim-equiv* (*conn c l*) $\psi \Longrightarrow \neg$*no-equiv-symb* (*conn c l*) **using** *a1* **by** *metis*
}
moreover have H': $\forall\psi.\ \neg$*elim-equiv FT* ψ $\forall\psi.\ \neg$*elim-equiv FF* ψ $\forall\psi\ x.\ \neg$*elim-equiv* (*FVar* x) ψ
using *elim-equiv.cases* **by** *auto*
moreover have $\bigwedge\varphi.\ \neg$ *no-equiv-symb* $\varphi \Longrightarrow \exists\psi.$ *elim-equiv* $\varphi\ \psi$
by (*case-tac* φ, *auto simp*: *elim-equiv.simps*)
then have $\bigwedge\varphi'.\ \varphi' \preceq \varphi \Longrightarrow \neg$*no-equiv-symb* $\varphi' \Longrightarrow\ \exists\psi.$ *elim-equiv* $\varphi'\ \psi$ **by** *force*
ultimately show *?thesis*
using *no-test-symb-step-exists no-equiv test-symb-false-nullary* **unfolding** *no-equiv-def* **by** *blast*
qed

Given all the previous theorem and the characterization, once we have rewritten everything, there is no equivalence symbol any more.

lemma *no-equiv-full-propo-rew-step-elim-equiv*:
full (*propo-rew-step elim-equiv*) $\varphi\ \psi \Longrightarrow$ *no-equiv* ψ
using *full-propo-rew-step-subformula no-equiv-elim-equiv-step* **by** *blast*

3.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: *'v propo* ⇒ *'v propo* ⇒ *bool* **where**
[*simp*]: *elim-imp* (*FImp* φ ψ) (*FOr* (*FNot* φ) ψ)

lemma *elim-imp-transformation-consistent*:
  $A \models$ *FImp* $\varphi\ \psi \longleftrightarrow A \models$ *FOr* (*FNot* φ) ψ
  by *auto*

lemma *elim-imp-explicit*: *elim-imp* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
  by (*induct* φ ψ *rule*: *elim-imp.induct*, *auto*)

lemma *elim-imp-consistent*: *preserves-un-sat elim-imp*
  unfolding *preserves-un-sat-def* **by** (*simp add*: *elim-imp-explicit*)

lemma *elim-imp-lifted-consistant*:
  preserves-un-sat (*full* (*propo-rew-step elim-imp*))
  by (*simp add*: *elim-imp-consistent*)

fun *no-imp-symb* **where**
no-imp-symb (*FImp* - -) = *False* |
no-imp-symb - = *True*

lemma *no-imp-symb-conn-characterization*:
  wf-conn c l $\Longrightarrow$ *no-imp-symb* (*conn c l*) $\longleftrightarrow c \neq$ *CImp*
  by (*induction rule*: *wf-conn-induct*) *auto*

definition *no-imp* **where** *no-imp* ≡ *all-subformula-st no-imp-symb*
declare *no-imp-def*[*simp*]

lemma *no-imp-Imp*[*simp*]:
  ¬*no-imp* (*FImp* φ ψ)
  no-imp FT
  no-imp FF
  unfolding *no-imp-def* **by** *auto*

lemma *all-subformula-st-decomp-explicit-imp*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows
    no-imp (*FNot* φ) $\longleftrightarrow$ *no-imp* φ
    no-imp (*FAnd* φ ψ) $\longleftrightarrow$ (*no-imp* φ ∧ *no-imp* ψ)
    no-imp (*FOr* φ ψ) $\longleftrightarrow$ (*no-imp* φ ∧ *no-imp* ψ)
  by *auto*

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
  elim-imp $\varphi\ \psi \Longrightarrow$ *no-equiv* $\varphi \Longrightarrow$ *no-equiv* ψ
  by (*induct* φ ψ *rule*: *elim-imp.induct*, *auto*)

lemma *elim-imp-inv*:
  fixes φ ψ :: *'v propo*
  assumes *full* (*propo-rew-step elim-imp*) φ ψ **and** *no-equiv* φ
  shows *no-equiv* ψ
  using *full-propo-rew-step-inv-stay-conn*[*of elim-imp no-equiv-symb* φ ψ] *assms elim-imp-no-equiv*

no-equiv-symb-conn-characterization **unfolding** *no-equiv-def* **by** *metis*

lemma *no-no-imp-elim-imp-step-exists*:
fixes φ :: *'v propo*
assumes *no-equiv*: $\neg$ *no-imp* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elim-imp* $\psi\ \psi'$
proof −
have *test-symb-false-nullary*: $\forall x.$ *no-imp-symb FF* $\wedge$ *no-imp-symb FT* $\wedge$ *no-imp-symb* (*FVar* (*x*:: *'v*))
by *auto*
moreover **{**
fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** ψ :: *'v propo*
have *H*: *elim-imp* (*conn c l*) $\psi \Longrightarrow \neg$*no-imp-symb* (*conn c l*)
by (*auto elim*: *elim-imp.cases*)
}
moreover
have *H'*: $\forall\psi.\ \neg$*elim-imp FT* ψ $\forall\psi.\ \neg$*elim-imp FF* ψ $\forall\psi\ x.\ \neg$*elim-imp* (*FVar x*) ψ
by (*auto elim*: *elim-imp.cases*)+
moreover
have $\bigwedge\varphi.\ \neg$ *no-imp-symb* $\varphi \Longrightarrow \exists\psi.$ *elim-imp* $\varphi\ \psi$
by (*case-tac* φ) (*force simp*: *elim-imp.simps*)+
then have $\bigwedge\varphi'.\ \varphi' \preceq \varphi \Longrightarrow \neg$*no-imp-symb* $\varphi' \Longrightarrow\ \exists\ \psi.$ *elim-imp* $\varphi'\ \psi$ **by** *force*
ultimately show *?thesis*
using *no-test-symb-step-exists no-equiv test-symb-false-nullary* **unfolding** *no-imp-def* **by** *blast*
qed

lemma *no-imp-full-propo-rew-step-elim-imp*: *full* (*propo-rew-step elim-imp*) $\varphi\ \psi \Longrightarrow$ *no-imp* ψ
using *full-propo-rew-step-subformula no-no-imp-elim-imp-step-exists* **by** *blast*

3.5.3 Eliminate all the True and False in the formula

Contrary to the book, we have to give the transformation and the "commutative" transformation. The latter is implicit in the book.

inductive *elimTB* **where**
ElimTB1: *elimTB* (*FAnd* φ *FT*) φ |
ElimTB1': *elimTB* (*FAnd FT* φ) φ |

ElimTB2: *elimTB* (*FAnd* φ *FF*) *FF* |
ElimTB2': *elimTB* (*FAnd FF* φ) *FF* |

ElimTB3: *elimTB* (*FOr* φ *FT*) *FT* |
ElimTB3': *elimTB* (*FOr FT* φ) *FT* |

ElimTB4: *elimTB* (*FOr* φ *FF*) φ |
ElimTB4': *elimTB* (*FOr FF* φ) φ |

ElimTB5: *elimTB* (*FNot FT*) *FF* |
ElimTB6: *elimTB* (*FNot FF*) *FT*

lemma *elimTB-consistent*: *preserves-un-sat elimTB*
proof −
{
fix $\varphi\ \psi$:: *'b propo*
have *elimTB* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$ **by** (*induction rule*: *elimTB.inducts*) *auto*
}

then show *?thesis* **using** *preserves-un-sat-def* **by** *auto*
qed

inductive *no-T-F-symb* :: *'v propo* $\Rightarrow$ *bool* **where**
no-T-F-symb-comp: $c \neq CF \Longrightarrow c \neq CT \Longrightarrow$ *wf-conn c l* $\Longrightarrow (\forall \varphi \in set\ l.\ \varphi \neq FT \wedge \varphi \neq FF)$
$\Longrightarrow$ *no-T-F-symb* (*conn c l*)

lemma *wf-conn-no-T-F-symb-iff*[*simp*]:
wf-conn c $\psi s \Longrightarrow$
no-T-F-symb (*conn c* ψs) $\longleftrightarrow$ $(c \neq CF \wedge c \neq CT \wedge (\forall \psi \in set\ \psi s.\ \psi \neq FF \wedge \psi \neq FT))$
unfolding *no-T-F-symb.simps* **apply** (*cases c*)
using *wf-conn-list*(*1*) **apply** *fastforce*
using *wf-conn-list*(*2*) **apply** *fastforce*
using *wf-conn-list*(*3*) **apply** *fastforce*
apply (*metis* (*no-types*, *hide-lams*) *conn-inj connective.distinct*(*5*,*17*))
using *conn-inj* **apply** *blast*+
done

lemma *wf-conn-no-T-F-symb-iff-explicit*[*simp*]:
no-T-F-symb (*FAnd* φ ψ) $\longleftrightarrow$ $(\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FOr* φ ψ) $\longleftrightarrow$ $(\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FEq* φ ψ) $\longleftrightarrow$ $(\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FImp* φ ψ) $\longleftrightarrow$ $(\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
apply (*metis conn.simps*(*36*) *conn.simps*(*37*) *conn.simps*(*5*) *propo.distinct*(*19*)
wf-conn-helper-facts(*5*) *wf-conn-no-T-F-symb-iff*)
apply (*metis conn.simps*(*36*) *conn.simps*(*37*) *conn.simps*(*6*) *propo.distinct*(*22*)
wf-conn-helper-facts(*6*) *wf-conn-no-T-F-symb-iff*)
using *wf-conn-no-T-F-symb-iff* **apply** *fastforce*
by (*metis conn.simps*(*36*) *conn.simps*(*37*) *conn.simps*(*7*) *propo.distinct*(*23*) *wf-conn-helper-facts*(*7*)
wf-conn-no-T-F-symb-iff)

lemma *no-T-F-symb-false*[*simp*]:
fixes *c* :: *'v connective*
shows
$\neg$*no-T-F-symb* (*FT* :: *'v propo*)
$\neg$*no-T-F-symb* (*FF* :: *'v propo*)
by (*metis* (*no-types*) *conn.simps*(*1*,*2*) *wf-conn-no-T-F-symb-iff wf-conn-nullary*)+

lemma *no-T-F-symb-bool*[*simp*]:
fixes *x* :: *'v*
shows *no-T-F-symb* (*FVar x*)
using *no-T-F-symb-comp wf-conn-nullary* **by** (*metis connective.distinct*(*3*, *15*) *conn.simps*(*3*)
empty-iff list.set(*1*))

lemma *no-T-F-symb-fnot-imp*:
$\neg$*no-T-F-symb* (*FNot* φ) $\Longrightarrow \varphi = FT \vee \varphi = FF$
proof (*rule ccontr*)
assume *n*: $\neg$ *no-T-F-symb* (*FNot* φ)
assume $\neg$ $(\varphi = FT \vee \varphi = FF)$
then have $\forall \varphi' \in set\ [\varphi].\ \varphi' \neq FT \wedge \varphi' \neq FF$ **by** *auto*
moreover have *wf-conn CNot* [φ] **by** *simp*
ultimately have *no-T-F-symb* (*FNot* φ)
using *no-T-F-symb.intros* **by** (*metis conn.simps*(*4*) *connective.distinct*(*5*,*17*))

then show *False* **using** *n* **by** *blast*
qed

lemma *no-T-F-symb-fnot*[*simp*]:
no-T-F-symb (*FNot* φ) $\longleftrightarrow$ $\neg(\varphi = FT \vee \varphi = FF)$
using *no-T-F-symb.simps no-T-F-symb-fnot-imp* **by** (*metis conn-inj-not*(*2*) *list.set-intros*(*1*))

Actually it is not possible to remover every FT and FF: if the formula is equal to true or false, we can not remove it.

inductive *no-T-F-symb-except-toplevel* **where**
no-T-F-symb-except-toplevel-true[*simp*]: *no-T-F-symb-except-toplevel FT* |
no-T-F-symb-except-toplevel-false[*simp*]: *no-T-F-symb-except-toplevel FF* |
noTrue-no-T-F-symb-except-toplevel[*simp*]: *no-T-F-symb* φ $\Longrightarrow$ *no-T-F-symb-except-toplevel* φ

lemma *no-T-F-symb-except-toplevel-bool*:
fixes *x* :: *'v*
shows *no-T-F-symb-except-toplevel* (*FVar x*)
by *simp*

lemma *no-T-F-symb-except-toplevel-not-decom*:
$\varphi \neq FT \Longrightarrow \varphi \neq FF \Longrightarrow$ *no-T-F-symb-except-toplevel* (*FNot* φ)
by *simp*

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes φ ψ :: *'v propo*
assumes $\varphi \neq FT$ **and** $\varphi \neq FF$ **and** $\psi \neq FT$ **and** $\psi \neq FF$
and *c*: *c*$\in$ *binary-connectives*
shows *no-T-F-symb-except-toplevel* (*conn c* [φ, ψ])
by (*metis* (*no-types*, *lifting*) *assms c conn.simps*(*4*) *list.discI noTrue-no-T-F-symb-except-toplevel wf-conn-no-T-F-symb-iff no-T-F-symb-fnot set-ConsD wf-conn-binary wf-conn-helper-facts*(*1*) *wf-conn-list-decomp*(*1*,*2*))

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:
fixes *l* :: *'v propo list* **and** *c* :: *'v connective*
assumes *corr*: *wf-conn c l*
and $FT \in$ *set l* $\vee$ $FF \in$ *set l*
shows $\neg$*no-T-F-symb-except-toplevel* (*conn c l*)
by (*metis assms empty-iff no-T-F-symb-except-toplevel.simps wf-conn-no-T-F-symb-iff set-empty wf-conn-list*(*1*,*2*))

lemma *no-T-F-symb-except-top-level-false-example*[*simp*]:
fixes φ ψ :: *'v propo*
assumes $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FAnd* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FOr* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FImp* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FEq* φ ψ)
using *assms no-T-F-symb-except-toplevel-if-is-a-true-false* **unfolding** *binary-connectives-def*
by (*metis* (*no-types*) *conn.simps*(*5*−*8*) *insert-iff list.simps*(*14*−*15*) *wf-conn-helper-facts*(*5*−*8*))+

lemma *no-T-F-symb-except-top-level-false-not*[*simp*]:
fixes φ ψ :: *'v propo*
assumes $\varphi = FT \vee \varphi = FF$
shows

$\neg$ *no-T-F-symb-except-toplevel* (*FNot* φ)
by (*simp add*: *assms no-T-F-symb-except-toplevel.simps*)

This is the local extension of *no-T-F-symb-except-toplevel*.

definition *no-T-F-except-top-level* **where**
no-T-F-except-top-level $\equiv$ *all-subformula-st no-T-F-symb-except-toplevel*

This is another property we will use. While this version might seem to be the one we want to prove, it is not since *FT* can not be reduced.

definition *no-T-F* **where**
no-T-F $\equiv$ *all-subformula-st no-T-F-symb*

lemma *no-T-F-except-top-level-false*:
fixes l :: $'v$ *propo list* **and** c :: $'v$ *connective*
assumes *wf-conn c l*
and *FT* $\in$ *set l* $\vee$ *FF* $\in$ *set l*
shows $\neg$*no-T-F-except-top-level* (*conn c l*)
by (*simp add*: *all-subformula-st-decomp assms no-T-F-except-top-level-def*
no-T-F-symb-except-toplevel-if-is-a-true-false)

lemma *no-T-F-except-top-level-false-example*[*simp*]:
fixes φ ψ :: $'v$ *propo*
assumes φ = *FT* $\vee$ ψ = *FT* $\vee$ φ = *FF* $\vee$ ψ = *FF*
shows
$\neg$*no-T-F-except-top-level* (*FAnd* φ ψ)
$\neg$*no-T-F-except-top-level* (*FOr* φ ψ)
$\neg$*no-T-F-except-top-level* (*FEq* φ ψ)
$\neg$*no-T-F-except-top-level* (*FImp* φ ψ)
by (*metis all-subformula-st-test-symb-true-phi assms no-T-F-except-top-level-def*
no-T-F-symb-except-top-level-false-example)+

lemma *no-T-F-symb-except-toplevel-no-T-F-symb*:
no-T-F-symb-except-toplevel $\varphi \Longrightarrow \varphi \neq$ *FF* $\Longrightarrow \varphi \neq$ *FT* $\Longrightarrow$ *no-T-F-symb* φ
by (*induct rule*: *no-T-F-symb-except-toplevel.induct*, *auto*)

The two following lemmas give the precise link between the two definitions.

lemma *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*:
no-T-F-except-top-level $\varphi \Longrightarrow \varphi \neq$ *FF* $\Longrightarrow \varphi \neq$ *FT* $\Longrightarrow$ *no-T-F* φ
unfolding *no-T-F-except-top-level-def no-T-F-def* **apply** (*induct* φ)
using *no-T-F-symb-fnot* **by** *fastforce*+

lemma *no-T-F-no-T-F-except-top-level*:
no-T-F $\varphi \Longrightarrow$ *no-T-F-except-top-level* φ
unfolding *no-T-F-except-top-level-def no-T-F-def*
unfolding *all-subformula-st-def* **by** *auto*

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level FF no-T-F-except-top-level FT*
unfolding *no-T-F-except-top-level-def* **by** *auto*

lemma *no-T-F-no-T-F-except-top-level'*[*simp*]:
no-T-F-except-top-level $\varphi \longleftrightarrow (\varphi$ = *FF* $\vee$ φ = *FT* $\vee$ *no-T-F* $\varphi)$
using *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-no-T-F-except-top-level*
by *auto*

lemma *no-T-F-bin-decomp*[*simp*]:
  assumes *c*: $c \in$ *binary-connectives*
  shows *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
proof −
  have *wf*: *wf-conn c* [φ, ψ] **using** *c* **by** *auto*
  then have *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F-symb* (*conn c* [φ, ψ]) $\wedge$ *no-T-F* φ $\wedge$ *no-T-F* ψ)
    by (*simp add*: *all-subformula-st-decomp no-T-F-def*)
  then show *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
    using *c wf all-subformula-st-decomp list.discI no-T-F-def no-T-F-symb-except-toplevel-bin-decom*
      no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(*1*,*2*) *wf-conn-helper-facts*(*2*,*3*)
      wf-conn-list(*1*,*2*) **by** *metis*
qed

lemma *no-T-F-bin-decomp-expanded*[*simp*]:
  assumes *c*: $c = CAnd \vee c = COr \vee c = CEq \vee c = CImp$
  shows *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
  using *no-T-F-bin-decomp assms* **unfolding** *binary-connectives-def* **by** *blast*

lemma *no-T-F-comp-expanded-explicit*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows
    no-T-F (*FAnd* φ ψ) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
    no-T-F (*FOr* φ ψ) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
    no-T-F (*FEq* φ ψ) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
    no-T-F (*FImp* φ ψ) $\longleftrightarrow$ (*no-T-F* φ $\wedge$ *no-T-F* ψ)
  using *assms conn.simps*(*5*−*8*) *no-T-F-bin-decomp-expanded* **by** (*metis* (*no-types*))+

lemma *no-T-F-comp-not*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows *no-T-F* (*FNot* φ) $\longleftrightarrow$ *no-T-F* φ
  by (*metis all-subformula-st-decomp-explicit*(*3*) *all-subformula-st-test-symb-true-phi no-T-F-def*
    no-T-F-symb-false(*1*,*2*) *no-T-F-symb-fnot-imp*)

lemma *no-T-F-decomp*:
  fixes φ ψ :: *'v propo*
  assumes φ: *no-T-F* (*FAnd* φ ψ) $\vee$ *no-T-F* (*FOr* φ ψ) $\vee$ *no-T-F* (*FEq* φ ψ) $\vee$ *no-T-F* (*FImp* φ ψ)
  shows *no-T-F* ψ **and** *no-T-F* φ
  using *assms* **by** *auto*

lemma *no-T-F-decomp-not*:
  fixes φ :: *'v propo*
  assumes φ: *no-T-F* (*FNot* φ)
  shows *no-T-F* φ
  using *assms* **by** *auto*

lemma *no-T-F-symb-except-toplevel-step-exists*:
  fixes φ ψ :: *'v propo*
  assumes *no-equiv* φ **and** *no-imp* φ
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'$. *elimTB* ψ ψ'
proof (*induct* ψ *rule*: *propo-induct-arity*)
  case (*nullary* φ' *x*)
  then have *False* **using** *no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false* **by** *auto*
  then show *?case* **by** *blast*
next
  case (*unary* ψ)
  then have $\psi = FF \vee \psi = FT$ **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*

then show *?case* **using** *ElimTB5 ElimTB6* **by** *blast*
next
case (*binary* φ' *ψ1 ψ2*)
note *IH1* = *this*(*1*) **and** *IH2* = *this*(*2*) **and** φ' = *this*(*3*) **and** *Fφ* = *this*(*4*) **and** *n* = *this*(*5*)
{
assume φ' = *FImp ψ1 ψ2* ∨ φ' = *FEq ψ1 ψ2*
then have *False* **using** *n Fφ subformula-all-subformula-st assms*
by (*metis* (*no-types*) *no-equiv-eq*(*1*) *no-equiv-def no-imp-Imp*(*1*) *no-imp-def*)
then have *?case* **by** *blast*
}
moreover {
assume φ': φ' = *FAnd ψ1 ψ2* ∨ φ' = *FOr ψ1 ψ2*
then have *ψ1* = *FT* ∨ *ψ2* = *FT* ∨ *ψ1* = *FF* ∨ *ψ2* = *FF*
using *no-T-F-symb-except-toplevel-bin-decom conn.simps*(*5*,*6*) *n* **unfolding** *binary-connectives-def*
by *fastforce+*
then have *?case* **using** *elimTB.intros* φ' **by** *blast*
}
ultimately show *?case* **using** φ' **by** *blast*
qed

lemma *no-T-F-except-top-level-rew*:
fixes *φ* :: *'v propo*
assumes *noTB*: ¬ *no-T-F-except-top-level φ* **and** *no-equiv*: *no-equiv φ* **and** *no-imp*: *no-imp φ*
shows ∃ *ψ ψ'*. *ψ* ⪯ *φ* ∧ *elimTB ψ ψ'*
proof −
have *test-symb-false-nullary*: ∀ *x*. *no-T-F-symb-except-toplevel* (*FF*:: *'v propo*)
∧ *no-T-F-symb-except-toplevel FT* ∧ *no-T-F-symb-except-toplevel* (*FVar* (*x*:: *'v*)) **by** *auto*
moreover {
fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** *ψ* :: *'v propo*
have *H*: *elimTB* (*conn c l*) *ψ* ⟹ ¬*no-T-F-symb-except-toplevel* (*conn c l*)
by (*cases conn c l rule*: *elimTB.cases*, *auto*)
}
moreover {
fix *x* :: *'v*
have *H'*: *no-T-F-except-top-level FT* *no-T-F-except-top-level FF*
no-T-F-except-top-level (*FVar x*)
by (*auto simp*: *no-T-F-except-top-level-def test-symb-false-nullary*)
}
moreover {
fix *ψ*
have *ψ* ⪯ *φ* ⟹ ¬ *no-T-F-symb-except-toplevel ψ* ⟹ ∃ *ψ'*. *elimTB ψ ψ'*
using *no-T-F-symb-except-toplevel-step-exists no-equiv no-imp* **by** *auto*
}
ultimately show *?thesis*
using *no-test-symb-step-exists noTB* **unfolding** *no-T-F-except-top-level-def* **by** *blast*
qed

lemma *elimTB-inv*:
fixes *φ ψ* :: *'v propo*
assumes *full* (*propo-rew-step elimTB*) *φ ψ*
and *no-equiv φ* **and** *no-imp φ*
shows *no-equiv ψ* **and** *no-imp ψ*
proof −
{
fix *φ ψ* :: *'v propo*
have *H*: *elimTB φ ψ* ⟹ *no-equiv φ* ⟹ *no-equiv ψ*

by (*induct* φ ψ *rule*: *elimTB.induct*, *auto*)
}
then show *no-equiv* ψ
using *full-propo-rew-step-inv-stay-conn*[*of elimTB no-equiv-symb* φ ψ]
no-equiv-symb-conn-characterization assms **unfolding** *no-equiv-def* **by** *metis*
next
{
fix φ ψ :: *'v propo*
have *H*: *elimTB* φ ψ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *no-imp* ψ
by (*induct* φ ψ *rule*: *elimTB.induct*, *auto*)
}
then show *no-imp* ψ
using *full-propo-rew-step-inv-stay-conn*[*of elimTB no-imp-symb* φ ψ] *assms*
no-imp-symb-conn-characterization **unfolding** *no-imp-def* **by** *metis*
qed

lemma *elimTB-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes *no-equiv* φ **and** *no-imp* φ **and** *full* (*propo-rew-step elimTB*) φ ψ
shows *no-T-F-except-top-level* ψ
using *full-propo-rew-step-subformula no-T-F-except-top-level-rew assms elimTB-inv* **by** *fastforce*

3.5.4 PushNeg

Push the negation inside the formula, until the litteral.

inductive *pushNeg* **where**
PushNeg1[*simp*]: *pushNeg* (*FNot* (*FAnd* φ ψ)) (*FOr* (*FNot* φ) (*FNot* ψ)) |
PushNeg2[*simp*]: *pushNeg* (*FNot* (*FOr* φ ψ)) (*FAnd* (*FNot* φ) (*FNot* ψ)) |
PushNeg3[*simp*]: *pushNeg* (*FNot* (*FNot* φ)) φ

lemma *pushNeg-transformation-consistent*:
$A \models FNot\ (FAnd\ \varphi\ \psi) \longleftrightarrow A \models (FOr\ (FNot\ \varphi)\ (FNot\ \psi))$
$A \models FNot\ (FOr\ \varphi\ \psi)\ \longleftrightarrow A \models (FAnd\ (FNot\ \varphi)\ (FNot\ \psi))$
$A \models FNot\ (FNot\ \varphi)\quad \longleftrightarrow A \models \varphi$
by *auto*

lemma *pushNeg-explicit*: *pushNeg* φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct* φ ψ *rule*: *pushNeg.induct*, *auto*)

lemma *pushNeg-consistent*: *preserves-un-sat pushNeg*
unfolding *preserves-un-sat-def* **by** (*simp add*: *pushNeg-explicit*)

lemma *pushNeg-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step pushNeg*))
by (*simp add*: *pushNeg-consistent*)

fun *simple* **where**
simple FT = *True* |
simple FF = *True* |
simple (*FVar* -) = *True* |
simple - = *False*

lemma *simple-decomp*:
simple $\varphi \longleftrightarrow (\varphi = FT \vee \varphi = FF \vee (\exists\, x.\ \varphi = FVar\ x))$
by (*cases* φ) *auto*

lemma *subformula-conn-decomp-simple*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes *s*: *simple* ψ
shows $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \vee \varphi = \psi)$
proof −
have $\varphi \preceq conn\ CNot\ [\psi] \longleftrightarrow (\varphi = conn\ CNot\ [\psi] \vee (\exists\ \psi \in set\ [\psi].\ \varphi \preceq \psi))$
using *subformula-conn-decomp wf-conn-helper-facts*(*1*) **by** *metis*
then show $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \vee \varphi = \psi)$ **using** *s* **by** (*auto simp*: *simple-decomp*)
qed

lemma *subformula-conn-decomp-explicit*[*simp*]:
fixes φ :: *'v propo* **and** *x* :: *'v*
shows
$\varphi \preceq FNot\ FT \longleftrightarrow (\varphi = FNot\ FT \vee \varphi = FT)$
$\varphi \preceq FNot\ FF \longleftrightarrow (\varphi = FNot\ FF \vee \varphi = FF)$
$\varphi \preceq FNot\ (FVar\ x) \longleftrightarrow (\varphi = FNot\ (FVar\ x) \vee \varphi = FVar\ x)$
by (*auto simp*: *subformula-conn-decomp-simple*)

fun *simple-not-symb* **where**
simple-not-symb (*FNot* φ) = (*simple* φ) |
simple-not-symb - = *True*

definition *simple-not* **where**
simple-not = *all-subformula-st simple-not-symb*
declare *simple-not-def*[*simp*]

lemma *simple-not-Not*[*simp*]:
$\neg$ *simple-not* (*FNot* (*FAnd* $\varphi\ \psi$))
$\neg$ *simple-not* (*FNot* (*FOr* $\varphi\ \psi$))
by *auto*

lemma *simple-not-step-exists*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes *no-equiv* φ **and** *no-imp* φ
shows $\psi \preceq \varphi \Longrightarrow \neg$ *simple-not-symb* $\psi \Longrightarrow \exists\, \psi'.$ *pushNeg* $\psi\ \psi'$
apply (*induct* ψ, *auto*)
apply (*rename-tac* ψ, *case-tac* ψ, *auto intro*: *pushNeg.intros*)
by (*metis assms*(*1*,*2*) *no-imp-Imp*(*1*) *no-equiv-eq*(*1*) *no-imp-def no-equiv-def*
subformula-in-subformula-not subformula-all-subformula-st)+

lemma *simple-not-rew*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$ *simple-not* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists\, \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *pushNeg* $\psi\ \psi'$
proof −
have $\forall\, x.$ *simple-not-symb* (*FF*:: *'v propo*) $\wedge$ *simple-not-symb FT* $\wedge$ *simple-not-symb* (*FVar* (*x*:: *'v*))
by *auto*
moreover {
fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** ψ :: *'v propo*
have *H*: *pushNeg* (*conn c l*) $\psi \Longrightarrow \neg$*simple-not-symb* (*conn c l*)
by (*cases conn c l rule*: *pushNeg.cases*) *auto*

}
moreover {
fix x :: $'v$
have H': *simple-not FT simple-not FF simple-not* (*FVar* x)
by *simp-all*
}
moreover {
fix ψ :: $'v$ *propo*
have $\psi \preceq \varphi \Longrightarrow \neg$ *simple-not-symb* $\psi \Longrightarrow \exists \psi'$. *pushNeg* ψ ψ'
using *simple-not-step-exists no-equiv no-imp* **by** *blast*
}
ultimately show *?thesis* **using** *no-test-symb-step-exists noTB* **unfolding** *simple-not-def* **by** *blast*
qed

lemma *no-T-F-except-top-level-pushNeg1*:
no-T-F-except-top-level (*FNot* (*FAnd* φ ψ)) $\Longrightarrow$ *no-T-F-except-top-level* (*FOr* (*FNot* φ) (*FNot* ψ))
using *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-comp-not no-T-F-decomp*(*1*)
no-T-F-decomp(*2*) *no-T-F-no-T-F-except-top-level* **by** (*metis no-T-F-comp-expanded-explicit*(*2*)
propo.distinct(*5*,*17*))

lemma *no-T-F-except-top-level-pushNeg2*:
no-T-F-except-top-level (*FNot* (*FOr* φ ψ)) $\Longrightarrow$ *no-T-F-except-top-level* (*FAnd* (*FNot* φ) (*FNot* ψ))
by *auto*

lemma *no-T-F-symb-pushNeg*:
no-T-F-symb (*FOr* (*FNot* φ') (*FNot* ψ'))
no-T-F-symb (*FAnd* (*FNot* φ') (*FNot* ψ'))
no-T-F-symb (*FNot* (*FNot* φ'))
by *auto*

lemma *propo-rew-step-pushNeg-no-T-F-symb*:
propo-rew-step pushNeg φ $\psi \Longrightarrow$ *no-T-F-except-top-level* $\varphi \Longrightarrow$ *no-T-F-symb* $\varphi \Longrightarrow$ *no-T-F-symb* ψ
apply (*induct rule*: *propo-rew-step.induct*)
apply (*cases rule*: *pushNeg.cases*)
apply *simp-all*
apply (*metis no-T-F-symb-pushNeg*(*1*))
apply (*metis no-T-F-symb-pushNeg*(*2*))
apply (*simp*, *metis all-subformula-st-test-symb-true-phi no-T-F-def*)
proof −
fix φ φ':: $'a$ *propo* **and** c:: $'a$ *connective* **and** ξ ξ':: $'a$ *propo list*
assume *rel*: *propo-rew-step pushNeg* φ φ'
and *IH*: *no-T-F* $\varphi \Longrightarrow$ *no-T-F-symb* $\varphi \Longrightarrow$ *no-T-F-symb* φ'
and *wf*: *wf-conn* c (ξ @ φ # ξ')
and n: *conn* c (ξ @ φ # ξ') = *FF* $\vee$ *conn* c (ξ @ φ # ξ') = *FT* $\vee$ *no-T-F* (*conn* c (ξ @ φ # ξ'))
and x: $c \neq$ *CF* $\wedge$ $c \neq$ *CT* $\wedge$ $\varphi \neq$ *FF* $\wedge$ $\varphi \neq$ *FT* $\wedge$ ($\forall \psi \in$ *set* $\xi \cup$ *set* ξ'. $\psi \neq$ *FF* $\wedge$ $\psi \neq$ *FT*)
then have $c \neq$ *CF* $\wedge$ $c \neq$ *CF* $\wedge$ *wf-conn* c (ξ @ φ' # ξ')
using *wf-conn-no-arity-change-helper wf-conn-no-arity-change* **by** *metis*
moreover have n': *no-T-F* (*conn* c (ξ @ φ # ξ')) **using** n **by** (*simp add*: *wf wf-conn-list*(*1*,*2*))
moreover
{
have *no-T-F* φ
by (*metis Un-iff all-subformula-st-decomp list.set-intros*(*1*) n' *wf no-T-F-def set-append*)
moreover then have *no-T-F-symb* φ
by (*simp add*: *all-subformula-st-test-symb-true-phi no-T-F-def*)
ultimately have $\varphi' \neq$ *FF* $\wedge$ $\varphi' \neq$ *FT*
using *IH no-T-F-symb-false*(*1*) *no-T-F-symb-false*(*2*) **by** *blast*

then have $\forall \psi \in$ *set* $(\xi \ @ \ \varphi' \ \# \ \xi')$. $\psi \neq FF \wedge \psi \neq FT$ **using** *x* **by** *auto*
}
ultimately show *no-T-F-symb* (*conn c* $(\xi \ @ \ \varphi' \ \# \ \xi')$) **by** (*simp add*: *x*)
qed

lemma *propo-rew-step-pushNeg-no-T-F*:
propo-rew-step pushNeg φ ψ $\Longrightarrow$ *no-T-F* φ $\Longrightarrow$ *no-T-F* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case *global-rel*
then show *?case*
by (*metis* (*no-types*, *lifting*) *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-def no-T-F-except-top-level-pushNeg1 no-T-F-except-top-level-pushNeg2 no-T-F-no-T-F-except-top-level all-subformula-st-decomp-explicit*(*3*) *pushNeg.simps simple.simps*(*1*,*2*,*5*,*6*))
next
case (*propo-rew-one-step-lift* φ φ' *c* ξ ξ')
note *rel* = *this*(*1*) **and** *IH* = *this*(*2*) **and** *wf* = *this*(*3*) **and** *no-T-F* = *this*(*4*)
moreover have *wf'*: *wf-conn c* $(\xi \ @ \ \varphi' \ \# \ \xi')$
using *wf-conn-no-arity-change wf-conn-no-arity-change-helper wf* **by** *metis*
ultimately show *no-T-F* (*conn c* $(\xi \ @ \ \varphi' \ \# \ \xi')$)
using *all-subformula-st-test-symb-true-phi*
by (*fastforce simp*: *no-T-F-def all-subformula-st-decomp wf wf'*)
qed

lemma *pushNeg-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step pushNeg*) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ
proof –
{
fix φ ψ :: *'v propo*
assume *rel*: *propo-rew-step pushNeg* φ ψ
and *no*: *no-T-F-except-top-level* φ
then have *no-T-F-except-top-level* ψ
proof –
{
assume $\varphi = FT \vee \varphi = FF$
from *rel this* **have** *False*
apply (*induct rule*: *propo-rew-step.induct*)
using *pushNeg.cases* **apply** *blast*
using *wf-conn-list*(*1*) *wf-conn-list*(*2*) **by** *auto*
then have *no-T-F-except-top-level* ψ **by** *blast*
}
moreover {
assume $\varphi \neq FT \wedge \varphi \neq FF$
then have *no-T-F* φ
by (*metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*)
then have *no-T-F* ψ
using *propo-rew-step-pushNeg-no-T-F rel* **by** *auto*
then have *no-T-F-except-top-level* ψ **by** (*simp add*: *no-T-F-no-T-F-except-top-level*)
}
ultimately show *no-T-F-except-top-level* ψ **by** *metis*
qed
}

```
  moreover {
    fix c :: 'v connective and ξ ξ' :: 'v propo list and ζ ζ' :: 'v propo
    assume rel: propo-rew-step pushNeg ζ ζ'
    and incl: ζ ≼ φ
    and corr: wf-conn c (ξ @ ζ # ξ')
    and no-T-F: no-T-F-symb-except-toplevel (conn c (ξ @ ζ # ξ'))
    and n: no-T-F-symb-except-toplevel ζ'
    have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
    proof
      have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
        using corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F
        by blast
      have l: ∀φ∈set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
        using corr wf-conn-no-T-F-symb-iff p by blast
      from rel incl have ζ'≠FT ∧ζ'≠FF
        apply (induction ζ ζ' rule: propo-rew-step.induct)
        apply (cases rule: pushNeg.cases, auto)
        by (metis assms(4) no-T-F-symb-except-top-level-false-not no-T-F-except-top-level-def
          all-subformula-st-test-symb-true-phi subformula-in-subformula-not
          subformula-all-subformula-st append-is-Nil-conv list.distinct(1)
          wf-conn-no-arity-change-helper wf-conn-list(1,2) wf-conn-no-arity-change)+
      then have ∀φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
      moreover have c ≠ CT ∧ c ≠ CF using corr by auto
      ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
        by (metis corr no-T-F-symb-comp wf-conn-no-arity-change wf-conn-no-arity-change-helper)
    qed
  }
  ultimately show no-T-F-except-top-level ψ
    using full-propo-rew-step-inv-stay-with-inc[of pushNeg no-T-F-symb-except-toplevel φ] assms
      subformula-refl unfolding no-T-F-except-top-level-def full-unfold by metis
next
  {
    fix φ ψ :: 'v propo
    have H: pushNeg φ ψ ⟹ no-equiv φ ⟹ no-equiv ψ
      by (induct φ ψ rule: pushNeg.induct, auto)
  }
  then show no-equiv ψ
    using full-propo-rew-step-inv-stay-conn[of pushNeg no-equiv-symb φ ψ]
    no-equiv-symb-conn-characterization assms unfolding no-equiv-def full-unfold by metis
next
  {
    fix φ ψ :: 'v propo
    have H: pushNeg φ ψ ⟹ no-imp φ ⟹  no-imp ψ
      by (induct φ ψ rule: pushNeg.induct, auto)
  }
  then show no-imp ψ
    using full-propo-rew-step-inv-stay-conn[of pushNeg no-imp-symb φ ψ] assms
      no-imp-symb-conn-characterization unfolding no-imp-def full-unfold by metis
qed


lemma pushNeg-full-propo-rew-step:
  fixes φ ψ :: 'v propo
  assumes
    no-equiv φ and
    no-imp φ and
```

full (*propo-rew-step pushNeg*) φ ψ **and**
no-T-F-except-top-level φ
shows *simple-not* ψ
using *assms full-propo-rew-step-subformula pushNeg-inv*(*1*,*2*) *simple-not-rew* **by** *blast*

3.5.5 Push inside

inductive *push-conn-inside* :: *'v connective* ⇒ *'v connective* ⇒ *'v propo* ⇒ *'v propo* ⇒ *bool*
for *c c'*:: *'v connective* **where**
push-conn-inside-l[*simp*]: $c = CAnd \lor c = COr \Longrightarrow c' = CAnd \lor c' = COr$
$\Longrightarrow$ *push-conn-inside c c'* (*conn c* [*conn c'* [φ*1*, φ*2*], ψ])
(*conn c'* [*conn c* [φ*1*, ψ], *conn c* [φ*2*, ψ]]) |
push-conn-inside-r[*simp*]: $c = CAnd \lor c = COr \Longrightarrow c' = CAnd \lor c' = COr$
$\Longrightarrow$ *push-conn-inside c c'* (*conn c* [ψ, *conn c'* [φ*1*, φ*2*]])
(*conn c'* [*conn c* [ψ, φ*1*], *conn c* [ψ, φ*2*]])

lemma *push-conn-inside-explicit*: *push-conn-inside c c'* φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct* φ ψ *rule*: *push-conn-inside.induct*, *auto*)

lemma *push-conn-inside-consistent*: *preserves-un-sat* (*push-conn-inside c c'*)
unfolding *preserves-un-sat-def* **by** (*simp add*: *push-conn-inside-explicit*)

lemma *propo-rew-step-push-conn-inside*[*simp*]:
¬*propo-rew-step* (*push-conn-inside c c'*) *FT* ψ ¬*propo-rew-step* (*push-conn-inside c c'*) *FF* ψ
proof −
{
{
fix φ ψ
have *push-conn-inside c c'* φ ψ $\Longrightarrow \varphi = FT \lor \varphi = FF \Longrightarrow$ *False*
by (*induct rule*: *push-conn-inside.induct*, *auto*)
} **note** H = *this*
fix φ
have *propo-rew-step* (*push-conn-inside c c'*) φ ψ $\Longrightarrow \varphi = FT \lor \varphi = FF \Longrightarrow$ *False*
apply (*induct rule*: *propo-rew-step.induct*, *auto simp*: *wf-conn-list*(*1*) *wf-conn-list*(*2*))
using H **by** *blast*+
}
then show
¬*propo-rew-step* (*push-conn-inside c c'*) *FT* ψ
¬*propo-rew-step* (*push-conn-inside c c'*) *FF* ψ **by** *blast*+
qed

inductive *not-c-in-c'-symb*:: *'v connective* ⇒ *'v connective* ⇒ *'v propo* ⇒ *bool* **for** *c c'* **where**
not-c-in-c'-symb-l[*simp*]: *wf-conn c* [*conn c'* [φ, φ'], ψ] $\Longrightarrow$ *wf-conn c'* [φ, φ']
$\Longrightarrow$ *not-c-in-c'-symb c c'* (*conn c* [*conn c'* [φ, φ'], ψ]) |
not-c-in-c'-symb-r[*simp*]: *wf-conn c* [ψ, *conn c'* [φ, φ']] $\Longrightarrow$ *wf-conn c'* [φ, φ']
$\Longrightarrow$ *not-c-in-c'-symb c c'* (*conn c* [ψ, *conn c'* [φ, φ']])

abbreviation *c-in-c'-symb c c'* φ ≡ ¬*not-c-in-c'-symb c c'* φ

lemma *c-in-c'-symb-simp*:
not-c-in-c'-symb c c' $\xi \Longrightarrow \xi = FF \lor \xi = FT \lor \xi = FVar\ x \lor \xi = FNot\ FF \lor \xi = FNot\ FT$
$\lor\ \xi = FNot\ (FVar\ x) \Longrightarrow$ *False*
apply (*induct rule*: *not-c-in-c'-symb.induct*, *auto simp*: *wf-conn.simps wf-conn-list*(*1*−*3*))

using *conn-inj-not*(*2*) *wf-conn-binary* **unfolding** *binary-connectives-def* **by** *fastforce*+

lemma *c-in-c′-symb-simp′*[*simp*]:
$\neg$*not-c-in-c′-symb c c′ FF*
$\neg$*not-c-in-c′-symb c c′ FT*
$\neg$*not-c-in-c′-symb c c′* (*FVar x*)
$\neg$*not-c-in-c′-symb c c′* (*FNot FF*)
$\neg$*not-c-in-c′-symb c c′* (*FNot FT*)
$\neg$*not-c-in-c′-symb c c′* (*FNot* (*FVar x*))
using *c-in-c′-symb-simp* **by** *metis*+

definition *c-in-c′-only* **where**
c-in-c′-only c c′ $\equiv$ *all-subformula-st* (*c-in-c′-symb c c′*)

lemma *c-in-c′-only-simp*[*simp*]:
c-in-c′-only c c′ FF
c-in-c′-only c c′ FT
c-in-c′-only c c′ (*FVar x*)
c-in-c′-only c c′ (*FNot FF*)
c-in-c′-only c c′ (*FNot FT*)
c-in-c′-only c c′ (*FNot* (*FVar x*))
unfolding *c-in-c′-only-def* **by** *auto*

lemma *not-c-in-c′-symb-commute*:
not-c-in-c′-symb c c′ $\xi \Longrightarrow$ *wf-conn c* $[\varphi, \psi] \Longrightarrow \xi =$ *conn c* $[\varphi, \psi]$
$\Longrightarrow$ *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
proof (*induct rule*: *not-c-in-c′-symb.induct*)
case (*not-c-in-c′-symb-r* φ' φ'' ψ') **note** H = *this*
then have ψ: ψ = *conn c′* $[\varphi'', \psi']$ **using** *conn-inj* **by** *auto*
have *wf-conn c* [*conn c′* $[\varphi'', \psi']$, φ]
using *H*(*1*) *wf-conn-no-arity-change length-Cons* **by** *metis*
then show *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
unfolding ψ **using** *not-c-in-c′-symb.intros*(*1*) *H* **by** *auto*
next
case (*not-c-in-c′-symb-l* φ' φ'' ψ') **note** H = *this*
then have φ = *conn c′* $[\varphi', \varphi'']$ **using** *conn-inj* **by** *auto*
moreover have *wf-conn c* [ψ', *conn c′* $[\varphi', \varphi'']$]
using *H*(*1*) *wf-conn-no-arity-change length-Cons* **by** *metis*
ultimately show *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
using *not-c-in-c′-symb.intros*(*2*) *conn-inj not-c-in-c′-symb-l.hyps*
not-c-in-c′-symb-l.prems(*1*,*2*) **by** *blast*
qed

lemma *not-c-in-c′-symb-commute′*:
wf-conn c $[\varphi, \psi] \Longrightarrow$ *c-in-c′-symb c c′* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ *c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
using *not-c-in-c′-symb-commute wf-conn-no-arity-change* **by** (*metis length-Cons*)

lemma *not-c-in-c′-comm*:
assumes *wf*: *wf-conn c* $[\varphi, \psi]$
shows *c-in-c′-only c c′* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ *c-in-c′-only c c′* (*conn c* $[\psi, \varphi]$) (**is** *?A* $\longleftrightarrow$ *?B*)
proof −
have *?A* $\longleftrightarrow$ (*c-in-c′-symb c c′* (*conn c* $[\varphi, \psi]$)
$\wedge$ ($\forall \xi \in$ *set* $[\varphi, \psi]$. *all-subformula-st* (*c-in-c′-symb c c′*) ξ))
using *all-subformula-st-decomp wf* **unfolding** *c-in-c′-only-def* **by** *fastforce*
also have ... $\longleftrightarrow$ (*c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)

$\wedge$ ($\forall\xi \in$ *set* [ψ, φ]. *all-subformula-st* (*c-in-c'-symb c c'*) ξ))
using *not-c-in-c'-symb-commute' wf* **by** *auto*
also
have *wf-conn c* [ψ, φ] **using** *wf-conn-no-arity-change wf* **by** (*metis length-Cons*)
then have (*c-in-c'-symb c c'* (*conn c* [ψ, φ])
$\wedge$ ($\forall\xi \in$ *set* [ψ, φ]. *all-subformula-st* (*c-in-c'-symb c c'*) ξ))
$\longleftrightarrow$ *?B*
using *all-subformula-st-decomp* **unfolding** *c-in-c'-only-def* **by** *fastforce*
finally show *?thesis* **.**
qed

lemma *not-c-in-c'-simp*[*simp*]:
fixes $\varphi 1$ $\varphi 2$ ψ :: *'v propo* **and** *x* :: *'v*
shows
c-in-c'-symb c c' FT
c-in-c'-symb c c' FF
c-in-c'-symb c c' (*FVar x*)
wf-conn c [*conn c'* [$\varphi 1$, $\varphi 2$], ψ] $\Longrightarrow$ *wf-conn c'* [$\varphi 1$, $\varphi 2$]
$\Longrightarrow$ $\neg$ *c-in-c'-only c c'* (*conn c* [*conn c'* [$\varphi 1$, $\varphi 2$], ψ])
apply (*simp-all add*: *c-in-c'-only-def*)
using *all-subformula-st-test-symb-true-phi not-c-in-c'-symb-l* **by** *blast*

lemma *c-in-c'-symb-not*[*simp*]:
fixes *c c'* :: *'v connective* **and** ψ :: *'v propo*
shows *c-in-c'-symb c c'* (*FNot* ψ)
proof −
{
fix ξ :: *'v propo*
have *not-c-in-c'-symb c c'* (*FNot* ψ) $\Longrightarrow$ *False*
apply (*induct FNot* ψ *rule*: *not-c-in-c'-symb.induct*)
using *conn-inj-not*(*2*) **by** *blast+*
}
then show *?thesis* **by** *auto*
qed

lemma *c-in-c'-symb-step-exists*:
fixes φ :: *'v propo*
assumes *c*: *c* = *CAnd* $\vee$ *c* = *COr* **and** *c'*: *c'* = *CAnd* $\vee$ *c'* = *COr*
shows $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi \Longrightarrow \exists\psi'$. *push-conn-inside c c'* ψ ψ'
apply (*induct* ψ *rule*: *propo-induct-arity*)
apply *auto*[*2*]
proof −
fix $\psi 1$ $\psi 2$ φ':: *'v propo*
assume *IH*$\psi 1$: $\psi 1 \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi 1$ $\Longrightarrow$ *Ex* (*push-conn-inside c c'* $\psi 1$)
and *IH*$\psi 2$: $\psi 1 \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi 1$ $\Longrightarrow$ *Ex* (*push-conn-inside c c'* $\psi 1$)
and φ': φ' = *FAnd* $\psi 1$ $\psi 2$ $\vee$ φ' = *FOr* $\psi 1$ $\psi 2$ $\vee$ φ' = *FImp* $\psi 1$ $\psi 2$ $\vee$ φ' = *FEq* $\psi 1$ $\psi 2$
and *in*φ: $\varphi' \preceq \varphi$ **and** *n0*: $\neg$*c-in-c'-symb c c'* φ'
then have *n*: *not-c-in-c'-symb c c'* φ' **by** *auto*
{
assume φ': φ' = *conn c* [$\psi 1$, $\psi 2$]
obtain *a b* **where** $\psi 1$ = *conn c'* [*a*, *b*] $\vee$ $\psi 2$ = *conn c'* [*a*, *b*]
using *n* φ' **apply** (*induct rule*: *not-c-in-c'-symb.induct*)
using *c* **by** *force+*
then have *Ex* (*push-conn-inside c c'* φ')
unfolding φ' **apply** *auto*
using *push-conn-inside.intros*(*1*) *c c'* **apply** *blast*

using *push-conn-inside.intros*(*2*) *c c'* **by** *blast*
}
moreover {
assume φ': $\varphi' \neq$ *conn c* [$\psi 1$, $\psi 2$]
have $\forall \varphi$ *c ca*. $\exists \varphi 1$ $\psi 1$ $\psi 2$ $\psi 1'$ $\psi 2'$ $\varphi 2'$. *conn* (*c*::*'v connective*) [$\varphi 1$, *conn ca* [$\psi 1$, $\psi 2$]] = φ $\vee$ *conn c* [*conn ca* [$\psi 1'$, $\psi 2'$], $\varphi 2'$] = φ $\vee$ *c-in-c'-symb c ca* φ
by (*metis not-c-in-c'-symb.cases*)
then have *Ex* (*push-conn-inside c c'* φ')
by (*metis* (*no-types*) *c c' n push-conn-inside-l push-conn-inside-r*)
}
ultimately show *Ex* (*push-conn-inside c c'* φ') **by** *blast*
qed

lemma *c-in-c'-symb-rew*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$*c-in-c'-only c c'* φ
and *c*: *c* = *CAnd* $\vee$ *c* = *COr* **and** *c'*: *c'* = *CAnd* $\vee$ *c'* = *COr*
shows $\exists \psi$ ψ'. $\psi \preceq \varphi$ $\wedge$ *push-conn-inside c c'* ψ ψ'
proof −
have *test-symb-false-nullary*:
$\forall x$. *c-in-c'-symb c c'* (*FF*:: *'v propo*) $\wedge$ *c-in-c'-symb c c' FT* $\wedge$ *c-in-c'-symb c c'* (*FVar* (*x*:: *'v*))
by *auto*
moreover {
fix *x* :: *'v*
have *H'*: *c-in-c'-symb c c' FT c-in-c'-symb c c' FF c-in-c'-symb c c'* (*FVar x*)
by *simp+*
}
moreover {
fix ψ :: *'v propo*
have $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi \Longrightarrow \exists \psi'$. *push-conn-inside c c'* ψ ψ'
by (*auto simp*: *assms*(*2*) *c' c-in-c'-symb-step-exists*)
}
ultimately show *?thesis* **using** *noTB no-test-symb-step-exists*[*of c-in-c'-symb c c'*]
unfolding *c-in-c'-only-def* **by** *metis*
qed

lemma *push-conn-insidec-in-c'-symb-no-T-F*:
fixes φ ψ :: *'v propo*
shows *propo-rew-step* (*push-conn-inside c c'*) φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case (*global-rel* φ ψ)
then show *no-T-F* ψ
by (*cases rule*: *push-conn-inside.cases*, *auto*)
next
case (*propo-rew-one-step-lift* φ φ' *c* ξ ξ')
note *rel* = *this*(*1*) **and** *IH* = *this*(*2*) **and** *wf* = *this*(*3*) **and** *no-T-F* = *this*(*4*)
have *no-T-F* φ
using *wf no-T-F no-T-F-def subformula-into-subformula subformula-all-subformula-st subformula-refl* **by** (*metis* (*no-types*) *in-set-conv-decomp*)
then have φ': *no-T-F* φ' **using** *IH* **by** *blast*

have $\forall \zeta \in$ *set* (ξ @ φ # ξ'). *no-T-F* ζ **by** (*metis wf no-T-F no-T-F-def all-subformula-st-decomp*)
then have *n*: $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). *no-T-F* ζ **using** φ' **by** *auto*
then have *n'*: $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). $\zeta \neq$ *FF* $\wedge$ $\zeta \neq$ *FT*

using φ' **by** (*metis no-T-F-symb-false*(*1*) *no-T-F-symb-false*(*2*) *no-T-F-def all-subformula-st-test-symb-true-phi*)

have *wf'*: *wf-conn c* (ξ @ φ' # ξ')
using *wf wf-conn-no-arity-change* **by** (*metis wf-conn-no-arity-change-helper*)
{
fix x :: *'v*
assume $c = CT \vee c = CF \vee c = CVar\ x$
then have *False* **using** *wf* **by** *auto*
then have *no-T-F* (*conn c* (ξ @ φ' # ξ')) **by** *blast*
}
moreover {
assume *c*: $c = CNot$
then have $\xi = []$ $\xi' = []$ **using** *wf* **by** *auto*
then have *no-T-F* (*conn c* (ξ @ φ' # ξ'))
using *c* **by** (*metis* φ' *conn.simps*(*4*) *no-T-F-symb-false*(*1*,*2*) *no-T-F-symb-fnot no-T-F-def all-subformula-st-decomp-explicit*(*3*) *all-subformula-st-test-symb-true-phi self-append-conv2*)
}
moreover {
assume *c*: $c \in$ *binary-connectives*
then have *no-T-F-symb* (*conn c* (ξ @ φ' # ξ')) **using** *wf' n' no-T-F-symb.simps* **by** *fastforce*
then have *no-T-F* (*conn c* (ξ @ φ' # ξ'))
by (*metis all-subformula-st-decomp-imp wf' n no-T-F-def*)
}
ultimately show *no-T-F* (*conn c* (ξ @ φ' # ξ')) **using** *connective-cases-arity* **by** *auto*
qed

lemma *simple-propo-rew-step-push-conn-inside-inv*:
propo-rew-step (*push-conn-inside c c'*) φ ψ $\Longrightarrow$ *simple* φ $\Longrightarrow$ *simple* ψ
apply (*induct rule*: *propo-rew-step.induct*)
apply (*rename-tac* φ, *case-tac* φ, *auto simp*: *push-conn-inside.simps*)[]
by (*metis append-is-Nil-conv list.distinct*(*1*) *simple.elims*(*2*) *wf-conn-list*(*1*−*3*))

lemma *simple-propo-rew-step-inv-push-conn-inside-simple-not*:
fixes *c c'* :: *'v connective* **and** φ ψ :: *'v propo*
shows *propo-rew-step* (*push-conn-inside c c'*) φ ψ $\Longrightarrow$ *simple-not* φ $\Longrightarrow$ *simple-not* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case (*global-rel* φ ψ)
then show *?case* **by** (*cases* φ, *auto simp*: *push-conn-inside.simps*)
next
case (*propo-rew-one-step-lift* φ φ' *ca* ξ ξ') **note** *rew* = *this*(*1*) **and** *IH* = *this*(*2*) **and** *wf* = *this*(*3*) **and** *simple* = *this*(*4*)
show *?case*
proof (*cases ca rule*: *connective-cases-arity*)
case *nullary*
then show *?thesis* **using** *propo-rew-one-step-lift* **by** *auto*
next
case *binary* **note** *ca* = *this*
obtain *a b* **where** *ab*: ξ @ φ' # ξ' = [*a*, *b*]
using *wf ca list-length2-decomp wf-conn-bin-list-length*
by (*metis* (*no-types*) *wf-conn-no-arity-change-helper*)
have $\forall \zeta \in$ *set* (ξ @ φ # ξ'). *simple-not* ζ
by (*metis wf all-subformula-st-decomp simple simple-not-def*)
then have $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). *simple-not* ζ **using** *IH* **by** *simp*

moreover have *simple-not-symb* (*conn ca* (ξ @ φ' # ξ')) **using** *ca*
by (*metis ab conn.simps*(*5*−*8*) *helper-fact simple-not-symb.simps*(*5*) *simple-not-symb.simps*(*6*) *simple-not-symb.simps*(*7*) *simple-not-symb.simps*(*8*))
ultimately show *?thesis*
by (*simp add*: *ab all-subformula-st-decomp ca*)
next
case *unary*
then show *?thesis*
using *rew simple-propo-rew-step-push-conn-inside-inv*[*OF rew*] *IH local.wf simple* **by** *auto*
qed
qed

lemma *propo-rew-step-push-conn-inside-simple-not*:
fixes φ φ' :: *'v propo* **and** ξ ξ' :: *'v propo list* **and** *c* :: *'v connective*
assumes
propo-rew-step (*push-conn-inside c c'*) φ φ' **and**
wf-conn c (ξ @ φ # ξ') **and**
simple-not-symb (*conn c* (ξ @ φ # ξ')) **and**
simple-not-symb φ'
shows *simple-not-symb* (*conn c* (ξ @ φ' # ξ'))
using *assms*
proof (*induction rule*: *propo-rew-step.induct*)
print-cases
case (*global-rel*)
then show *?case*
by (*metis conn.simps*(*12*,*17*) *list.discI push-conn-inside.cases simple-not-symb.elims*(*3*) *wf-conn-helper-facts*(*5*) *wf-conn-list*(*2*) *wf-conn-list*(*8*) *wf-conn-no-arity-change wf-conn-no-arity-change-helper*)
next
case (*propo-rew-one-step-lift* φ φ' *c'* χs $\chi s'$) **note** *tel* = *this*(*1*) **and** *wf* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *wf'* = *this*(*4*) **and** *simple'* = *this*(*5*) **and** *simple* = *this*(*6*)
then show *?case*
proof (*cases c' rule*: *connective-cases-arity*)
case *nullary*
then show *?thesis* **using** *wf simple simple'* **by** *auto*
next
case *binary* **note** *c* = *this*(*1*)
have *corr'*: *wf-conn c* (ξ @ *conn c'* (χs @ φ' # $\chi s'$) # ξ')
using *wf wf-conn-no-arity-change*
by (*metis wf' wf-conn-no-arity-change-helper*)
then show *?thesis*
using *c propo-rew-one-step-lift wf*
by (*metis conn.simps*(*17*) *connective.distinct*(*37*) *propo-rew-step-subformula-imp push-conn-inside.cases simple-not-symb.elims*(*3*) *wf-conn.simps wf-conn-list*(*2*,*8*))
next
case *unary*
then have *empty*: χs = [] $\chi s'$ = [] **using** *wf* **by** *auto*
then show *?thesis* **using** *simple unary simple' wf wf'*
by (*metis connective.distinct*(*37*) *connective.distinct*(*39*) *propo-rew-step-subformula-imp push-conn-inside.cases simple-not-symb.elims*(*3*) *tel wf-conn-list*(*8*) *wf-conn-no-arity-change wf-conn-no-arity-change-helper*)
qed
qed

lemma *push-conn-inside-not-true-false*:
push-conn-inside c c' φ ψ $\Longrightarrow$ $\psi \neq FT \land \psi \neq FF$

```
  by (induct rule: push-conn-inside.induct, auto)

lemma push-conn-inside-inv:
  fixes φ ψ :: 'v propo
  assumes full (propo-rew-step (push-conn-inside c c')) φ ψ
  and no-equiv φ and no-imp φ and no-T-F-except-top-level φ and simple-not φ
  shows no-equiv ψ and no-imp ψ and no-T-F-except-top-level ψ and simple-not ψ
proof -
  {
    {
      fix φ ψ :: 'v propo
      have H: push-conn-inside c c' φ ψ ⟹ all-subformula-st simple-not-symb φ
        ⟹ all-subformula-st simple-not-symb ψ
        by (induct φ ψ rule: push-conn-inside.induct, auto)
    } note H = this

    fix φ ψ :: 'v propo
    have H: propo-rew-step (push-conn-inside c c') φ ψ ⟹ all-subformula-st simple-not-symb φ
      ⟹ all-subformula-st simple-not-symb ψ
      apply (induct φ ψ rule: propo-rew-step.induct)
      using H apply simp
      proof (rename-tac φ φ' ca ψs ψs', case-tac ca rule: connective-cases-arity)
        fix φ φ' :: 'v propo and c:: 'v connective and ξ ξ':: 'v propo list
        and x:: 'v
        assume wf-conn c (ξ @ φ # ξ')
        and  c = CT ∨ c = CF ∨ c = CVar x
        then have ξ @ φ # ξ' = [] by auto
        then have False by auto
        then show all-subformula-st simple-not-symb (conn c (ξ @ φ' # ξ')) by blast
      next
        fix φ φ' :: 'v propo and ca:: 'v connective and ξ ξ':: 'v propo list
        and x :: 'v
        assume rel: propo-rew-step (push-conn-inside c c') φ φ'
        and φ-φ': all-subformula-st simple-not-symb φ ⟹ all-subformula-st simple-not-symb φ'
        and corr: wf-conn ca (ξ @ φ # ξ')
        and n: all-subformula-st simple-not-symb (conn ca (ξ @ φ # ξ'))
        and c: ca = CNot

        have empty: ξ = [] ξ' = [] using c corr by auto
        then have simple-not:all-subformula-st simple-not-symb (FNot φ) using corr c n by auto
        then have simple φ
          using all-subformula-st-test-symb-true-phi simple-not-symb.simps(1) by blast
        then have simple φ'
          using rel simple-propo-rew-step-push-conn-inside-inv by blast
        then show all-subformula-st simple-not-symb (conn ca (ξ @ φ' # ξ')) using c empty
          by (metis simple-not φ-φ' append-Nil conn.simps(4) all-subformula-st-decomp-explicit(3)
            simple-not-symb.simps(1))
      next
        fix φ φ' :: 'v propo and ca :: 'v connective and ξ ξ' :: 'v propo list
        and x :: 'v
        assume rel: propo-rew-step (push-conn-inside c c') φ φ'
        and nφ: all-subformula-st simple-not-symb φ ⟹  all-subformula-st simple-not-symb φ'
        and corr: wf-conn ca (ξ @ φ # ξ')
        and n: all-subformula-st simple-not-symb (conn ca (ξ @ φ # ξ'))
        and c: ca ∈ binary-connectives
```

have *all-subformula-st simple-not-symb* φ
  using *n c corr all-subformula-st-decomp* **by** *fastforce*
then have φ': *all-subformula-st simple-not-symb* φ' **using** *n*φ **by** *blast*
obtain *a b* **where** *ab*: $[a, b] = (\xi\ @\ \varphi\ \#\ \xi')$
  using *corr c list-length2-decomp wf-conn-bin-list-length* **by** *metis*
then have $\xi\ @\ \varphi'\ \#\ \xi' = [a, \varphi'] \lor (\xi\ @\ \varphi'\ \#\ \xi') = [\varphi', b]$
  using *ab* **by** (*metis* (*no-types*, *hide-lams*) *append-Cons append-Nil append-Nil2*
    append-is-Nil-conv butlast.simps(*2*) *butlast-append list.sel*(*3*) *tl-append2*)
moreover
{
  fix χ :: *'v propo*
  have *wf'*: *wf-conn ca* $[a, b]$
    using *ab corr* **by** *presburger*
  have *all-subformula-st simple-not-symb* (*conn ca* $[a, b]$)
    using *ab n* **by** *presburger*
  then have *all-subformula-st simple-not-symb* $\chi \lor \chi \notin$ *set* $(\xi\ @\ \varphi'\ \#\ \xi')$
    using *wf'* **by** (*metis* (*no-types*) φ' *all-subformula-st-decomp calculation insert-iff*
      list.set(*2*))
}
then have $\forall \varphi.\ \varphi \in$ *set* $(\xi\ @\ \varphi'\ \#\ \xi') \longrightarrow$ *all-subformula-st simple-not-symb* φ
    by (*metis* (*no-types*))

moreover have *simple-not-symb* (*conn ca* $(\xi\ @\ \varphi'\ \#\ \xi')$)
  using *ab conn-inj-not*(*1*) *corr wf-conn-list-decomp*(*4*) *wf-conn-no-arity-change*
    not-Cons-self2 self-append-conv2 simple-not-symb.elims(*3*) **by** (*metis* (*no-types*) *c*
    calculation(*1*) *wf-conn-binary*)
moreover have *wf-conn ca* $(\xi\ @\ \varphi'\ \#\ \xi')$ **using** *c calculation*(*1*) **by** *auto*
ultimately show *all-subformula-st simple-not-symb* (*conn ca* $(\xi\ @\ \varphi'\ \#\ \xi')$)
  by (*metis all-subformula-st-decomp-imp*)
qed
}
moreover {
  fix *ca* :: *'v connective* **and** $\xi\ \xi'$ :: *'v propo list* **and** $\varphi\ \varphi'$ :: *'v propo*
  have *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \varphi' \Longrightarrow$ *wf-conn ca* $(\xi\ @\ \varphi\ \#\ \xi')$
    $\Longrightarrow$ *simple-not-symb* (*conn ca* $(\xi\ @\ \varphi\ \#\ \xi')$) $\Longrightarrow$ *simple-not-symb* φ'
    $\Longrightarrow$ *simple-not-symb* (*conn ca* $(\xi\ @\ \varphi'\ \#\ \xi')$)
    by (*metis append-self-conv2 conn.simps*(*4*) *conn-inj-not*(*1*) *simple-not-symb.elims*(*3*)
      simple-not-symb.simps(*1*) *simple-propo-rew-step-push-conn-inside-inv*
      wf-conn-no-arity-change-helper wf-conn-list-decomp(*4*) *wf-conn-no-arity-change*)
}
ultimately show *simple-not* ψ
  using *full-propo-rew-step-inv-stay'*[*of push-conn-inside c c' simple-not-symb*] *assms*
  unfolding *no-T-F-except-top-level-def simple-not-def full-unfold* **by** *metis*
next
{
  fix $\varphi\ \psi$ :: *'v propo*
  have *H*: *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \psi \Longrightarrow$ *no-T-F-except-top-level* φ
    $\Longrightarrow$ *no-T-F-except-top-level* ψ
    proof −
      assume *rel*: *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \psi$
      and *no-T-F-except-top-level* φ
      then have *no-T-F* $\varphi \lor \varphi = FF \lor \varphi = FT$
        by (*metis no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*)
      moreover {
        assume $\varphi = FF \lor \varphi = FT$
        then have *False* **using** *rel propo-rew-step-push-conn-inside* **by** *blast*

then have *no-T-F-except-top-level* ψ **by** *blast*
}
moreover {
assume *no-T-F* $\varphi \wedge \varphi \neq FF \wedge \varphi \neq FT$
then have *no-T-F* ψ **using** *rel push-conn-insidec-in-c′-symb-no-T-F* **by** *blast*
then have *no-T-F-except-top-level* ψ **using** *no-T-F-no-T-F-except-top-level* **by** *blast*
}
ultimately show *no-T-F-except-top-level* ψ **by** *blast*
qed
}
moreover {
fix *ca* :: *′v connective* **and** $\xi\ \xi'$ :: *′v propo list* **and** $\varphi\ \varphi'$ :: *′v propo*
assume *rel*: *propo-rew-step* (*push-conn-inside c c′*) $\varphi\ \varphi'$
assume *corr*: *wf-conn ca* (ξ @ φ # ξ')
then have *c*: $ca \neq CT \wedge ca \neq CF$ **by** *auto*
assume *no-T-F*: *no-T-F-symb-except-toplevel* (*conn ca* (ξ @ φ # ξ'))
have *no-T-F-symb-except-toplevel* (*conn ca* (ξ @ φ' # ξ'))
proof
have *c*: $ca \neq CT \wedge ca \neq CF$ **using** *corr* **by** *auto*
have ζ: $\forall\zeta\in$ *set* (ξ @ φ # ξ'). $\zeta\neq FT \wedge \zeta \neq FF$
using *corr no-T-F no-T-F-symb-except-toplevel-if-is-a-true-false* **by** *blast*
then have $\varphi \neq FT \wedge \varphi \neq FF$ **by** *auto*
from *rel this* **have** $\varphi' \neq FT \wedge \varphi' \neq FF$
apply (*induct rule*: *propo-rew-step.induct*)
by (*metis append-is-Nil-conv conn.simps*(*2*) *conn-inj list.distinct*(*1*)
wf-conn-helper-facts(*3*) *wf-conn-list*(*1*) *wf-conn-no-arity-change*
wf-conn-no-arity-change-helper push-conn-inside-not-true-false)+
then have $\forall\,\zeta \in$ *set* (ξ @ φ' # ξ'). $\zeta \neq FT \wedge \zeta \neq FF$ **using** ζ **by** *auto*
moreover have *wf-conn ca* (ξ @ φ' # ξ')
using *corr wf-conn-no-arity-change* **by** (*metis wf-conn-no-arity-change-helper*)
ultimately show *no-T-F-symb* (*conn ca* (ξ @ φ' # ξ')) **using** *no-T-F-symb.intros c* **by** *metis*
qed
}
ultimately show *no-T-F-except-top-level* ψ
using *full-propo-rew-step-inv-stay′*[*of push-conn-inside c c′ no-T-F-symb-except-toplevel*]
assms **unfolding** *no-T-F-except-top-level-def full-unfold* **by** *metis*

next
{
fix $\varphi\ \psi$ :: *′v propo*
have *H*: *push-conn-inside c c′* $\varphi\ \psi \Longrightarrow$ *no-equiv* $\varphi \Longrightarrow$ *no-equiv* ψ
by (*induct* $\varphi\ \psi$ *rule*: *push-conn-inside.induct*, *auto*)
}
then show *no-equiv* ψ
using *full-propo-rew-step-inv-stay-conn*[*of push-conn-inside c c′ no-equiv-symb*] *assms*
no-equiv-symb-conn-characterization **unfolding** *no-equiv-def* **by** *metis*

next
{
fix $\varphi\ \psi$ :: *′v propo*
have *H*: *push-conn-inside c c′* $\varphi\ \psi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *no-imp* ψ
by (*induct* $\varphi\ \psi$ *rule*: *push-conn-inside.induct*, *auto*)
}
then show *no-imp* ψ
using *full-propo-rew-step-inv-stay-conn*[*of push-conn-inside c c′ no-imp-symb*] *assms*
no-imp-symb-conn-characterization **unfolding** *no-imp-def* **by** *metis*

qed

lemma *push-conn-inside-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step* (*push-conn-inside c c'*)) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ **and**
$c = CAnd \lor c = COr$ **and**
$c' = CAnd \lor c' = COr$
shows *c-in-c'-only c c'* ψ
using *c-in-c'-symb-rew assms full-propo-rew-step-subformula* **by** *blast*

Only one type of connective in the formula (+ not)

inductive *only-c-inside-symb* :: *'v connective* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **for** *c* :: *'v connective* **where**
simple-only-c-inside[*simp*]: *simple* $\varphi \Longrightarrow$ *only-c-inside-symb c* φ |
simple-cnot-only-c-inside[*simp*]: *simple* $\varphi \Longrightarrow$ *only-c-inside-symb c* (*FNot* φ) |
only-c-inside-into-only-c-inside: *wf-conn c l* $\Longrightarrow$ *only-c-inside-symb c* (*conn c l*)

lemma *only-c-inside-symb-simp*[*simp*]:
only-c-inside-symb c FF only-c-inside-symb c FT only-c-inside-symb c (*FVar x*) **by** *auto*

definition *only-c-inside* **where** *only-c-inside c = all-subformula-st* (*only-c-inside-symb c*)

lemma *only-c-inside-symb-decomp*:
only-c-inside-symb c $\psi \longleftrightarrow$ (*simple* ψ
$\lor$ ($\exists$ φ'. ψ = *FNot* φ' $\land$ *simple* φ')
$\lor$ ($\exists$ *l*. ψ = *conn c l* $\land$ *wf-conn c l*))
by (*auto simp*: *only-c-inside-symb.intros*(*3*)) (*induct rule*: *only-c-inside-symb.induct*, *auto*)

lemma *only-c-inside-symb-decomp-not*[*simp*]:
fixes *c* :: *'v connective*
assumes *c*: $c \neq CNot$
shows *only-c-inside-symb c* (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
apply (*auto simp*: *only-c-inside-symb.intros*(*3*))
by (*induct FNot* ψ *rule*: *only-c-inside-symb.induct*, *auto simp*: *wf-conn-list*(*8*) *c*)

lemma *only-c-inside-decomp-not*[*simp*]:
assumes *c*: $c \neq CNot$
shows *only-c-inside c* (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
by (*metis* (*no-types*, *hide-lams*) *all-subformula-st-def all-subformula-st-test-symb-true-phi c*
only-c-inside-def only-c-inside-symb-decomp-not simple-only-c-inside
subformula-conn-decomp-simple)

lemma *only-c-inside-decomp*:
only-c-inside c $\varphi \longleftrightarrow$
($\forall \psi$. $\psi \preceq \varphi \longrightarrow$ (*simple* ψ $\lor$ ($\exists$ φ'. ψ = *FNot* φ' $\land$ *simple* φ')
$\lor$ ($\exists$ *l*. ψ = *conn c l* $\land$ *wf-conn c l*)))
unfolding *only-c-inside-def* **by** (*auto simp*: *all-subformula-st-def only-c-inside-symb-decomp*)

```
lemma only-c-inside-c-c'-false:
  fixes c c' :: 'v connective and l :: 'v propo list and φ :: 'v propo
  assumes cc': c ≠ c' and c: c = CAnd ∨ c = COr and c': c' = CAnd ∨ c' = COr
  and only: only-c-inside c φ and incl: conn c' l ≼ φ and wf: wf-conn c' l
  shows False
proof –
  let ?ψ = conn c' l
  have simple ?ψ ∨ (∃ φ'. ?ψ = FNot φ' ∧ simple φ') ∨ (∃ l. ?ψ = conn c l ∧ wf-conn c l)
    using only-c-inside-decomp only incl by blast
  moreover have ¬ simple ?ψ
    using wf simple-decomp by (metis c' connective.distinct(19) connective.distinct(7,9,21,29,31)
      wf-conn-list(1–3))
  moreover
    {
      fix φ'
      have ?ψ ≠ FNot φ' using c' conn-inj-not(1) wf by blast
    }
  ultimately obtain l :: 'v propo list where ?ψ = conn c l ∧ wf-conn c l by metis
  then have c = c' using conn-inj wf by metis
  then show False using cc' by auto
qed

lemma only-c-inside-implies-c-in-c'-symb:
  assumes δ: c ≠ c'  and c: c = CAnd ∨ c = COr and c': c' = CAnd ∨ c' = COr
  shows only-c-inside c φ ⟹ c-in-c'-symb c c' φ
  apply (rule ccontr)
  apply (cases rule: not-c-in-c'-symb.cases, auto)
  by (metis δ c c' connective.distinct(37,39) list.distinct(1) only-c-inside-c-c'-false
    subformula-in-binary-conn(1,2) wf-conn.simps)+

lemma c-in-c'-symb-decomp-level1:
  fixes l :: 'v propo list and c c' ca :: 'v connective
  shows wf-conn ca l ⟹ ca ≠ c ⟹ c-in-c'-symb c c' (conn ca l)
proof –
  have not-c-in-c'-symb c c' (conn ca l) ⟹  wf-conn ca l ⟹ ca = c
    by (induct conn ca l rule: not-c-in-c'-symb.induct, auto simp: conn-inj)
  then show wf-conn ca l ⟹ ca ≠ c ⟹ c-in-c'-symb c c' (conn ca l) by blast
qed


lemma only-c-inside-implies-c-in-c'-only:
  assumes δ: c ≠ c'  and c: c = CAnd ∨ c = COr and c': c' = CAnd ∨ c' = COr
  shows only-c-inside c φ ⟹ c-in-c'-only c c' φ
  unfolding  c-in-c'-only-def all-subformula-st-def
  using only-c-inside-implies-c-in-c'-symb
    by (metis all-subformula-st-def assms(1) c c' only-c-inside-def subformula-trans)


lemma c-in-c'-symb-c-implies-only-c-inside:
  assumes δ: c = CAnd ∨ c = COr c' = CAnd ∨ c' = COr c ≠ c' and wf: wf-conn c [φ, ψ]
  and inv: no-equiv (conn c l) no-imp (conn c l) simple-not (conn c l)
  shows wf-conn c l ⟹ c-in-c'-only c c' (conn c l) ⟹ (∀ψ∈ set l. only-c-inside c ψ)
using inv
proof (induct conn c l arbitrary: l rule: propo-induct-arity)
  case (nullary x)
```

then show *?case* **by** (*auto simp*: *wf-conn-list assms*)
next
case (*unary* φ *la*)
then have $c = CNot \wedge la = [\varphi]$ **by** (*metis* (*no-types*) *wf-conn-list*(*8*))
then show *?case* **using** *assms*(*2*) *assms*(*1*) **by** *blast*
next
case (*binary* $\varphi 1$ $\varphi 2$)
note $IH\varphi 1 = this(1)$ **and** $IH\varphi 2 = this(2)$ **and** $\varphi = this(3)$ **and** $only = this(5)$ **and** $wf = this(4)$
and $no\text{-}equiv = this(6)$ **and** $no\text{-}imp = this(7)$ **and** $simple\text{-}not = this(8)$
then have *l*: $l = [\varphi 1, \varphi 2]$ **by** (*meson wf-conn-list*(*4*−*7*))
let $?\varphi = conn\ c\ l$

obtain *c1 l1 c2 l2* **where** $\varphi 1$: $\varphi 1 = conn\ c1\ l1$ **and** $wf\varphi 1$: *wf-conn c1 l1*
and $\varphi 2$: $\varphi 2 = conn\ c2\ l2$ **and** $wf\varphi 2$: *wf-conn c2 l2* **using** *exists-c-conn* **by** *metis*
then have *c-in-only*$\varphi 1$: *c-in-c′-only c c′* (*conn c1 l1*) **and** *c-in-c′-only c c′* (*conn c2 l2*)
using *only l* **unfolding** *c-in-c′-only-def* **using** *assms*(*1*) **by** *auto*
have *inc*$\varphi 1$: $\varphi 1 \preceq ?\varphi$ **and** *inc*$\varphi 2$: $\varphi 2 \preceq ?\varphi$
using $\varphi 1$ $\varphi 2$ φ *local.wf* **by** (*metis conn.simps*(*5*−*8*) *helper-fact subformula-in-binary-conn*(*1*,*2*))+

have *c1-eq*: $c1 \neq CEq$ **and** *c2-eq*: $c2 \neq CEq$
unfolding *no-equiv-def* **using** *inc*$\varphi 1$ *inc*$\varphi 2$ **by** (*metis* $\varphi 1$ $\varphi 2$ *wf*$\varphi 1$ *wf*$\varphi 2$ *assms*(*1*) *no-equiv*
no-equiv-eq(*1*) *no-equiv-symb.elims*(*3*) *no-equiv-symb-conn-characterization wf-conn-list*(*4*,*5*)
no-equiv-def subformula-all-subformula-st)+
have *c1-imp*: $c1 \neq CImp$ **and** *c2-imp*: $c2 \neq CImp$
using *no-imp* **by** (*metis* $\varphi 1$ $\varphi 2$ *all-subformula-st-decomp-explicit-imp*(*2*,*3*) *assms*(*1*)
conn.simps(*5*,*6*) *l no-imp-Imp*(*1*) *no-imp-symb.elims*(*3*) *no-imp-symb-conn-characterization*
wf$\varphi 1$ *wf*$\varphi 2$ *all-subformula-st-decomp no-imp-symb-conn-characterization*)+
have *c1c*: $c1 \neq c'$
proof
assume *c1c*: $c1 = c'$
then obtain $\xi 1$ $\xi 2$ **where** *l1*: $l1 = [\xi 1, \xi 2]$
by (*metis assms*(*2*) *connective.distinct*(*37*,*39*) *helper-fact wf*$\varphi 1$ *wf-conn.simps*
wf-conn-list-decomp(*1*−*3*))
have *c-in-c′-only c c′* (*conn c* [*conn c′ l1*, $\varphi 2$]) **using** *c1c l only* $\varphi 1$ **by** *auto*
moreover have *not-c-in-c′-symb c c′* (*conn c* [*conn c′ l1*, $\varphi 2$])
using *l1* $\varphi 1$ *c1c l local.wf not-c-in-c′-symb-l wf*$\varphi 1$ **by** *blast*
ultimately show *False* **using** $\varphi 1$ *c1c l l1 local.wf not-c-in-c′-simp*(*4*) *wf*$\varphi 1$ **by** *blast*
qed
then have $(\varphi 1 = conn\ c\ l1 \wedge wf\text{-}conn\ c\ l1) \vee (\exists \psi 1.\ \varphi 1 = FNot\ \psi 1) \vee simple\ \varphi 1$
by (*metis* $\varphi 1$ *assms*(*1*−*3*) *c1-eq c1-imp simple.elims*(*3*) *wf*$\varphi 1$ *wf-conn-list*(*4*) *wf-conn-list*(*5*−*7*))
moreover {
assume $\varphi 1 = conn\ c\ l1 \wedge wf\text{-}conn\ c\ l1$
then have *only-c-inside c* $\varphi 1$
by (*metis IH*$\varphi 1$ $\varphi 1$ *all-subformula-st-decomp-imp inc*$\varphi 1$ *no-equiv no-equiv-def no-imp no-imp-def*
c-in-only$\varphi 1$ *only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def*
subformula-all-subformula-st)
}
moreover {
assume $\exists \psi 1.\ \varphi 1 = FNot\ \psi 1$
then obtain $\psi 1$ **where** $\varphi 1 = FNot\ \psi 1$ **by** *metis*
then have *only-c-inside c* $\varphi 1$
by (*metis all-subformula-st-def assms*(*1*) *connective.distinct*(*37*,*39*) *inc*$\varphi 1$
only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(*1*))
}
moreover {
assume *simple* $\varphi 1$

then have *only-c-inside c φ1*
by (*metis all-subformula-st-decomp-explicit*(*3*) *assms*(*1*) *connective.distinct*(*37*,*39*)
only-c-inside-decomp-not only-c-inside-def)
}
ultimately have *only-c-insideφ1*: *only-c-inside c φ1* **by** *metis*

have *c-in-onlyφ2*: *c-in-c′-only c c′* (*conn c2 l2*)
using *only l φ2 wfφ2 assms* **unfolding** *c-in-c′-only-def* **by** *auto*
have *c2c*: *c2* ≠ *c′*
proof
assume *c2c*: *c2* = *c′*
then obtain *ξ1 ξ2* **where** *l2*: *l2* = [*ξ1*, *ξ2*]
by (*metis assms*(*2*) *wfφ2 wf-conn.simps connective.distinct*(*7*,*9*,*19*,*21*,*29*,*31*,*37*,*39*))
then have *c-in-c′-symb c c′* (*conn c* [*φ1*, *conn c′ l2*])
using *c2c l only φ2 all-subformula-st-test-symb-true-phi* **unfolding** *c-in-c′-only-def* **by** *auto*
moreover have *not-c-in-c′-symb c c′* (*conn c* [*φ1*, *conn c′ l2*])
using *assms*(*1*) *c2c l2 not-c-in-c′-symb-r wfφ2 wf-conn-helper-facts*(*5*,*6*) **by** *metis*
ultimately show *False* **by** *auto*
qed
then have (*φ2* = *conn c l2* ∧ *wf-conn c l2*) ∨ (∃*ψ2*. *φ2* = *FNot ψ2*) ∨ *simple φ2*
using *c2-eq* **by** (*metis φ2 assms*(*1*−*3*) *c2-eq c2-imp simple.elims*(*3*) *wfφ2 wf-conn-list*(*4*−*7*))
moreover {
assume *φ2* = *conn c l2* ∧ *wf-conn c l2*
then have *only-c-inside c φ2*
by (*metis IHφ2 φ2 all-subformula-st-decomp incφ2 no-equiv no-equiv-def no-imp no-imp-def*
c-in-onlyφ2 only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
subformula-all-subformula-st)
}
moreover {
assume ∃*ψ2*. *φ2* = *FNot ψ2*
then obtain *ψ2* **where** *φ2* = *FNot ψ2* **by** *metis*
then have *only-c-inside c φ2*
by (*metis all-subformula-st-def assms*(*1*−*3*) *connective.distinct*(*38*,*40*) *incφ2*
only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(*1*))
}
moreover {
assume *simple φ2*
then have *only-c-inside c φ2*
by (*metis all-subformula-st-decomp-explicit*(*3*) *assms*(*1*) *connective.distinct*(*37*,*39*)
only-c-inside-decomp-not only-c-inside-def)
}
ultimately have *only-c-insideφ2*: *only-c-inside c φ2* **by** *metis*
show *?case* **using** *l only-c-insideφ1 only-c-insideφ2* **by** *auto*
qed

Push Conjunction

definition *pushConj* **where** *pushConj* = *push-conn-inside CAnd COr*

lemma *pushConj-consistent*: *preserves-un-sat pushConj*
unfolding *pushConj-def* **by** (*simp add*: *push-conn-inside-consistent*)

definition *and-in-or-symb* **where** *and-in-or-symb* = *c-in-c′-symb CAnd COr*

definition *and-in-or-only* **where**
and-in-or-only = *all-subformula-st* (*c-in-c′-symb CAnd COr*)

lemma *pushConj-inv*:
fixes φ ψ :: $'v$ *propo*
assumes *full* (*propo-rew-step pushConj*) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
using *push-conn-inside-inv assms* **unfolding** *pushConj-def* **by** *metis+*

lemma *pushConj-full-propo-rew-step*:
fixes φ ψ :: $'v$ *propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step pushConj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *and-in-or-only* ψ
using *assms push-conn-inside-full-propo-rew-step*
unfolding *pushConj-def and-in-or-only-def c-in-c′-only-def* **by** (*metis* (*no-types*))

Push Disjunction

definition *pushDisj* **where** *pushDisj* = *push-conn-inside COr CAnd*

lemma *pushDisj-consistent*: *preserves-un-sat pushDisj*
unfolding *pushDisj-def* **by** (*simp add*: *push-conn-inside-consistent*)

definition *or-in-and-symb* **where** *or-in-and-symb* = *c-in-c′-symb COr CAnd*

definition *or-in-and-only* **where**
or-in-and-only = *all-subformula-st* (*c-in-c′-symb COr CAnd*)

lemma *not-or-in-and-only-or-and*[*simp*]:
~*or-in-and-only* (*FOr* (*FAnd* $\psi 1$ $\psi 2$) φ')
unfolding *or-in-and-only-def*
by (*metis all-subformula-st-test-symb-true-phi conn.simps*(5−6) *not-c-in-c′-symb-l*
wf-conn-helper-facts(5) *wf-conn-helper-facts*(6))

lemma *pushDisj-inv*:
fixes φ ψ :: $'v$ *propo*
assumes *full* (*propo-rew-step pushDisj*) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
using *push-conn-inside-inv assms* **unfolding** *pushDisj-def* **by** *metis+*

lemma *pushDisj-full-propo-rew-step*:
fixes φ ψ :: $'v$ *propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step pushDisj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *or-in-and-only* ψ

using *assms push-conn-inside-full-propo-rew-step*
unfolding *pushDisj-def or-in-and-only-def c-in-c′-only-def* **by** (*metis* (*no-types*))

3.6 The full transformations

3.6.1 Abstract Property characterizing that only some connective are inside the others

Definition

The normal is a super group of groups

inductive *grouped-by* :: *′a connective* ⇒ *′a propo* ⇒ *bool* **for** *c* **where**
simple-is-grouped[*simp*]: *simple* $\varphi \Longrightarrow$ *grouped-by c* φ |
simple-not-is-grouped[*simp*]: *simple* $\varphi \Longrightarrow$ *grouped-by c* (*FNot* φ) |
connected-is-group[*simp*]: *grouped-by c* $\varphi \Longrightarrow$ *grouped-by c* $\psi \Longrightarrow$ *wf-conn c* [φ, ψ]
$\Longrightarrow$ *grouped-by c* (*conn c* [φ, ψ])

lemma *simple-clause*[*simp*]:
grouped-by c FT
grouped-by c FF
grouped-by c (*FVar x*)
grouped-by c (*FNot FT*)
grouped-by c (*FNot FF*)
grouped-by c (*FNot* (*FVar x*))
by *simp+*

lemma *only-c-inside-symb-c-eq-c′*:
only-c-inside-symb c (*conn c′* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \lor c' = COr$ $\Longrightarrow$ *wf-conn c′* [$\varphi 1$, $\varphi 2$]
$\Longrightarrow$ $c' = c$
by (*induct conn c′* [$\varphi 1$, $\varphi 2$] *rule*: *only-c-inside-symb.induct*, *auto simp*: *conn-inj*)

lemma *only-c-inside-c-eq-c′*:
only-c-inside c (*conn c′* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \lor c' = COr$ $\Longrightarrow$ *wf-conn c′* [$\varphi 1$, $\varphi 2$] $\Longrightarrow$ $c = c'$
unfolding *only-c-inside-def all-subformula-st-def* **using** *only-c-inside-symb-c-eq-c′ subformula-refl*
by *blast*

lemma *only-c-inside-imp-grouped-by*:
assumes *c*: $c \neq CNot$ **and** *c′*: $c' = CAnd \lor c' = COr$
shows *only-c-inside c* $\varphi \Longrightarrow$ *grouped-by c* φ (**is** *?O* $\varphi \Longrightarrow$ *?G* φ)
proof (*induct* φ *rule*: *propo-induct-arity*)
case (*nullary* φ *x*)
then show *?G* φ **by** *auto*
next
case (*unary* ψ)
then show *?G* (*FNot* ψ) **by** (*auto simp*: *c*)
next
case (*binary* φ $\varphi 1$ $\varphi 2$)
note *IH*$\varphi 1$ = *this*(*1*) **and** *IH*$\varphi 2$ = *this*(*2*) **and** φ = *this*(*3*) **and** *only* = *this*(*4*)
have φ*-conn*: φ = *conn c* [$\varphi 1$, $\varphi 2$] **and** *wf*: *wf-conn c* [$\varphi 1$, $\varphi 2$]
proof −
obtain *c″ l″* **where** φ*-c″*: φ = *conn c″ l″* **and** *wf*: *wf-conn c″ l″*
using *exists-c-conn* **by** *metis*
then have *l″*: *l″* = [$\varphi 1$, $\varphi 2$] **using** φ **by** (*metis wf-conn-list*(*4*−*7*))
have *only-c-inside-symb c* (*conn c″* [$\varphi 1$, $\varphi 2$])

using *only all-subformula-st-test-symb-true-phi*
unfolding *only-c-inside-def* φ*-c''* *l''* **by** *metis*
then have $c = c''$
by (*metis* φ φ*-c''* *conn-inj conn-inj-not*(*2*) *l''* *list.distinct*(*1*) *list.inject wf* *only-c-inside-symb.cases simple.simps*(*5*−*8*))
then show φ = *conn c* [φ*1*, φ*2*] **and** *wf-conn c* [φ*1*, φ*2*] **using** φ*-c''* *wf l''* **by** *auto*
qed
have *grouped-by c* φ*1* **using** *wf IH*φ*1 IH*φ*2* φ*-conn only* φ **unfolding** *only-c-inside-def* **by** *auto*
moreover have *grouped-by c* φ*2*
using *wf* φ *IH*φ*1 IH*φ*2* φ*-conn only* **unfolding** *only-c-inside-def* **by** *auto*
ultimately show *?G* φ **using** φ*-conn connected-is-group local.wf* **by** *blast*
qed

lemma *grouped-by-false*:
grouped-by c (*conn c'* [φ, ψ]) $\Longrightarrow c \neq c' \Longrightarrow$ *wf-conn c'* [φ, ψ] $\Longrightarrow$ *False*
apply (*induct conn c'* [φ, ψ] *rule*: *grouped-by.induct*)
apply (*auto simp*: *simple-decomp wf-conn-list*, *auto simp*: *conn-inj*)
by (*metis list.distinct*(*1*) *list.sel*(*3*) *wf-conn-list*(*8*))+

Then the CNF form is a conjunction of clauses: every clause is in CNF form and two formulas in CNF form can be related by an and.

inductive *super-grouped-by*:: *'a connective* ⇒ *'a connective* ⇒ *'a propo* ⇒ *bool* **for** *c c'* **where**
grouped-is-super-grouped[*simp*]: *grouped-by c* $\varphi \Longrightarrow$ *super-grouped-by c c'* φ |
connected-is-super-group: *super-grouped-by c c'* $\varphi \Longrightarrow$ *super-grouped-by c c'* $\psi \Longrightarrow$ *wf-conn c* [φ, ψ]
$\Longrightarrow$ *super-grouped-by c c'* (*conn c'* [φ, ψ])

lemma *simple-cnf*[*simp*]:
super-grouped-by c c' FT
super-grouped-by c c' FF
super-grouped-by c c' (*FVar x*)
super-grouped-by c c' (*FNot FT*)
super-grouped-by c c' (*FNot FF*)
super-grouped-by c c' (*FNot* (*FVar x*))
by *auto*

lemma *c-in-c'-only-super-grouped-by*:
assumes *c*: *c* = *CAnd* ∨ *c* = *COr* **and** *c'*: *c'* = *CAnd* ∨ *c'* = *COr* **and** *cc'*: $c \neq c'$
shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *c-in-c'-only c c'* φ
$\Longrightarrow$ *super-grouped-by c c'* φ
(**is** *?NE* $\varphi \Longrightarrow$ *?NI* $\varphi \Longrightarrow$ *?SN* $\varphi \Longrightarrow$ *?C* $\varphi \Longrightarrow$ *?S* φ)
proof (*induct* φ *rule*: *propo-induct-arity*)
case (*nullary* φ *x*)
then show *?S* φ **by** *auto*
next
case (*unary* φ)
then have *simple-not-symb* (*FNot* φ)
using *all-subformula-st-test-symb-true-phi* **unfolding** *simple-not-def* **by** *blast*
then have φ = *FT* ∨ φ = *FF* ∨ (∃ *x*. φ = *FVar x*) **by** (*cases* φ, *auto*)
then show *?S* (*FNot* φ) **by** *auto*
next
case (*binary* φ φ*1* φ*2*)
note *IH*φ*1* = *this*(*1*) **and** *IH*φ*2* = *this*(*2*) **and** *no-equiv* = *this*(*4*) **and** *no-imp* = *this*(*5*)
and *simpleN* = *this*(*6*) **and** *c-in-c'-only* = *this*(*7*) **and** φ' = *this*(*3*)
{
assume φ = *FImp* φ*1* φ*2* ∨ φ = *FEq* φ*1* φ*2*

then have *False* **using** *no-equiv no-imp* **by** *auto*
then have *?S* φ **by** *auto*
}
moreover {
assume φ: φ = *conn* c' [$\varphi 1$, $\varphi 2$] $\wedge$ *wf-conn* c' [$\varphi 1$, $\varphi 2$]
have *c-in-c′-only*: *c-in-c′-only c c′* $\varphi 1$ $\wedge$ *c-in-c′-only c c′* $\varphi 2$ $\wedge$ *c-in-c′-symb c c′* φ
using *c-in-c′-only* φ' **unfolding** *c-in-c′-only-def* **by** *auto*
have *super-grouped-by c c′* $\varphi 1$ **using** φ *c′ no-equiv no-imp simpleN IH*$\varphi 1$ *c-in-c′-only* **by** *auto*
moreover have *super-grouped-by c c′* $\varphi 2$
using φ *c′ no-equiv no-imp simpleN IH*$\varphi 2$ *c-in-c′-only* **by** *auto*
ultimately have *?S* φ
using *super-grouped-by.intros*(*2*) φ **by** (*metis c wf-conn-helper-facts*(*5*,*6*))
}
moreover {
assume φ: φ = *conn c* [$\varphi 1$, $\varphi 2$] $\wedge$ *wf-conn c* [$\varphi 1$, $\varphi 2$]
then have *only-c-inside c* $\varphi 1$ $\wedge$ *only-c-inside c* $\varphi 2$
using *c-in-c′-symb-c-implies-only-c-inside c c′ c-in-c′-only list.set-intros*(*1*)
wf-conn-helper-facts(*5*,*6*) *no-equiv no-imp simpleN last-ConsL last-ConsR last-in-set*
list.distinct(*1*) **by** (*metis* (*no-types*, *hide-lams*) *cc′*)
then have *only-c-inside c* (*conn c* [$\varphi 1$, $\varphi 2$])
unfolding *only-c-inside-def* **using** φ
by (*simp add*: *only-c-inside-into-only-c-inside all-subformula-st-decomp*)
then have *grouped-by c* φ **using** φ *only-c-inside-imp-grouped-by c* **by** *blast*
then have *?S* φ **using** *super-grouped-by.intros*(*1*) **by** *metis*
}
ultimately show *?S* φ **by** (*metis* φ' *c c′ cc′ conn.simps*(*5*,*6*) *wf-conn-helper-facts*(*5*,*6*))
qed

3.6.2 Conjunctive Normal Form

definition *is-conj-with-TF* **where** *is-conj-with-TF* == *super-grouped-by COr CAnd*

lemma *or-in-and-only-conjunction-in-disj*:
shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *or-in-and-only* $\varphi \Longrightarrow$ *is-conj-with-TF* φ
using *c-in-c′-only-super-grouped-by*
unfolding *is-conj-with-TF-def or-in-and-only-def c-in-c′-only-def*
by (*simp add*: *c-in-c′-only-def c-in-c′-only-super-grouped-by*)

definition *is-cnf* **where**
is-cnf $\varphi \equiv$ *is-conj-with-TF* φ $\wedge$ *no-T-F-except-top-level* φ

Full CNF transformation

The full1 CNF transformation consists simply in chaining all the transformation defined before.

definition *cnf-rew* **where** *cnf-rew* =
(*full* (*propo-rew-step elim-equiv*)) *OO*
(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step elimTB*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushDisj*))

lemma *cnf-rew-consistent*: *preserves-un-sat cnf-rew*
by (*simp add*: *cnf-rew-def elimEquv-lifted-consistant elim-imp-lifted-consistant elimTB-consistent*
preserves-un-sat-OO pushDisj-consistent pushNeg-lifted-consistant)

lemma *cnf-rew-is-cnf*: *cnf-rew* φ φ' $\Longrightarrow$ *is-cnf* φ'
  apply (*unfold cnf-rew-def OO-def*)
  apply *auto*
proof −
  fix φ φEq φImp φTB φNeg $\varphi Disj$ :: *'v propo*
  assume *Eq*: *full* (*propo-rew-step elim-equiv*) φ φEq
  then have *no-equiv*: *no-equiv* φEq **using** *no-equiv-full-propo-rew-step-elim-equiv* **by** *blast*

  assume *Imp*: *full* (*propo-rew-step elim-imp*) φEq φImp
  then have *no-imp*: *no-imp* φImp **using** *no-imp-full-propo-rew-step-elim-imp* **by** *blast*
  have *no-imp-inv*: *no-equiv* φImp **using** *no-equiv Imp elim-imp-inv* **by** *blast*

  assume *TB*: *full* (*propo-rew-step elimTB*) φImp φTB
  then have *noTB*: *no-T-F-except-top-level* φTB
    using *no-imp-inv no-imp elimTB-full-propo-rew-step* **by** *blast*
  have *noTB-inv*: *no-equiv* φTB *no-imp* φTB **using** *elimTB-inv TB no-imp no-imp-inv* **by** *blast+*

  assume *Neg*: *full* (*propo-rew-step pushNeg*) φTB φNeg
  then have *noNeg*: *simple-not* φNeg
    using *noTB-inv noTB pushNeg-full-propo-rew-step* **by** *blast*
  have *noNeg-inv*: *no-equiv* φNeg *no-imp* φNeg *no-T-F-except-top-level* φNeg
    using *pushNeg-inv Neg noTB noTB-inv* **by** *blast+*

  assume *Disj*: *full* (*propo-rew-step pushDisj*) φNeg $\varphi Disj$
  then have *no-Disj*: *or-in-and-only* $\varphi Disj$
    using *noNeg-inv noNeg pushDisj-full-propo-rew-step* **by** *blast*
  have *noDisj-inv*: *no-equiv* $\varphi Disj$ *no-imp* $\varphi Disj$ *no-T-F-except-top-level* $\varphi Disj$
    simple-not $\varphi Disj$
  using *pushDisj-inv Disj noNeg noNeg-inv* **by** *blast+*

  moreover have *is-conj-with-TF* $\varphi Disj$
    using *or-in-and-only-conjunction-in-disj noDisj-inv no-Disj* **by** *blast*
  ultimately show *is-cnf* $\varphi Disj$ **unfolding** *is-cnf-def* **by** *blast*
qed

3.6.3 Disjunctive Normal Form

definition *is-disj-with-TF* **where** *is-disj-with-TF* $\equiv$ *super-grouped-by CAnd COr*

lemma *and-in-or-only-conjunction-in-disj*:
  shows *no-equiv* φ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *simple-not* φ $\Longrightarrow$ *and-in-or-only* φ $\Longrightarrow$ *is-disj-with-TF* φ
  using *c-in-c'-only-super-grouped-by*
  unfolding *is-disj-with-TF-def and-in-or-only-def c-in-c'-only-def*
  by (*simp add*: *c-in-c'-only-def c-in-c'-only-super-grouped-by*)

definition *is-dnf* :: *'a propo* $\Rightarrow$ *bool* **where**
is-dnf φ $\longleftrightarrow$ *is-disj-with-TF* φ $\wedge$ *no-T-F-except-top-level* φ

Full DNF transform

The full1 DNF transformation consists simply in chaining all the transformation defined before.

definition *dnf-rew* **where** *dnf-rew* $\equiv$
  (*full* (*propo-rew-step elim-equiv*)) *OO*
  (*full* (*propo-rew-step elim-imp*)) *OO*

(*full* (*propo-rew-step elimTB*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushConj*))

lemma *dnf-rew-consistent*: *preserves-un-sat dnf-rew*
by (*simp add*: *dnf-rew-def elimEquv-lifted-consistant elim-imp-lifted-consistant elimTB-consistent preserves-un-sat-OO pushConj-consistent pushNeg-lifted-consistant*)

theorem *dnf-transformation-correction*:
dnf-rew φ φ' $\Longrightarrow$ *is-dnf* φ'
apply (*unfold dnf-rew-def OO-def*)
by (*meson and-in-or-only-conjunction-in-disj elimTB-full-propo-rew-step elimTB-inv*(*1*,*2*) *elim-imp-inv is-dnf-def no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv*(*1*−*4*) *pushNeg-full-propo-rew-step pushNeg-inv*(*1*−*3*))

3.7 More aggressive simplifications: Removing true and false at the beginning

3.7.1 Transformation

We should remove *FT* and *FF* at the beginning and not in the middle of the algorithm. To do this, we have to use more rules (one for each connective):

inductive *elimTBFull* **where**
ElimTBFull1[*simp*]: *elimTBFull* (*FAnd* φ *FT*) φ |
ElimTBFull1'[*simp*]: *elimTBFull* (*FAnd FT* φ) φ |

ElimTBFull2[*simp*]: *elimTBFull* (*FAnd* φ *FF*) *FF* |
ElimTBFull2'[*simp*]: *elimTBFull* (*FAnd FF* φ) *FF* |

ElimTBFull3[*simp*]: *elimTBFull* (*FOr* φ *FT*) *FT* |
ElimTBFull3'[*simp*]: *elimTBFull* (*FOr FT* φ) *FT* |

ElimTBFull4[*simp*]: *elimTBFull* (*FOr* φ *FF*) φ |
ElimTBFull4'[*simp*]: *elimTBFull* (*FOr FF* φ) φ |

ElimTBFull5[*simp*]: *elimTBFull* (*FNot FT*) *FF* |
ElimTBFull5'[*simp*]: *elimTBFull* (*FNot FF*) *FT* |

ElimTBFull6-l[*simp*]: *elimTBFull* (*FImp FT* φ) φ |
ElimTBFull6-l'[*simp*]: *elimTBFull* (*FImp FF* φ) *FT* |
ElimTBFull6-r[*simp*]: *elimTBFull* (*FImp* φ *FT*) *FT* |
ElimTBFull6-r'[*simp*]: *elimTBFull* (*FImp* φ *FF*) (*FNot* φ) |

ElimTBFull7-l[*simp*]: *elimTBFull* (*FEq FT* φ) φ |
ElimTBFull7-l'[*simp*]: *elimTBFull* (*FEq FF* φ) (*FNot* φ) |
ElimTBFull7-r[*simp*]: *elimTBFull* (*FEq* φ *FT*) φ |
ElimTBFull7-r'[*simp*]: *elimTBFull* (*FEq* φ *FF*) (*FNot* φ)

The transformation is still consistent.

lemma *elimTBFull-consistent*: *preserves-un-sat elimTBFull*
proof −
{
fix φ ψ:: *'b propo*

have *elimTBFull* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct-tac rule*: *elimTBFull.inducts*, *auto*)
}
then show *?thesis* **using** *preserves-un-sat-def* **by** *auto*
qed

Contrary to the theorem *no-T-F-symb-except-toplevel-step-exists*, we do not need the assumption *no-equiv* φ and *no-imp* φ, since our transformation is more general.

lemma *no-T-F-symb-except-toplevel-step-exists'*:
fixes φ :: *'v propo*
shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'.$ *elimTBFull* ψ ψ'
proof (*induct* ψ *rule*: *propo-induct-arity*)
case (*nullary* φ')
then have *False* **using** *no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false* **by** *auto*
then show *Ex* (*elimTBFull* φ') **by** *blast*
next
case (*unary* ψ)
then have $\psi = FF \vee \psi = FT$ **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*
then show *Ex* (*elimTBFull* (*FNot* ψ)) **using** *ElimTBFull5 ElimTBFull5'* **by** *blast*
next
case (*binary* φ' $\psi 1$ $\psi 2$)
then have $\psi 1 = FT \vee \psi 2 = FT \vee \psi 1 = FF \vee \psi 2 = FF$
by (*metis binary-connectives-def conn.simps*(*5*−*8*) *insertI1 insert-commute no-T-F-symb-except-toplevel-bin-decom binary.hyps*(*3*))
then show *Ex* (*elimTBFull* φ') **using** *elimTBFull.intros binary.hyps*(*3*) **by** *blast*
qed

The same applies here. We do not need the assumption, but the deep link between $\neg$ *no-T-F-except-top-level* φ and the existence of a rewriting step, still exists.

lemma *no-T-F-except-top-level-rew'*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$ *no-T-F-except-top-level* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elimTBFull* ψ ψ'
proof −
have *test-symb-false-nullary*:
$\forall x.$ *no-T-F-symb-except-toplevel* (*FF*:: *'v propo*) $\wedge$ *no-T-F-symb-except-toplevel FT* $\wedge$ *no-T-F-symb-except-toplevel* (*FVar* (*x*:: *'v*))
by *auto*
moreover {
fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** ψ :: *'v propo*
have *H*: *elimTBFull* (*conn c l*) $\psi \Longrightarrow \neg$*no-T-F-symb-except-toplevel* (*conn c l*)
by (*cases conn c l rule*: *elimTBFull.cases*) *auto*
}
ultimately show *?thesis*
using *no-test-symb-step-exists*[*of no-T-F-symb-except-toplevel* φ *elimTBFull*] *noTB no-T-F-symb-except-toplevel-step-exists'* **unfolding** *no-T-F-except-top-level-def* **by** *metis*
qed

lemma *elimTBFull-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elimTBFull*) φ ψ
shows *no-T-F-except-top-level* ψ
using *full-propo-rew-step-subformula no-T-F-except-top-level-rew' assms* **by** *fastforce*

3.7.2 More invariants

As the aim is to use the transformation as the first transformation, we have to show some more invariants for *elim-equiv* and *elim-imp*. For the other transformation, we have already proven it.

lemma *propo-rew-step-ElimEquiv-no-T-F*: *propo-rew-step elim-equiv* φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
proof (*induct rule*: *propo-rew-step.induct*)
  fix φ' :: *'v propo* **and** ψ' :: *'v propo*
  assume *a1*: *no-T-F* φ'
  assume *a2*: *elim-equiv* φ' ψ'
  have $\forall$ *x0 x1*. ($\neg$ *elim-equiv* (*x1* :: *'v propo*) *x0* $\vee$ ($\exists$ *v2 v3 v4 v5 v6 v7*. *x1* = *FEq v2 v3*
    $\wedge$ *x0* = *FAnd* (*FImp v4 v5*) (*FImp v6 v7*) $\wedge$ *v2* = *v4* $\wedge$ *v4* = *v7* $\wedge$ *v3* = *v5* $\wedge$ *v3* = *v6*))
  = ($\neg$ *elim-equiv x1 x0* $\vee$ ($\exists$ *v2 v3 v4 v5 v6 v7*. *x1* = *FEq v2 v3*
    $\wedge$ *x0* = *FAnd* (*FImp v4 v5*) (*FImp v6 v7*) $\wedge$ *v2* = *v4* $\wedge$ *v4* = *v7* $\wedge$ *v3* = *v5* $\wedge$ *v3* = *v6*))
    by *meson*
  then have $\forall$ *p pa*. $\neg$ *elim-equiv* (*p* :: *'v propo*) *pa* $\vee$ ($\exists$ *pb pc pd pe pf pg*. *p* = *FEq pb pc*
    $\wedge$ *pa* = *FAnd* (*FImp pd pe*) (*FImp pf pg*) $\wedge$ *pb* = *pd* $\wedge$ *pd* = *pg* $\wedge$ *pc* = *pe* $\wedge$ *pc* = *pf*)
    using *elim-equiv.cases* **by** *force*
  then show *no-T-F* ψ' **using** *a1 a2* **by** *fastforce*
next
  fix φ φ' :: *'v propo* **and** ξ ξ' :: *'v propo list* **and** *c* :: *'v connective*
  assume *rel*: *propo-rew-step elim-equiv* φ φ'
  and *IH*: *no-T-F* $\varphi \Longrightarrow$ *no-T-F* φ'
  and *corr*: *wf-conn c* (ξ @ φ # ξ')
  and *no-T-F*: *no-T-F* (*conn c* (ξ @ φ # ξ'))
  {
    assume *c*: *c* = *CNot*
    then have *empty*: ξ = [] ξ' = [] **using** *corr* **by** *auto*
    then have *no-T-F* φ **using** *no-T-F c no-T-F-decomp-not* **by** *auto*
    then have *no-T-F* (*conn c* (ξ @ φ' # ξ')) **using** *c empty no-T-F-comp-not IH* **by** *auto*
  }
  moreover {
    assume *c*: *c* $\in$ *binary-connectives*
    obtain *a b* **where** *ab*: ξ @ φ # ξ' = [*a*, *b*]
      using *corr c list-length2-decomp wf-conn-bin-list-length* **by** *metis*
    then have φ: φ = *a* $\vee$ φ = *b*
      by (*metis append.simps*(*1*) *append-is-Nil-conv list.distinct*(*1*) *list.sel*(*3*) *nth-Cons-0*
        tl-append2)
    have ζ: $\forall\zeta\in$ *set* (ξ @ φ # ξ'). *no-T-F* ζ
      using *no-T-F* **unfolding** *no-T-F-def* **using** *corr all-subformula-st-decomp* **by** *blast*

    then have φ': *no-T-F* φ' **using** *ab IH* φ **by** *auto*
    have *l'*: ξ @ φ' # ξ' = [φ', *b*] $\vee$ ξ @ φ' # ξ' = [*a*, φ']
      by (*metis* (*no-types*, *hide-lams*) *ab append-Cons append-Nil append-Nil2 butlast.simps*(*2*)
        butlast-append list.distinct(*1*) *list.sel*(*3*))
    then have $\forall\zeta \in$ *set* (ξ @ φ' # ξ'). *no-T-F* ζ **using** ζ φ' *ab* **by** *fastforce*
    moreover
      have $\forall\zeta \in$ *set* (ξ @ φ # ξ'). $\zeta \neq$ *FT* $\wedge$ $\zeta \neq$ *FF*
        using ζ *corr no-T-F no-T-F-except-top-level-false no-T-F-no-T-F-except-top-level* **by** *blast*
      then have *no-T-F-symb* (*conn c* (ξ @ φ' # ξ'))
        by (*metis* φ' *l' ab all-subformula-st-test-symb-true-phi c list.distinct*(*1*)
          list.set-intros(*1*,*2*) *no-T-F-symb-except-toplevel-bin-decom*
          no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(*1*,*2*) *no-T-F-def wf-conn-binary*
          wf-conn-list(*1*,*2*))
    ultimately have *no-T-F* (*conn c* (ξ @ φ' # ξ'))

```
      by (metis l' all-subformula-st-decomp-imp c no-T-F-def wf-conn-binary)
  }
  moreover {
    fix x
    assume c = CVar x ∨ c = CF ∨ c = CT
    then have False using corr by auto
    then have  no-T-F (conn c (ξ @ φ' # ξ')) by auto
  }
  ultimately show no-T-F (conn c (ξ @ φ' # ξ')) using corr wf-conn.cases by metis
qed

lemma elim-equiv-inv':
  fixes φ ψ :: 'v propo
  assumes full (propo-rew-step elim-equiv) φ ψ and no-T-F-except-top-level φ
  shows no-T-F-except-top-level ψ
proof -
  {
   fix φ ψ :: 'v propo
   have propo-rew-step elim-equiv φ ψ ⟹ no-T-F-except-top-level φ
     ⟹ no-T-F-except-top-level ψ
     proof -
       assume rel: propo-rew-step elim-equiv φ ψ
       and no: no-T-F-except-top-level φ
       {
         assume φ = FT ∨ φ = FF
         from rel this have False
           apply (induct rule: propo-rew-step.induct, auto simp: wf-conn-list(1,2))
           using elim-equiv.simps by blast+
         then have no-T-F-except-top-level ψ by blast
       }
       moreover {
         assume φ ≠ FT ∧ φ ≠ FF
         then have no-T-F φ
           by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
         then have no-T-F ψ using propo-rew-step-ElimEquiv-no-T-F rel by blast
         then have no-T-F-except-top-level ψ by (simp add: no-T-F-no-T-F-except-top-level)
       }
       ultimately show no-T-F-except-top-level ψ by metis
     qed
  }
  moreover {
    fix c :: 'v connective and ξ ξ' :: 'v propo list and ζ ζ' :: 'v propo
    assume rel: propo-rew-step elim-equiv ζ ζ'
    and incl: ζ ≼ φ
    and corr: wf-conn c (ξ @ ζ # ξ')
    and no-T-F: no-T-F-symb-except-toplevel (conn c (ξ @ ζ # ξ'))
    and n: no-T-F-symb-except-toplevel ζ'
    have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
    proof
      have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
        using corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F
        by blast
      have l: ∀φ∈set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
        using corr wf-conn-no-T-F-symb-iff p by blast
      from rel incl have ζ'≠FT ∧ζ'≠FF
        apply (induction ζ ζ' rule: propo-rew-step.induct)
```

```
      apply (cases rule: elim-equiv.cases, auto simp: elim-equiv.simps)
      by (metis append-is-Nil-conv list.distinct wf-conn-list(1,2) wf-conn-no-arity-change
        wf-conn-no-arity-change-helper)+
    then have ∀φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
    moreover have c ≠ CT ∧ c ≠ CF using corr by auto
    ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
      by (metis corr wf-conn-no-arity-change wf-conn-no-arity-change-helper no-T-F-symb-comp)
  qed
 }
 ultimately show no-T-F-except-top-level ψ
   using full-propo-rew-step-inv-stay-with-inc[of elim-equiv no-T-F-symb-except-toplevel φ]
     assms subformula-refl unfolding no-T-F-except-top-level-def by metis
qed


lemma propo-rew-step-ElimImp-no-T-F: propo-rew-step elim-imp φ ψ ⟹ no-T-F φ ⟹ no-T-F ψ
proof (induct rule: propo-rew-step.induct)
 case (global-rel φ' ψ')
 then show no-T-F ψ'
   using elim-imp.cases no-T-F-comp-not no-T-F-decomp(1,2)
   by (metis no-T-F-comp-expanded-explicit(2))
next
 case (propo-rew-one-step-lift φ φ' c ξ ξ')
 note rel = this(1) and IH = this(2) and corr = this(3) and no-T-F = this(4)
 {
   assume c: c = CNot
   then have empty: ξ = [] ξ' = [] using corr by auto
   then have no-T-F φ using no-T-F c no-T-F-decomp-not by auto
   then have no-T-F (conn c (ξ @ φ' # ξ')) using c empty no-T-F-comp-not IH by auto
 }
 moreover {
   assume c: c ∈ binary-connectives
   then obtain a b where ab: ξ @ φ # ξ' = [a, b]
     using corr list-length2-decomp wf-conn-bin-list-length by metis
   then have φ: φ = a ∨ φ = b
     by (metis append-self-conv2 wf-conn-list-decomp(4) wf-conn-unary list.discI list.sel(3)
       nth-Cons-0 tl-append2)
   have ζ: ∀ζ ∈ set (ξ @ φ # ξ'). no-T-F ζ using ab c propo-rew-one-step-lift.prems by auto

   then have φ': no-T-F φ'
     using ab IH φ corr no-T-F no-T-F-def all-subformula-st-decomp-explicit by auto
   have χ: ξ @ φ' # ξ' = [φ', b] ∨ ξ @ φ' # ξ' = [a, φ']
     by (metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps(2)
       butlast-append list.distinct(1) list.sel(3))
   then have ∀ζ∈ set (ξ @ φ' # ξ'). no-T-F ζ using ζ φ' ab by fastforce
   moreover
     have no-T-F (last (ξ @ φ' # ξ')) by (simp add: calculation)
     then have no-T-F-symb (conn c (ξ @ φ' # ξ'))
       by (metis χ φ' ζ ab all-subformula-st-test-symb-true-phi c last.simps list.distinct(1)
         list.set-intros(1) no-T-F-bin-decomp no-T-F-def)
   ultimately have no-T-F (conn c (ξ @ φ' # ξ')) using c χ by fastforce
 }
 moreover {
   fix x
   assume c = CVar x ∨ c = CF ∨ c = CT
   then have False using corr by auto
```

```
    then have no-T-F (conn c (ξ @ φ' # ξ')) by auto
  }
  ultimately show no-T-F (conn c (ξ @ φ' # ξ')) using corr wf-conn.cases by blast
qed


lemma elim-imp-inv':
  fixes φ ψ :: 'v propo
  assumes full (propo-rew-step elim-imp) φ ψ and no-T-F-except-top-level φ
  showsno-T-F-except-top-level ψ
proof −
  {
    {
      fix φ ψ :: 'v propo
      have H: elim-imp φ ψ ⟹ no-T-F-except-top-level φ ⟹ no-T-F-except-top-level ψ
        by (induct φ ψ rule: elim-imp.induct, auto)
    } note H = this
    fix φ ψ :: 'v propo
    have propo-rew-step elim-imp φ ψ ⟹ no-T-F-except-top-level φ ⟹ no-T-F-except-top-level ψ
      proof −
        assume rel: propo-rew-step elim-imp φ ψ
        and no: no-T-F-except-top-level φ
        {
          assume φ = FT ∨ φ = FF
          from rel this have False
            apply (induct rule: propo-rew-step.induct)
            by (cases rule: elim-imp.cases, auto simp: wf-conn-list(1,2))
          then have no-T-F-except-top-level ψ by blast
        }
        moreover {
          assume φ ≠ FT ∧ φ ≠ FF
          then have no-T-F φ
            by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
          then have no-T-F ψ
            using rel propo-rew-step-ElimImp-no-T-F by blast
          then have no-T-F-except-top-level ψ by (simp add: no-T-F-no-T-F-except-top-level)
        }
        ultimately show no-T-F-except-top-level ψ by metis
      qed
  }
  moreover {
     fix c :: 'v connective and ξ ξ' :: 'v propo list and ζ ζ' :: 'v propo
     assume rel: propo-rew-step elim-imp ζ ζ'
     and incl: ζ ≼ φ
     and corr: wf-conn c (ξ @ ζ # ξ')
     and no-T-F: no-T-F-symb-except-toplevel (conn c (ξ @ ζ # ξ'))
     and n: no-T-F-symb-except-toplevel ζ'
     have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
     proof
       have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
         by (simp add: corr no-T-F no-T-F-symb-except-toplevel-no-T-F-symb wf-conn-list(1,2))

       have l: ∀φ∈set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
         using corr wf-conn-no-T-F-symb-iff p by blast
       from rel incl have ζ'≠FT ∧ζ'≠FF
         apply (induction ζ ζ' rule: propo-rew-step.induct)
```

apply (*cases rule*: *elim-imp.cases*, *auto*)
using *wf-conn-list*(*1*,*2*) *wf-conn-no-arity-change wf-conn-no-arity-change-helper*
by (*metis append-is-Nil-conv list.distinct*(*1*))+
then have $\forall \varphi \in set\ (\xi\ @\ \zeta'\ \#\ \xi').\ \varphi \neq FT \wedge \varphi \neq FF$ **using** *l* **by** *auto*
moreover have $c \neq CT \wedge c \neq CF$ **using** *corr* **by** *auto*
ultimately show *no-T-F-symb* (*conn c* (ξ @ ζ' # ξ'))
using *corr wf-conn-no-arity-change no-T-F-symb-comp*
by (*metis wf-conn-no-arity-change-helper*)
qed
}
ultimately show *no-T-F-except-top-level* ψ
using *full-propo-rew-step-inv-stay-with-inc*[*of elim-imp no-T-F-symb-except-toplevel* φ]
assms subformula-refl **unfolding** *no-T-F-except-top-level-def* **by** *metis*
qed

3.7.3 The new CNF and DNF transformation

The transformation is the same as before, but the order is not the same.

definition *dnf-rew'* :: *'a propo* ⇒ *'a propo* ⇒ *bool* **where**
dnf-rew' =
(*full* (*propo-rew-step elimTBFull*)) *OO*
(*full* (*propo-rew-step elim-equiv*)) *OO*
(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushConj*))

lemma *dnf-rew'-consistent*: *preserves-un-sat dnf-rew'*
by (*simp add*: *dnf-rew'-def elimEquv-lifted-consistant elim-imp-lifted-consistant*
elimTBFull-consistent preserves-un-sat-OO pushConj-consistent pushNeg-lifted-consistant)

theorem *cnf-transformation-correction*:
dnf-rew' φ φ' ⟹ *is-dnf* φ'
unfolding *dnf-rew'-def OO-def*
by (*meson and-in-or-only-conjunction-in-disj elimTBFull-full-propo-rew-step elim-equiv-inv'*
elim-imp-inv elim-imp-inv' is-dnf-def no-equiv-full-propo-rew-step-elim-equiv
no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(*1*−*4*)
pushNeg-full-propo-rew-step pushNeg-inv(*1*−*3*))

Given all the lemmas before the CNF transformation is easy to prove:

definition *cnf-rew'* :: *'a propo* ⇒ *'a propo* ⇒ *bool* **where**
cnf-rew' =
(*full* (*propo-rew-step elimTBFull*)) *OO*
(*full* (*propo-rew-step elim-equiv*)) *OO*
(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushDisj*))

lemma *cnf-rew'-consistent*: *preserves-un-sat cnf-rew'*
by (*simp add*: *cnf-rew'-def elimEquv-lifted-consistant elim-imp-lifted-consistant*
elimTBFull-consistent preserves-un-sat-OO pushDisj-consistent pushNeg-lifted-consistant)

theorem *cnf'-transformation-correction*:
cnf-rew' φ φ' ⟹ *is-cnf* φ'
unfolding *cnf-rew'-def OO-def*
by (*meson elimTBFull-full-propo-rew-step elim-equiv-inv' elim-imp-inv elim-imp-inv' is-cnf-def*

no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp or-in-and-only-conjunction-in-disj pushDisj-full-propo-rew-step pushDisj-inv(1−4) pushNeg-full-propo-rew-step pushNeg-inv(1) pushNeg-inv(2) pushNeg-inv(3))

end
theory *Prop-Logic-Multiset*
imports *../lib/Multiset-More Prop-Normalisation Partial-Clausal-Logic*
begin

3.8 Link with Multiset Version

3.8.1 Transformation to Multiset

fun *mset-of-conj* :: $'a$ *propo* $\Rightarrow$ $'a$ *literal multiset* **where**
mset-of-conj $(FOr\ \varphi\ \psi)$ = *mset-of-conj* φ + *mset-of-conj* ψ |
mset-of-conj $(FVar\ v)$ = {# *Pos* v #} |
mset-of-conj $(FNot\ (FVar\ v))$ = {# *Neg* v #} |
mset-of-conj *FF* = {#}

fun *mset-of-formula* :: $'a$ *propo* $\Rightarrow$ $'a$ *literal multiset set* **where**
mset-of-formula $(FAnd\ \varphi\ \psi)$ = *mset-of-formula* φ $\cup$ *mset-of-formula* ψ |
mset-of-formula $(FOr\ \varphi\ \psi)$ = {*mset-of-conj* $(FOr\ \varphi\ \psi)$} |
mset-of-formula $(FVar\ \psi)$ = {*mset-of-conj* $(FVar\ \psi)$} |
mset-of-formula $(FNot\ \psi)$ = {*mset-of-conj* $(FNot\ \psi)$} |
mset-of-formula *FF* = {{#}} |
mset-of-formula *FT* = {}

3.8.2 Equisatisfiability of the two Version

lemma *is-conj-with-TF-FNot*:
  is-conj-with-TF $(FNot\ \varphi) \longleftrightarrow (\exists v.\ \varphi = FVar\ v \lor \varphi = FF \lor \varphi = FT)$
  unfolding *is-conj-with-TF-def* **apply** (*rule iffI*)
  apply (*induction FNot* φ *rule*: *super-grouped-by.induct*)
  apply (*induction FNot* φ *rule*: *grouped-by.induct*)
     apply *simp*
    apply (*cases* φ; *simp*)
  apply *auto*
  done

lemma *grouped-by-COr-FNot*:
  grouped-by COr $(FNot\ \varphi) \longleftrightarrow (\exists v.\ \varphi = FVar\ v \lor \varphi = FF \lor \varphi = FT)$
  unfolding *is-conj-with-TF-def* **apply** (*rule iffI*)
  apply (*induction FNot* φ *rule*: *grouped-by.induct*)
     apply *simp*
    apply (*cases* φ; *simp*)
  apply *auto*
  done

lemma
  shows *no-T-F-FF*[*simp*]: $\neg$*no-T-F FF* **and**
    no-T-F-FT[*simp*]: $\neg$*no-T-F FT*
  unfolding *no-T-F-def all-subformula-st-def* **by** *auto*

lemma *grouped-by-CAnd-FAnd*:
  grouped-by CAnd $(FAnd\ \varphi 1\ \varphi 2) \longleftrightarrow$ *grouped-by CAnd* $\varphi 1 \land$ *grouped-by CAnd* $\varphi 2$
  apply (*rule iffI*)

apply (*induction FAnd* $\varphi 1$ $\varphi 2$ *rule*: *grouped-by.induct*)
using *connected-is-group*[*of CAnd* $\varphi 1$ $\varphi 2$] **by** *auto*

lemma *grouped-by-COr-FOr*:
grouped-by COr (*FOr* $\varphi 1$ $\varphi 2$) $\longleftrightarrow$ *grouped-by COr* $\varphi 1$ $\wedge$ *grouped-by COr* $\varphi 2$
apply (*rule iffI*)
apply (*induction FOr* $\varphi 1$ $\varphi 2$ *rule*: *grouped-by.induct*)
using *connected-is-group*[*of COr* $\varphi 1$ $\varphi 2$] **by** *auto*

lemma *grouped-by-COr-FAnd*[*simp*]: $\neg$ *grouped-by COr* (*FAnd* $\varphi 1$ $\varphi 2$)
apply *clarify*
apply (*induction FAnd* $\varphi 1$ $\varphi 2$ *rule*: *grouped-by.induct*)
apply *auto*
done

lemma *grouped-by-COr-FEq*[*simp*]: $\neg$ *grouped-by COr* (*FEq* $\varphi 1$ $\varphi 2$)
apply *clarify*
apply (*induction FEq* $\varphi 1$ $\varphi 2$ *rule*: *grouped-by.induct*)
apply *auto*
done

lemma [*simp*]: $\neg$*grouped-by COr* (*FImp* φ ψ)
apply *clarify*
by (*induction FImp* φ ψ *rule*: *grouped-by.induct*) *simp-all*

lemma [*simp*]: $\neg$ *is-conj-with-TF* (*FImp* φ ψ)
unfolding *is-conj-with-TF-def* **apply** *clarify*
by (*induction FImp* φ ψ *rule*: *super-grouped-by.induct*) *simp-all*

lemma [*simp*]: $\neg$*grouped-by COr* (*FEq* φ ψ)
apply *clarify*
by (*induction FEq* φ ψ *rule*: *grouped-by.induct*) *simp-all*

lemma [*simp*]: $\neg$ *is-conj-with-TF* (*FEq* φ ψ)
unfolding *is-conj-with-TF-def* **apply** *clarify*
by (*induction FEq* φ ψ *rule*: *super-grouped-by.induct*) *simp-all*

lemma *is-conj-with-TF-Fand*:
is-conj-with-TF (*FAnd* $\varphi 1$ $\varphi 2$) $\Longrightarrow$ *is-conj-with-TF* $\varphi 1$ $\wedge$ *is-conj-with-TF* $\varphi 2$
unfolding *is-conj-with-TF-def*
apply (*induction FAnd* $\varphi 1$ $\varphi 2$ *rule*: *super-grouped-by.induct*)
apply (*auto simp*: *grouped-by-CAnd-FAnd intro*: *grouped-is-super-grouped*)[]
apply *auto*[]
done

lemma *is-conj-with-TF-FOr*:
is-conj-with-TF (*FOr* $\varphi 1$ $\varphi 2$) $\Longrightarrow$ *grouped-by COr* $\varphi 1$ $\wedge$ *grouped-by COr* $\varphi 2$
unfolding *is-conj-with-TF-def*
apply (*induction FOr* $\varphi 1$ $\varphi 2$ *rule*: *super-grouped-by.induct*)
apply (*auto simp*: *grouped-by-COr-FOr*)[]
apply *auto*[]
done

lemma *grouped-by-COr-mset-of-formula*:
grouped-by COr $\varphi \Longrightarrow$ *mset-of-formula* φ = (*if* φ = *FT then* {} *else* {*mset-of-conj* φ})

by (*induction* φ) (*auto simp add*: *grouped-by-COr-FNot*)

When a formula is in CNF form, then there is equisatisfiability between the multiset version and the CNF form. Remark that the definition for the entailment are slightly different: *op* $\models$ uses a function assigning *True* or *False*, while *op* $\models s$ uses a set where being in the list means entailment of a literal.

theorem
  fixes φ :: *'v propo*
  assumes *is-cnf* φ
  shows *eval A* $\varphi \longleftrightarrow$ *Partial-Clausal-Logic.true-clss* ($\{Pos\ v|v.\ A\ v\} \cup \{Neg\ v|v.\ \neg A\ v\}$)
    (*mset-of-formula* φ)
  using *assms*
proof (*induction* φ)
  case *FF*
  then show *?case* **by** *auto*
next
  case *FT*
  then show *?case* **by** *auto*
next
  case (*FVar v*)
  then show *?case* **by** *auto*
next
  case (*FAnd* φ ψ)
  then show *?case*
  unfolding *is-cnf-def* **by** (*auto simp*: *is-conj-with-TF-FNot dest*: *is-conj-with-TF-Fand*
    dest!:*is-conj-with-TF-FOr*)
next
  case (*FOr* φ ψ)
  then have [*simp*]: *mset-of-formula* φ = {*mset-of-conj* φ} *mset-of-formula* ψ = {*mset-of-conj* ψ}
    unfolding *is-cnf-def* **by** (*auto dest*!:*is-conj-with-TF-FOr simp*: *grouped-by-COr-mset-of-formula*
      split: *if-splits*)
  have *is-conj-with-TF* φ *is-conj-with-TF* ψ
    using *FOr*(*3*) **unfolding** *is-cnf-def no-T-F-def*
    by (*metis grouped-is-super-grouped is-conj-with-TF-FOr is-conj-with-TF-def*)+
  then show *?case* **using** *FOr*
    unfolding *is-cnf-def* **by** *simp*
next
  case (*FImp* φ ψ)
  then show *?case*
    unfolding *is-cnf-def* **by** *auto*
next
  case (*FEq* φ ψ)
  then show *?case*
    unfolding *is-cnf-def* **by** *auto*
next
  case (*FNot* φ)
  then show *?case*
    unfolding *is-cnf-def* **by** (*auto simp*: *is-conj-with-TF-FNot*)
qed

end
theory *Prop-Resolution*
imports *Partial-Clausal-Logic List-More Wellfounded-More*

begin

Chapter 4

Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

4.1 Resolution

4.1.1 Simplification Rules

inductive *simplify* :: *'v clauses* ⇒ *'v clauses* ⇒ *bool* **for** *N* :: *'v clause set* **where**
tautology-deletion:
  $A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\} \in N \Longrightarrow simplify\ \ N\ (N - \{A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}\})$|
condensation:
  $A + \{\#L\#\} + \{\#L\#\} \in N \Longrightarrow simplify\ N\ (N - \{A + \{\#L\#\} + \{\#L\#\}\} \cup \{A + \{\#L\#\}\})$ |
subsumption:
  $A \in N \Longrightarrow A \subset\# B \Longrightarrow B \in N \Longrightarrow simplify\ N\ (N - \{B\})$

lemma *simplify-preserves-un-sat'*:
  fixes $N\ N'$ :: *'v clauses*
  assumes *simplify* $N\ N'$
  and *total-over-m* $I\ N$
  shows $I \models s\ N' \longrightarrow I \models s\ N$
  using *assms*
proof (*induct rule*: *simplify.induct*)
  case (*tautology-deletion* $A\ P$)
  then have $I \models A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}$
    by (*metis total-over-m-def total-over-set-literal-defined true-cls-singleton true-cls-union true-lit-def uminus-Neg union-commute*)
  then show *?case* **by** (*metis Un-Diff-cancel2 true-clss-singleton true-clss-union*)
next
  case (*condensation* $A\ P$)
  then show *?case* **by** (*metis Diff-insert-absorb Set.set-insert insertE true-cls-union true-clss-def true-clss-singleton true-clss-union*)
next
  case (*subsumption* $A\ B$)
  have $A \neq B$ **using** *subsumption.hyps*(*2*) **by** *auto*
  then have $I \models s\ N - \{B\} \Longrightarrow I \models A$ **using** ⟨$A \in N$⟩ **by** (*simp add*: *true-clss-def*)
  moreover have $I \models A \Longrightarrow I \models B$ **using** ⟨$A <\# B$⟩ **by** *auto*
  ultimately show *?case* **by** (*metis insert-Diff-single true-clss-insert*)
qed

lemma *simplify-preserves-un-sat*:
  fixes $N\ N'$ :: *'v clauses*

assumes *simplify N N'*
and *total-over-m I N*
shows $I \models s\ N \longrightarrow I \models s\ N'$
using *assms* **apply** (*induct rule*: *simplify.induct*)
using *true-clss-def* **by** *fastforce+*

lemma *simplify-preserves-un-sat''*:
fixes *N N'* :: *'v clauses*
assumes *simplify N N'*
and *total-over-m I N'*
shows $I \models s\ N \longrightarrow I \models s\ N'$
using *assms* **apply** (*induct rule*: *simplify.induct*)
using *true-clss-def* **by** *fastforce+*

lemma *simplify-preserves-un-sat-eq*:
fixes *N N'* :: *'v clauses*
assumes *simplify N N'*
and *total-over-m I N*
shows $I \models s\ N \longleftrightarrow I \models s\ N'$
using *simplify-preserves-un-sat simplify-preserves-un-sat' assms* **by** *blast*

lemma *simplify-preserves-finite*:
assumes *simplify* ψ ψ'
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
using *assms* **by** (*induct rule*: *simplify.induct*, *auto simp add*: *remove-def*)

lemma *rtranclp-simplify-preserves-finite*:
assumes *rtranclp simplify* ψ ψ'
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*auto simp add*: *simplify-preserves-finite*)

lemma *simplify-atms-of-ms*:
assumes *simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
using *assms* **unfolding** *atms-of-ms-def*
proof (*induct rule*: *simplify.induct*)
case (*tautology-deletion A P*)
then show *?case* **by** *auto*
next
case (*condensation A P*)
moreover have $A + \{\#P\#\} + \{\#P\#\} \in \psi \Longrightarrow \exists x{\in}\psi.$ *atm-of* $P \in$ *atm-of* ` *set-mset x*
by (*metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff*)
ultimately show *?case* **by** (*auto simp add*: *atms-of-def*)
next
case (*subsumption A P*)
then show *?case* **by** *auto*
qed

lemma *rtranclp-simplify-atms-of-ms*:
assumes *rtranclp simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply (*fastforce intro*: *simplify-atms-of-ms*)
using *simplify-atms-of-ms* **by** *blast*

lemma *factoring-imp-simplify*:

assumes $\{\#L\#\} + \{\#L\#\} + C \in N$
shows $\exists N'.$ *simplify* N N'
proof $-$
have $C + \{\#L\#\} + \{\#L\#\} \in N$ **using** *assms* **by** (*simp add*: *add.commute union-lcomm*)
from *condensation*[*OF this*] **show** *?thesis* **by** *blast*
qed

4.1.2 Unconstrained Resolution

type-synonym *'v uncon-state* = *'v clauses*
inductive *uncon-res* :: *'v uncon-state* $\Rightarrow$ *'v uncon-state* $\Rightarrow$ *bool* **where**
resolution:
$\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$ *already-used*
$\Longrightarrow$ *uncon-res* (N) $(N \cup \{C + D\})$ |
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$ *uncon-res* N $(N \cup \{C + \{\#L\#\}\})$

lemma *uncon-res-increasing*:
assumes *uncon-res* S S' **and** $\psi \in S$
shows $\psi \in S'$
using *assms* **by** (*induct rule*: *uncon-res.induct*) *auto*

lemma *rtranclp-uncon-inference-increasing*:
assumes *rtranclp uncon-res* S S' **and** $\psi \in S$
shows $\psi \in S'$
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*auto simp add*: *uncon-res-increasing*)

Subsumption

definition *subsumes* :: *'a literal multiset* $\Rightarrow$ *'a literal multiset* $\Rightarrow$ *bool* **where**
subsumes χ χ' $\longleftrightarrow$
$(\forall I.$ *total-over-m* I $\{\chi'\} \longrightarrow$ *total-over-m* I $\{\chi\})$
$\wedge$ $(\forall I.$ *total-over-m* I $\{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi')$

lemma *subsumes-refl*[*simp*]:
subsumes χ χ
unfolding *subsumes-def* **by** *auto*

lemma *subsumes-subsumption*:
assumes *subsumes* D χ
and $C \subset\# D$ **and** $\neg$*tautology* χ
shows *subsumes* C χ **unfolding** *subsumes-def*
using *assms subsumption-total-over-m subsumption-chained* **unfolding** *subsumes-def*
by (*blast intro*!: *subset-mset.less-imp-le*)

lemma *subsumes-tautology*:
assumes *subsumes* $(C + \{\#Pos\ P\#\} + \{\#Neg\ P\#\})$ χ
shows *tautology* χ
using *assms* **unfolding** *subsumes-def* **by** (*simp add*: *tautology-def*)

4.1.3 Inference Rule

type-synonym *'v state* = *'v clauses* $\times$ (*'v clause* $\times$ *'v clause*) *set*
inductive *inference-clause* :: *'v state* $\Rightarrow$ *'v clause* $\times$ (*'v clause* $\times$ *'v clause*) *set* $\Rightarrow$ *bool*
(**infix** $\Rightarrow_{\mathrm{Res}}$ *100*) **where**

resolution:
  $\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$ *already-used*
  $\Longrightarrow$ *inference-clause* (N, *already-used*) ($C + D$, *already-used* $\cup$ $\{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\}$) |
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$ *inference-clause* (N, *already-used*) ($C + \{\#L\#\}$, *already-used*)

inductive *inference* :: $'v$ *state* $\Rightarrow$ $'v$ *state* $\Rightarrow$ *bool* **where**
inference-step: *inference-clause* S (*clause*, *already-used*)
  $\Longrightarrow$ *inference* S (*fst* $S \cup \{$*clause*$\}$, *already-used*)

abbreviation *already-used-inv*
  :: $'a$ *literal multiset set* $\times$ ($'a$ *literal multiset* $\times$ $'a$ *literal multiset*) *set* $\Rightarrow$ *bool* **where**
already-used-inv state $\equiv$
  $(\forall (A, B) \in$ *snd state*. $\exists p.$ *Pos* $p \in\# A \wedge$ *Neg* $p \in\# B \wedge$
      $((\exists \chi \in$ *fst state*. *subsumes* χ $((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$
       $\vee$ *tautology* $((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

lemma *inference-clause-preserves-already-used-inv*:
  assumes *inference-clause* S S'
  and *already-used-inv* S
  shows *already-used-inv* (*fst* $S \cup \{$*fst* $S'\}$, *snd* S')
  using *assms* **apply** (*induct rule*: *inference-clause.induct*)
  by *fastforce+*

lemma *inference-preserves-already-used-inv*:
  assumes *inference* S S'
  and *already-used-inv* S
  shows *already-used-inv* S'
  using *assms*
proof (*induct rule*: *inference.induct*)
  case (*inference-step* S *clause already-used*)
  then show *?case*
    using *inference-clause-preserves-already-used-inv*[*of* S (*clause*, *already-used*)] **by** *simp*
qed

lemma *rtranclp-inference-preserves-already-used-inv*:
  assumes *rtranclp inference* S S'
  and *already-used-inv* S
  shows *already-used-inv* S'
  using *assms* **apply** (*induct rule*: *rtranclp-induct*, *simp*)
  using *inference-preserves-already-used-inv* **unfolding** *tautology-def* **by** *fast*

lemma *subsumes-condensation*:
  assumes *subsumes* ($C + \{\#L\#\} + \{\#L\#\}$) D
  shows *subsumes* ($C + \{\#L\#\}$) D
  using *assms* **unfolding** *subsumes-def* **by** *simp*

lemma *simplify-preserves-already-used-inv*:
  assumes *simplify* N N'
  and *already-used-inv* (N, *already-used*)
  shows *already-used-inv* (N', *already-used*)
  using *assms*
proof (*induct rule*: *simplify.induct*)
  case (*condensation* C L)

```
  then show ?case
    using subsumes-condensation by simp fast
next
  {
    fix a:: 'a and A :: 'a set and P
    have (∃ x ∈ Set.remove a A. P x) ⟷ (∃ x ∈ A. x ≠ a ∧ P x) by auto
  } note ex-member-remove = this
  {
    fix a a0 :: 'v clause and A :: 'v clauses and y
    assume a ∈ A and a0 ⊂# a
    then have (∃ x ∈ A. subsumes x y) ⟷ (subsumes a y ∨ (∃ x ∈ A. x ≠ a ∧ subsumes x y))
      by auto
  } note tt2 = this
  case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and inv = this(4)
  show ?case
    proof (standard, standard)
      fix x a b
      assume x: x ∈ snd (N − {B}, already-used) and [simp]: x = (a, b)
      obtain p where p: Pos p ∈# a ∧ Neg p ∈# b and
        q: (∃ χ∈N. subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
          ∨ tautology (a − {#Pos p#} + (b − {#Neg p#}))
        using inv x by fastforce
      consider (taut) tautology (a − {#Pos p#} + (b − {#Neg p#})) |
        (χ) χ where χ ∈ N subsumes χ (a − {#Pos p#} + (b − {#Neg p#}))
          ¬tautology (a − {#Pos p#} + (b − {#Neg p#}))
        using q by auto
      then show
        ∃ p. Pos p ∈# a ∧ Neg p ∈# b
            ∧ ((∃ χ∈fst (N − {B}, already-used). subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
                ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
        proof cases
          case taut
          then show ?thesis using p by auto
        next
          case χ note H = this
          show ?thesis using p A AB B subsumes-subsumption[OF - AB H(3)] H(1,2) by auto
        qed
    qed
next
  case (tautology-deletion C P)
  then show ?case apply clarify
  proof −
    fix a b
    assume C + {#Pos P#} + {#Neg P#} ∈ N
    assume already-used-inv (N, already-used)
    and (a, b) ∈ snd (N − {C + {#Pos P#} + {#Neg P#}}, already-used)
    then obtain p where
      Pos p ∈# a ∧ Neg p ∈# b ∧
        ((∃ χ∈fst (N ∪ {C + {#Pos P#} + {#Neg P#}}, already-used).
              subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
          ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
      by fastforce
    moreover have tautology (C + {#Pos P#} + {#Neg P#}) by auto
    ultimately show
      ∃ p. Pos p ∈# a ∧ Neg p ∈# b
      ∧ ((∃ χ∈fst (N − {C + {#Pos P#} + {#Neg P#}}, already-used).
```

subsumes χ $(a - \{\#Pos\ p\#\} + (b - \{\#Neg\ p\#\})))$
$\vee$ *tautology* $(a - \{\#Pos\ p\#\} + (b - \{\#Neg\ p\#\})))$
by (*metis* (*no-types*) *Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology sup-bot.right-neutral*)
qed
qed

lemma
factoring-satisfiable: $I \models \{\#L\#\} + \{\#L\#\} + C \longleftrightarrow I \models \{\#L\#\} + C$ **and**
resolution-satisfiable:
consistent-interp $I \Longrightarrow I \models \{\#Pos\ p\#\} + C \Longrightarrow I \models \{\#Neg\ p\#\} + D \Longrightarrow I \models C + D$ **and**
factoring-same-vars: *atms-of* $(\{\#L\#\} + \{\#L\#\} + C)$ = *atms-of* $(\{\#L\#\} + C)$
unfolding *true-cls-def consistent-interp-def* **by** (*fastforce split*: *if-split-asm*)+

lemma *inference-increasing*:
assumes *inference* $S\ S'$ **and** $\psi \in$ *fst* S
shows $\psi \in$ *fst* S'
using *assms* **by** (*induct rule*: *inference.induct*, *auto*)

lemma *rtranclp-inference-increasing*:
assumes *rtranclp inference* $S\ S'$ **and** $\psi \in$ *fst* S
shows $\psi \in$ *fst* S'
using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *inference-increasing*)

lemma *inference-clause-already-used-increasing*:
assumes *inference-clause* $S\ S'$
shows *snd* $S \subseteq$ *snd* S'
using *assms* **by** (*induct rule*:*inference-clause.induct*, *auto*)

lemma *inference-already-used-increasing*:
assumes *inference* $S\ S'$
shows *snd* $S \subseteq$ *snd* S'
using *assms* **apply** (*induct rule*:*inference.induct*)
using *inference-clause-already-used-increasing* **by** *fastforce*

lemma *inference-clause-preserves-un-sat*:
fixes $N\ N'$ :: *'v clauses*
assumes *inference-clause* $T\ T'$
and *total-over-m* I (*fst* T)
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $T \longleftrightarrow I \models s$ *fst* $T \cup \{$*fst* $T'\}$
using *assms* **apply** (*induct rule*: *inference-clause.induct*)
unfolding *consistent-interp-def true-clss-def* **by** *auto force*+

lemma *inference-preserves-un-sat*:
fixes $N\ N'$ :: *'v clauses*
assumes *inference* $T\ T'$
and *total-over-m* I (*fst* T)
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $T \longleftrightarrow I \models s$ *fst* T'
using *assms* **apply** (*induct rule*: *inference.induct*)
using *inference-clause-preserves-un-sat* **by** *fastforce*

lemma *inference-clause-preserves-atms-of-ms*:
assumes *inference-clause S S'*
shows *atms-of-ms* (*fst* (*fst S* ∪ {*fst S'*}, *snd S'*)) ⊆ *atms-of-ms* (*fst S*)
using *assms* **apply** (*induct rule*: *inference-clause.induct*)
apply *auto*
apply (*metis Set.set-insert UnCI atms-of-ms-insert atms-of-plus*)
apply (*metis Set.set-insert UnCI atms-of-ms-insert atms-of-plus*)
apply (*simp add*: *in-m-in-literals union-assoc*)
unfolding *atms-of-ms-def* **using** *assms* **by** *fastforce*

lemma *inference-preserves-atms-of-ms*:
fixes *N N'* :: *'v clauses*
assumes *inference T T'*
shows *atms-of-ms* (*fst T'*) ⊆ *atms-of-ms* (*fst T*)
using *assms* **apply** (*induct rule*: *inference.induct*)
using *inference-clause-preserves-atms-of-ms* **by** *fastforce*

lemma *inference-preserves-total*:
fixes *N N'* :: *'v clauses*
assumes *inference* (*N*, *already-used*) (*N'*, *already-used'*)
shows *total-over-m I N* ⟹ *total-over-m I N'*
using *assms inference-preserves-atms-of-ms* **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference T T'*
shows *total-over-m I* (*fst T*) ⟹ *total-over-m I* (*fst T'*)
using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *inference-preserves-total*)

lemma *rtranclp-inference-preserves-un-sat*:
assumes *rtranclp inference N N'*
and *total-over-m I* (*fst N*)
and *consistent*: *consistent-interp I*
shows *I* ⊨*s fst N* ⟷ *I* ⊨*s fst N'*
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply (*simp add*: *inference-preserves-un-sat*)
using *inference-preserves-un-sat rtranclp-inference-preserves-total* **by** *blast*

lemma *inference-preserves-finite*:
assumes *inference ψ ψ'* **and** *finite* (*fst ψ*)
shows *finite* (*fst ψ'*)
using *assms* **by** (*induct rule*: *inference.induct*, *auto simp add*: *simplify-preserves-finite*)

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause ψ ψ'* **and** *finite* (*snd ψ*)
shows *finite* (*snd ψ'*)
using *assms* **by** (*induct rule*: *inference-clause.induct*, *auto*)

lemma *inference-preserves-finite-snd*:
assumes *inference ψ ψ'* **and** *finite* (*snd ψ*)
shows *finite* (*snd ψ'*)
using *assms inference-clause-preserves-finite-snd* **by** (*induct rule*: *inference.induct*, *fastforce*)

lemma *rtranclp-inference-preserves-finite*:
  assumes *rtranclp inference* ψ ψ' **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  using *assms* **by** (*induct rule*: *rtranclp-induct*)
    (*auto simp add*: *simplify-preserves-finite inference-preserves-finite*)

lemma *consistent-interp-insert*:
  assumes *consistent-interp I*
  and *atm-of P* $\notin$ *atm-of* ` *I*
  shows *consistent-interp* (*insert P I*)
proof −
  have *P*: *insert P I* = *I* $\cup$ {*P*} **by** *auto*
  show *?thesis* **unfolding** *P*
  apply (*rule consistent-interp-disjoint*)
  using *assms* **by** (*auto simp*: *image-iff*)
qed

lemma *simplify-clause-preserves-sat*:
  assumes *simp*: *simplify* ψ ψ'
  and *satisfiable* ψ'
  shows *satisfiable* ψ
  using *assms*
proof *induction*
  case (*tautology-deletion A P*) **note** *AP* = *this*(*1*) **and** *sat* = *this*(*2*)
  let *?A'* = *A* + {#*Pos P*#} + {#*Neg P*#}
  let *?*ψ' = ψ − {*?A'*}
  obtain *I* **where**
    I: *I* $\models$*s ?*ψ' **and**
    cons: *consistent-interp I* **and**
    tot: *total-over-m I ?*ψ'
    using *sat* **unfolding** *satisfiable-def* **by** *auto*
  { **assume** *Pos P* $\in$ *I* $\vee$ *Neg P* $\in$ *I*
    then have *I* $\models$ *?A'* **by** *auto*
    then have *I* $\models$*s* ψ **using** *I* **by** (*metis insert-Diff tautology-deletion.hyps true-clss-insert*)
    then have *?case* **using** *cons tot* **by** *auto*
  }
  moreover {
    assume *Pos*: *Pos P* $\notin$ *I* **and** *Neg*: *Neg P* $\notin$ *I*
    then have *consistent-interp* (*I* $\cup$ {*Pos P*}) **using** *cons* **by** *simp*
    moreover have *I'A*: *I* $\cup$ {*Pos P*} $\models$ *?A'* **by** *auto*
    have {*Pos P*} $\cup$ *I* $\models$*s* ψ − {*A* + {#*Pos P*#} + {#*Neg P*#}}
      using ‹*I* $\models$*s* ψ − {*A* + {#*Pos P*#} + {#*Neg P*#}}› *true-clss-union-increase'* **by** *blast*
    then have *I* $\cup$ {*Pos P*} $\models$*s* ψ
      by (*metis* (*no-types*) *Un-empty-right Un-insert-left Un-insert-right I'A insert-Diff*
        sup-bot.left-neutral tautology-deletion.hyps true-clss-insert)
    ultimately have *?case* **using** *satisfiable-carac'* **by** *blast*
  }
  ultimately show *?case* **by** *blast*
next
  case (*condensation A L*) **note** *AL* = *this*(*1*) **and** *sat* = *this*(*2*)
  have *f3*: *simplify* ψ (ψ − {*A* + {#*L*#} + {#*L*#}} $\cup$ {*A* + {#*L*#}})
    using *AL simplify.condensation* **by** *blast*
  obtain *LL* :: *'a literal multiset set* $\Rightarrow$ *'a literal set* **where**
    f4: *LL* (ψ − {*A* + {#*L*#} + {#*L*#}} $\cup$ {*A* + {#*L*#}}) $\models$*s* ψ − {*A* + {#*L*#} + {#*L*#}} $\cup$ {*A*
+ {#*L*#}}

$\wedge$ *consistent-interp* (*LL* (ψ − {*A* + {#*L*#} + {#*L*#}} ∪ {*A* + {#*L*#}}))
$\wedge$ *total-over-m* (*LL* (ψ − {*A* + {#*L*#} + {#*L*#}}
∪ {*A* + {#*L*#}})) (ψ − {*A* + {#*L*#} + {#*L*#}} ∪ {*A* + {#*L*#}})
using *sat* **by** (*meson satisfiable-def*)
have *f5*: *insert* (*A* + {#*L*#} + {#*L*#}) (ψ − {*A* + {#*L*#} + {#*L*#}}) = ψ
using *AL* **by** *fastforce*
have *atms-of* (*A* + {#*L*#} + {#*L*#}) = *atms-of* ({#*L*#} + *A*)
by *simp*
then show *?case*
using *f5 f4 f3* **by** (*metis* (*no-types*) *add.commute satisfiable-def simplify-preserves-un-sat'*
total-over-m-insert total-over-m-union)
next
case (*subsumption A B*) **note** *A* = *this*(*1*) **and** *AB* = *this*(*2*) **and** *B* = *this*(*3*) **and** *sat* = *this*(*4*)
let *?*ψ' = ψ − {*B*}
obtain *I* **where** *I*: *I* $\models$*s ?*ψ' **and** *cons*: *consistent-interp I* **and** *tot*: *total-over-m I ?*ψ'
using *sat* **unfolding** *satisfiable-def* **by** *auto*
have *I* $\models$ *A* **using** *A I* **by** (*metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def*)
then have *I* $\models$ *B* **using** *AB subset-mset.less-imp-le true-cls-mono-leD* **by** *blast*
then have *I* $\models$*s* ψ **using** *I* **by** (*metis insert-Diff-single true-clss-insert*)
then show *?case* **using** *cons satisfiable-carac'* **by** *blast*
qed

lemma *simplify-preserves-unsat*:
assumes *inference* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
using *assms* **apply** (*induct rule*: *inference.induct*)
using *satisfiable-decreasing* **by** (*metis fst-conv*)+

lemma *inference-preserves-unsat*:
assumes *inference*** *S S'*
shows *satisfiable* (*fst S'*) $\longrightarrow$ *satisfiable* (*fst S*)
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply *simp-all*
using *simplify-preserves-unsat* **by** *blast*

datatype *'v sem-tree* = *Node 'v 'v sem-tree 'v sem-tree* | *Leaf*

fun *sem-tree-size* :: *'v sem-tree* ⇒ *nat* **where**
sem-tree-size Leaf = *0* |
sem-tree-size (*Node - ag ad*) = *1* + *sem-tree-size ag* + *sem-tree-size ad*

lemma *sem-tree-size*[*case-names bigger*]:
($\bigwedge$*xs*:: *'v sem-tree*. ($\bigwedge$*ys*:: *'v sem-tree*. *sem-tree-size ys* < *sem-tree-size xs* $\Longrightarrow$ *P ys*) $\Longrightarrow$ *P xs*)
$\Longrightarrow$ *P xs*
by (*fact Nat.measure-induct-rule*)

fun *partial-interps* :: *'v sem-tree* ⇒ *'v interp* ⇒ *'v clauses* ⇒ *bool* **where**
partial-interps Leaf I ψ = ($\exists\chi$. ¬ *I* $\models$ χ $\wedge$ χ ∈ ψ $\wedge$ *total-over-m I* {χ}) |
partial-interps (*Node v ag ad*) *I* ψ $\longleftrightarrow$
(*partial-interps ag* (*I* ∪ {*Pos v*}) ψ $\wedge$ *partial-interps ad* (*I*∪ {*Neg v*}) ψ)

lemma *simplify-preserve-partial-leaf*:
simplify N N' $\Longrightarrow$ *partial-interps Leaf I N* $\Longrightarrow$ *partial-interps Leaf I N'*
apply (*induct rule*: *simplify.induct*)

using *union-lcomm* **apply** *auto*[*1*]
apply (*simp*, *metis atms-of-plus total-over-set-union true-cls-union*)
apply *simp*
by (*metis atms-of-ms-singleton mset-le-exists-conv subset-mset-def true-cls-mono-leD total-over-m-def total-over-m-sum*)

lemma *simplify-preserve-partial-tree*:
assumes *simplify N N′*
and *partial-interps t I N*
shows *partial-interps t I N′*
using *assms* **apply** (*induct t arbitrary*: *I*, *simp*)
using *simplify-preserve-partial-leaf* **by** *metis*

lemma *inference-preserve-partial-tree*:
assumes *inference S S′*
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst S′*)
using *assms* **apply** (*induct t arbitrary*: *I*, *simp-all*)
by (*meson inference-increasing*)

lemma *rtranclp-inference-preserve-partial-tree*:
assumes *rtranclp inference N N′*
and *partial-interps t I* (*fst N*)
shows *partial-interps t I* (*fst N′*)
using *assms* **apply** (*induct rule*: *rtranclp-induct*, *auto*)
using *inference-preserve-partial-tree* **by** *force*

function *build-sem-tree* :: *′v* :: *linorder set* ⇒ *′v clauses* ⇒ *′v sem-tree* **where**
build-sem-tree atms ψ =
(*if atms* = {} ∨ ¬ *finite atms*
then Leaf
else Node (*Min atms*) (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) *ψ*)
(*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) *ψ*))
by *auto*
termination
apply (*relation measure* (λ(*A*, -). *card A*), *simp-all*)
apply (*metis Min-in card-Diff1-less remove-def*)+
done
declare *build-sem-tree.induct*[*case-names tree*]

lemma *unsatisfiable-empty*[*simp*]:
¬*unsatisfiable* {}
unfolding *satisfiable-def* **apply** *auto*
using *consistent-interp-def* **unfolding** *total-over-m-def total-over-set-def atms-of-ms-def* **by** *blast*

lemma *partial-interps-build-sem-tree-atms-general*:
fixes *ψ* :: *′v* :: *linorder clauses* **and** *p* :: *′v literal list*
assumes *unsat*: *unsatisfiable ψ* **and** *finite ψ* **and** *consistent-interp I*
and *finite atms*
and *atms-of-ms ψ* = *atms* ∪ *atms-of-s I* **and** *atms* ∩ *atms-of-s I* = {}
shows *partial-interps* (*build-sem-tree atms ψ*) *I ψ*
using *assms*

proof (*induct arbitrary*: *I rule*: *build-sem-tree.induct*)
case (*1 atms ψ Ia*) **note** *IH1* = *this*(*1*) **and** *IH2* = *this*(*2*) **and** *unsat* = *this*(*3*) **and** *finite* = *this*(*4*)
and *cons* = *this*(*5*) **and** *f* = *this*(*6*) **and** *un* = *this*(*7*) **and** *disj* = *this*(*8*)
{
assume *atms*: *atms* = {}
then have *atmsIa*: *atms-of-ms ψ* = *atms-of-s Ia* **using** *un* **by** *auto*
then have *total-over-m Ia ψ* **unfolding** *total-over-m-def atmsIa* **by** *auto*
then have χ: ∃χ ∈ ψ. ¬ *Ia* ⊨ χ
using *unsat cons* **unfolding** *true-clss-def satisfiable-def* **by** *auto*
then have *build-sem-tree atms ψ* = *Leaf* **using** *atms* **by** *auto*
moreover
have *tot*: ⋀χ. χ ∈ ψ ⟹ *total-over-m Ia* {χ}
unfolding *total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def*
using *atmsIa atms-of-ms-def* **by** *fastforce*
have *partial-interps Leaf Ia ψ*
using χ *tot* **by** (*auto simp add*: *total-over-m-def total-over-set-def atms-of-ms-def*)

ultimately have *?case* **by** *metis*
}
moreover {
assume *atms*: *atms* ≠ {}
have *build-sem-tree atms ψ* = *Node* (*Min atms*) (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
using *build-sem-tree.simps*[*of atms ψ*] *f atms* **by** *metis*

have *consistent-interp* (*Ia* ∪ {*Pos* (*Min atms*)}) **unfolding** *consistent-interp-def*
by (*metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff*
f in-atms-of-s-decomp insert-iff literal.distinct(*1*) *literal.exhaust-sel literal.sel*(*2*)
uminus-Neg uminus-Pos)
moreover have *atms-of-ms ψ* = *Set.remove* (*Min atms*) *atms* ∪ *atms-of-s* (*Ia* ∪ {*Pos* (*Min atms*)})
using *Min-in atms f un* **by** *fastforce*
moreover have *disj'*: *Set.remove* (*Min atms*) *atms* ∩ *atms-of-s* (*Ia* ∪ {*Pos* (*Min atms*)}) = {}
by *simp* (*metis disj disjoint-iff-not-equal member-remove*)
moreover have *finite* (*Set.remove* (*Min atms*) *atms*) **using** *f* **by** (*simp add*: *remove-def*)
ultimately have *subtree1*: *partial-interps* (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*Ia* ∪ {*Pos* (*Min atms*)}) ψ
using *IH1*[*of Ia* ∪ {*Pos* (*Min* (*atms*))}] *atms f unsat finite* **by** *metis*

have *consistent-interp* (*Ia* ∪ {*Neg* (*Min atms*)}) **unfolding** *consistent-interp-def*
by (*metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff*
f in-atms-of-s-decomp insert-iff literal.distinct(*1*) *literal.exhaust-sel literal.sel*(*2*)
uminus-Neg)
moreover have *atms-of-ms ψ* = *Set.remove* (*Min atms*) *atms* ∪ *atms-of-s* (*Ia* ∪ {*Neg* (*Min atms*)})
using ‹*atms-of-ms ψ* = *Set.remove* (*Min atms*) *atms* ∪ *atms-of-s* (*Ia* ∪ {*Pos* (*Min atms*)})› **by**
blast

moreover have *disj'*: *Set.remove* (*Min atms*) *atms* ∩ *atms-of-s* (*Ia* ∪ {*Neg* (*Min atms*)}) = {}
using *disj* **by** *auto*
moreover have *finite* (*Set.remove* (*Min atms*) *atms*) **using** *f* **by** (*simp add*: *remove-def*)
ultimately have *subtree2*: *partial-interps* (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*Ia* ∪ {*Neg* (*Min atms*)}) ψ
using *IH2*[*of Ia* ∪ {*Neg* (*Min* (*atms*))}] *atms f unsat finite* **by** *metis*

then have *?case*
using *IH1 subtree1 subtree2 f local.finite unsat atms* **by** *simp*
}

ultimately show *?case* **by** *metis*
qed

lemma *partial-interps-build-sem-tree-atms*:
fixes ψ :: *'v* :: *linorder clauses* **and** *p* :: *'v literal list*
assumes *unsat*: *unsatisfiable* ψ **and** *finite*: *finite* ψ
shows *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
proof −
have *consistent-interp* {} **unfolding** *consistent-interp-def* **by** *auto*
moreover have *atms-of-ms* ψ = *atms-of-ms* ψ $\cup$ *atms-of-s* {} **unfolding** *atms-of-s-def* **by** *auto*
moreover have *atms-of-ms* ψ $\cap$ *atms-of-s* {} = {} **unfolding** *atms-of-s-def* **by** *auto*
moreover have *finite* (*atms-of-ms* ψ) **unfolding** *atms-of-ms-def* **using** *finite* **by** *simp*
ultimately show *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
using *partial-interps-build-sem-tree-atms-general*[*of* ψ {} *atms-of-ms* ψ] *assms* **by** *metis*
qed

lemma *can-decrease-count*:
fixes ψ'' :: *'v clauses* $\times$ (*'v clause* $\times$ *'v clause* $\times$ *'v*) *set*
assumes *count* χ $L = n$
and $L \in\# \chi$ **and** $\chi \in$ *fst* ψ
shows $\exists \psi'\ \chi'$. *inference*** $\psi\ \psi' \wedge \chi' \in$ *fst* $\psi' \wedge (\forall L.\ L \in\# \chi \longleftrightarrow L \in\# \chi')$
$\wedge$ *count* χ' $L = 1$
$\wedge$ $(\forall \varphi.\ \varphi \in$ *fst* $\psi \longrightarrow \varphi \in$ *fst* $\psi')$
$\wedge$ $(I \models \chi \longleftrightarrow I \models \chi')$
$\wedge$ $(\forall I'.$ *total-over-m* I' $\{\chi\} \longrightarrow$ *total-over-m* I' $\{\chi'\})$
using *assms*
proof (*induct n arbitrary*: χ ψ)
case *0*
then show *?case* **by** (*simp add*: *not-in-iff*[*symmetric*])
next
case (*Suc n* χ)
note *IH* = *this*(*1*) **and** *count* = *this*(*2*) **and** *L* = *this*(*3*) **and** χ = *this*(*4*)
{
assume $n = 0$
then have *inference*** ψ ψ
and $\chi \in$ *fst* ψ
and $\forall L.\ (L \in\# \chi) \longleftrightarrow (L \in\# \chi)$
and *count* χ $L = (1$::*nat*)
and $\forall \varphi.\ \varphi \in$ *fst* $\psi \longrightarrow \varphi \in$ *fst* ψ
by (*auto simp add*: *count L* χ)
then have *?case* **by** *metis*
}
moreover {
assume $n > 0$
then have $\exists C.\ \chi = C + \{\#L, L\#\}$
by (*smt L Suc-eq-plus1-left add.left-commute add-diff-cancel-left' add-diff-cancel-right'*
count-greater-zero-iff count-single local.count multi-member-split plus-multiset.rep-eq)
then obtain *C* **where** *C*: $\chi = C + \{\#L, L\#\}$ **by** *metis*
let $?\chi' = C + \{\#L\#\}$
let $?\psi' = ($*fst* $\psi \cup \{?\chi'\}$, *snd* $\psi)$
have φ: $\forall \varphi \in$ *fst* ψ. $(\varphi \in$ *fst* $\psi \vee \varphi \neq ?\chi') \longleftrightarrow \varphi \in$ *fst* $?\psi'$ **unfolding** *C* **by** *auto*
have *inf*: *inference* ψ $?\psi'$
using *C factoring* χ *prod.collapse union-commute inference-step* **by** *metis*
moreover have *count'*: *count* $?\chi'$ $L = n$ **using** *C count* **by** *auto*
moreover have $L\chi'$: $L \in\# ?\chi'$ **by** *auto*

moreover have $\chi'\psi'$: *?*$\chi' \in$ *fst ?*ψ' **by** *auto*
ultimately obtain ψ'' **and** χ''
where
inference** *?*ψ' ψ'' **and**
α: $\chi'' \in$ *fst* ψ'' **and**
$\forall$ *La*. (*La* $\in\#$ *?*χ') $\longleftrightarrow$ (*La* $\in\#$ χ'') **and**
β: *count* χ'' *L* = (*1*::*nat*) **and**
φ': $\forall \varphi$. $\varphi \in$ *fst ?*ψ' $\longrightarrow$ $\varphi \in$ *fst* ψ'' **and**
$I\chi$: $I \models$ *?*χ' $\longleftrightarrow$ $I \models \chi''$ **and**
tot: $\forall I'$. *total-over-m* I' {*?*χ'} $\longrightarrow$ *total-over-m* I' {χ''}
using *IH*[*of ?*χ' *?*ψ'] *count'* $L\chi'$ $\chi'\psi'$ **by** *blast*

then have *inference*** ψ ψ''
and $\forall$ *La*. (*La* $\in\#$ χ) $\longleftrightarrow$ (*La* $\in\#$ χ'')
using *inf* **unfolding** *C* **by** *auto*
moreover have $\forall \varphi$. $\varphi \in$ *fst* ψ $\longrightarrow$ $\varphi \in$ *fst* ψ'' **using** φ φ' **by** *metis*
moreover have $I \models \chi \longleftrightarrow I \models \chi''$ **using** $I\chi$ **unfolding** *true-cls-def C* **by** *auto*
moreover have $\forall I'$. *total-over-m* I' {χ} $\longrightarrow$ *total-over-m* I' {χ''}
using *tot* **unfolding** *C total-over-m-def* **by** *auto*
ultimately have *?case* **using** φ φ' α β **by** *metis*
}
ultimately show *?case* **by** *auto*
qed

lemma *can-decrease-tree-size*:
fixes ψ :: *'v state* **and** *tree* :: *'v sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
shows $\exists$ (*tree'*:: *'v sem-tree*) ψ'. *inference*** ψ ψ' $\wedge$ *partial-interps tree' I* (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* < *sem-tree-size tree* $\vee$ *sem-tree-size tree* = *0*)
using *assms*
proof (*induct arbitrary*: *I rule*: *sem-tree-size*)
case (*bigger xs I*) **note** *IH* = *this*(*1*) **and** *finite* = *this*(*2*) **and** *a-u-i* = *this*(*3*) **and** *part* = *this*(*4*)

{
assume *sem-tree-size xs* = *0*
then have *?case* **using** *part* **by** *blast*
}

moreover {
assume *sn0*: *sem-tree-size xs* > *0*
obtain *ag ad v* **where** *xs*: *xs* = *Node v ag ad* **using** *sn0* **by** (*cases xs*, *auto*)
{
assume *sem-tree-size ag* = *0* **and** *sem-tree-size ad* = *0*
then have *ag*: *ag* = *Leaf* **and** *ad*: *ad* = *Leaf* **by** (*cases ag*, *auto*) (*cases ad*, *auto*)

then obtain χ χ' **where**
χ: $\neg$ $I \cup$ {*Pos v*} $\models \chi$ **and**
totχ: *total-over-m* ($I \cup$ {*Pos v*}) {χ} **and**
$\chi\psi$: $\chi \in$ *fst* ψ **and**
χ': $\neg$ $I \cup$ {*Neg v*} $\models \chi'$ **and**
totχ': *total-over-m* ($I \cup$ {*Neg v*}) {χ'} **and**
$\chi'\psi$: $\chi' \in$ *fst* ψ
using *part* **unfolding** *xs* **by** *auto*
have *Posv*: *Pos v* $\notin\#$ χ **using** χ **unfolding** *true-cls-def true-lit-def* **by** *auto*
have *Negv*: *Neg v* $\notin\#$ χ' **using** χ' **unfolding** *true-cls-def true-lit-def* **by** *auto*

{
  assume *Negχ*: *Neg v* ∉# χ
  have ¬ *I* ⊨ χ **using** χ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
  moreover have *total-over-m I* {χ}
    using *Posv Negχ atm-imp-pos-or-neg-lit totχ* **unfolding** *total-over-m-def total-over-set-def*
    by *fastforce*
  ultimately have *partial-interps Leaf I* (*fst* ψ)
  and *sem-tree-size Leaf* < *sem-tree-size xs*
  and *inference*** ψ ψ
    unfolding *xs* **by** (*auto simp add*: χψ)
}
moreover {
  assume *Posχ*: *Pos v* ∉# χ′
  then have *Iχ*: ¬ *I* ⊨ χ′ **using** χ′ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
  moreover have *total-over-m I* {χ′}
    using *Negv Posχ atm-imp-pos-or-neg-lit totχ′*
    unfolding *total-over-m-def total-over-set-def* **by** *fastforce*
  ultimately have *partial-interps Leaf I* (*fst* ψ) **and**
    sem-tree-size Leaf < *sem-tree-size xs* **and**
    *inference*** ψ ψ
    using χ′ψ *Iχ* **unfolding** *xs* **by** *auto*
}
moreover {
  assume *neg*: *Neg v* ∈# χ **and** *pos*: *Pos v* ∈# χ′
  then obtain ψ′ χ*2* **where** *inf*: *rtranclp inference* ψ ψ′ **and** χ*2incl*: χ*2* ∈ *fst* ψ′
    and χχ*2-incl*: ∀ *L*. *L* ∈# χ ⟷ *L* ∈# χ*2*
    and *countχ2*: *count* χ*2* (*Neg v*) = *1*
    and φ: ∀ φ::′*v literal multiset*. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ′
    and *Iχ*: *I* ⊨ χ ⟷ *I* ⊨ χ*2*
    and *tot-impχ*: ∀ *I*′. *total-over-m I*′ {χ} ⟶ *total-over-m I*′ {χ*2*}
    using *can-decrease-count*[*of* χ *Neg v count* χ (*Neg v*) ψ *I*] χψ χ′ψ **by** *auto*

  have χ′ ∈ *fst* ψ′ **by** (*simp add*: χ′ψ φ)
  with *pos*
  obtain ψ′′ χ*2*′ **where**
  inf′: *inference*** ψ′ ψ′′
  and χ*2*′*-incl*: χ*2*′ ∈ *fst* ψ′′
  and χ′χ*2-incl*: ∀ *L*::′*v literal*. (*L* ∈# χ′) = (*L* ∈# χ*2*′)
  and *countχ2*′: *count* χ*2*′ (*Pos v*) = (*1*::*nat*)
  and φ′: ∀ φ::′*v literal multiset*. φ ∈ *fst* ψ′ ⟶ φ ∈ *fst* ψ′′
  and *Iχ*′: *I* ⊨ χ′ ⟷ *I* ⊨ χ*2*′
  and *tot-impχ*′: ∀ *I*′. *total-over-m I*′ {χ′} ⟶ *total-over-m I*′ {χ*2*′}
  using *can-decrease-count*[*of* χ′ *Pos v count* χ′ (*Pos v*) ψ′ *I*] **by** *auto*

  obtain *C* **where** χ*2*: χ*2* = *C* + {#*Neg v*#} **and** *negC*: *Neg v* ∉# *C* **and** *posC*: *Pos v* ∉# *C*
    proof −
      have ⋀*m*. *Suc 0* − *count m* (*Neg v*) = *count* (χ*2* − *m*) (*Neg v*)
        by (*simp add*: *countχ2*)
      then show *?thesis*
        using *that* **by** (*metis* (*no-types*) *One-nat-def Posv Suc-inject Suc-pred χχ2-incl*
          count-diff count-single insert-DiffM2 mem-Collect-eq multi-member-skip neg
          not-gr0 set-mset-def union-commute)
    qed

  obtain *C*′ **where**
    χ*2*′: χ*2*′ = *C*′ + {#*Pos v*#} **and**

posC': *Pos v* $\notin\#$ *C'* **and**
negC': *Neg v* $\notin\#$ *C'*
proof −
assume *a1*: $\bigwedge C'$. $[\![\chi 2' = C' + \{\#Pos\ v\#\};\ Pos\ v \notin\#\ C';\ Neg\ v \notin\#\ C']\!] \Longrightarrow$ *thesis*
have *f2*: $\bigwedge n.\ (n{::}nat) - n = 0$
by *simp*
have *Neg v* $\notin\#\ \chi 2' - \{\#Pos\ v\#\}$
using *Negv* $\chi'\chi 2$*-incl* **by** (*auto simp*: *not-in-iff*)
have *count* $\{\#Pos\ v\#\}$ (*Pos v*) = *1*
by *simp*
then show *?thesis*
by (*metis* $\chi'\chi 2$*-incl* ‹*Neg v* $\notin\#\ \chi 2' - \{\#Pos\ v\#\}$› *a1 count*$\chi 2'$ *count-diff f2*
insert-DiffM2 less-numeral-extra(*3*) *mem-Collect-eq pos set-mset-def*)
qed

have *already-used-inv* ψ'
using *rtranclp-inference-preserves-already-used-inv*[*of* ψ ψ'] *a-u-i inf* **by** *blast*
then have *a-u-i-*ψ'': *already-used-inv* ψ''
using *rtranclp-inference-preserves-already-used-inv a-u-i inf'* **unfolding** *tautology-def*
by *simp*

have *totC*: *total-over-m I* $\{C\}$
using *tot-imp*χ *tot*χ *tot-over-m-remove*[*of I Pos v C*] *negC posC* **unfolding** $\chi 2$
by (*metis total-over-m-sum uminus-Neg uminus-of-uminus-id*)
have *totC'*: *total-over-m I* $\{C'\}$
using *tot-imp*χ' *tot*χ' *total-over-m-sum tot-over-m-remove*[*of I Neg v C'*] *negC' posC'*
unfolding $\chi 2'$ **by** (*metis total-over-m-sum uminus-Neg*)
have $\neg\ I \models C + C'$
using χ $I\chi$ χ' $I\chi'$ **unfolding** $\chi 2$ $\chi 2'$ *true-cls-def* **by** *auto*
then have *part-I-*ψ''': *partial-interps Leaf I* (*fst* $\psi'' \cup \{C + C'\}$)
using *totC totC'* **by** *simp*
(*metis* ‹$\neg\ I \models C + C'$› *atms-of-ms-singleton total-over-m-def total-over-m-sum*)
{
assume $(\{\#Pos\ v\#\} + C',\ \{\#Neg\ v\#\} + C) \notin$ *snd* ψ''
then have *inf''*: *inference* ψ'' (*fst* $\psi'' \cup \{C + C'\}$, *snd* $\psi'' \cup \{(\chi 2', \chi 2)\}$)
using *add.commute* φ' $\chi 2incl$ ‹$\chi 2' \in$ *fst* ψ''› **unfolding** $\chi 2$ $\chi 2'$
by (*metis prod.collapse inference-step resolution*)
have *inference*** ψ (*fst* $\psi'' \cup \{C + C'\}$, *snd* $\psi'' \cup \{(\chi 2', \chi 2)\}$)
using *inf inf' inf''* *rtranclp-trans* **by** *auto*
moreover have *sem-tree-size Leaf* < *sem-tree-size xs* **unfolding** *xs* **by** *auto*
ultimately have *?case* **using** *part-I-*ψ''' **by** (*metis fst-conv*)
}
moreover {
assume *a*: $(\{\#Pos\ v\#\} + C',\ \{\#Neg\ v\#\} + C) \in$ *snd* ψ''
then have $(\exists\chi \in$ *fst* ψ''. ($\forall I$. *total-over-m I* $\{C{+}C'\} \longrightarrow$ *total-over-m I* $\{\chi\}$)
$\wedge$ ($\forall I$. *total-over-m I* $\{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C$))
$\vee$ *tautology* $(C' + C)$
proof −
obtain *p* **where** *p*: *Pos p* $\in\#$ $(\{\#Pos\ v\#\} + C')$ **and**
n: *Neg p* $\in\#$ $(\{\#Neg\ v\#\} + C)$ **and**
decomp: (($\exists\chi\in$*fst* ψ''.
($\forall I$. *total-over-m I* $\{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\}$
$+ ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})\}$
$\longrightarrow$ *total-over-m I* $\{\chi\}$)
$\wedge$ ($\forall I$. *total-over-m I* $\{\chi\} \longrightarrow I \models \chi$
$\longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$

```
                )
              ∨ tautology (({#Pos v#} + C') − {#Pos p#} + (({#Neg v#} + C) − {#Neg p#})))
            using a by (blast intro: allE[OF a-u-i-ψ''[unfolded subsumes-def Ball-def],
              of ({#Pos v#} + C', {#Neg v#} + C)])
          { assume p ≠ v
            then have Pos p ∈# C' ∧ Neg p ∈# C using p n by force
            then have ?thesis unfolding Bex-def by auto
          }
          moreover {
            assume p = v
           then have ?thesis using decomp by (metis add.commute add-diff-cancel-left')
          }
          ultimately show ?thesis by auto
        qed
      moreover {
        assume ∃χ ∈ fst ψ''. (∀I. total-over-m I {C+C'} ⟶ total-over-m I {χ})
          ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ C' + C)
        then obtain ϑ where ϑ: ϑ ∈ fst ψ'' and
          tot-ϑ-CC': ∀I. total-over-m I {C+C'} ⟶ total-over-m I {ϑ} and
          ϑ-inv: ∀I. total-over-m I {ϑ} ⟶ I ⊨ ϑ ⟶ I ⊨ C' + C by blast
        have partial-interps Leaf I (fst ψ'')
          using tot-ϑ-CC' ϑ ϑ-inv totC totC' ‹¬ I ⊨ C + C'› total-over-m-sum by fastforce
        moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
        ultimately have ?case by (metis inf inf' rtranclp-trans)
      }
      moreover {
        assume tautCC': tautology (C' + C)
        have total-over-m I {C'+C} using totC totC' total-over-m-sum by auto
        then have ¬tautology (C' + C)
          using ‹¬ I ⊨ C + C'› unfolding add.commute[of C C'] total-over-m-def
          unfolding tautology-def by auto
        then have False using tautCC'  unfolding tautology-def by auto
      }
      ultimately have ?case by auto
    }
    ultimately have ?case by auto
  }
  ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
  assume size-ag: sem-tree-size ag > 0
  have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
  moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
    and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ag < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
    ⟶ ( partial-interps ag (I ∪ {Pos v}) (fst ψ) ⟶
    (∃tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
      ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)))
      using IH by auto
  ultimately obtain ψ' :: 'v state and  tree' :: 'v sem-tree  where
    inf: inference** ψ ψ'
    and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
    and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
    using finite part rtranclp.rtrancl-refl a-u-i by blast
```

have *partial-interps ad* $(I \cup \{Neg\ v\})$ $(fst\ \psi')$
using *rtranclp-inference-preserve-partial-tree inf partad* **by** *metis*
then have *partial-interps* $(Node\ v\ tree'\ ad)$ I $(fst\ \psi')$ **using** *part* **by** *auto*
then have *?case* **using** *inf size size-ag part* **unfolding** *xs* **by** *fastforce*
}
moreover {
assume *size-ad*: *sem-tree-size ad* > 0
have *sem-tree-size ad* $<$ *sem-tree-size xs* **unfolding** *xs* **by** *auto*
moreover have *partag*: *partial-interps ag* $(I \cup \{Pos\ v\})$ $(fst\ \psi)$ **and**
partial-interps ad $(I \cup \{Neg\ v\})$ $(fst\ \psi)$
using *part partial-interps.simps(2)* **unfolding** *xs* **by** *metis+*
moreover have *sem-tree-size ad* $<$ *sem-tree-size xs* $\longrightarrow$ *finite* $(fst\ \psi)$ $\longrightarrow$ *already-used-inv* ψ
$\longrightarrow$ (*partial-interps ad* $(I \cup \{Neg\ v\})$ $(fst\ \psi)$
$\longrightarrow (\exists\, tree'\ \psi'.\ inference^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ (I \cup \{Neg\ v\})\ (fst\ \psi')$
$\wedge\ (sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ ad \vee sem\text{-}tree\text{-}size\ ad = 0)))$
using *IH* **by** *auto*
ultimately obtain ψ' :: *'v state* **and** *tree'* :: *'v sem-tree* **where**
inf: $inference^{**}\ \psi\ \psi'$
and *part*: *partial-interps tree'* $(I \cup \{Neg\ v\})$ $(fst\ \psi')$
and *size*: *sem-tree-size tree'* $<$ *sem-tree-size ad* $\vee$ *sem-tree-size ad* $= 0$
using *finite part rtranclp.rtrancl-refl a-u-i* **by** *blast*

have *partial-interps ag* $(I \cup \{Pos\ v\})$ $(fst\ \psi')$
using *rtranclp-inference-preserve-partial-tree inf partag* **by** *metis*
then have *partial-interps* $(Node\ v\ ag\ tree')$ I $(fst\ \psi')$ **using** *part* **by** *auto*
then have *?case* **using** *inf size size-ad* **unfolding** *xs* **by** *fastforce*
}
ultimately have *?case* **by** *auto*
}
ultimately show *?case* **by** *auto*
qed

lemma *inference-completeness-inv*:
fixes ψ :: *'v ::linorder state*
assumes
unsat: $\neg$*satisfiable* $(fst\ \psi)$ **and**
finite: *finite* $(fst\ \psi)$ **and**
a-u-v: *already-used-inv* ψ
shows $\exists\psi'.\ (inference^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')$
proof –
obtain *tree* **where** *partial-interps tree* $\{\}$ $(fst\ \psi)$
using *partial-interps-build-sem-tree-atms assms* **by** *metis*
then show *?thesis*
using *unsat finite a-u-v*
proof (*induct tree arbitrary*: ψ *rule*: *sem-tree-size*)
case (*bigger tree* ψ) **note** $H = this$
{
fix χ
assume *tree*: *tree* $=$ *Leaf*
obtain χ **where** χ: $\neg\ \{\} \models \chi$ **and** *tot*χ: *total-over-m* $\{\}$ $\{\chi\}$ **and** $\chi\psi$: $\chi \in fst\ \psi$
using *H* **unfolding** *tree* **by** *auto*
moreover have $\{\#\} = \chi$
using *tot*χ **unfolding** *total-over-m-def total-over-set-def* **by** *fastforce*
moreover have $inference^{**}\ \psi\ \psi$ **by** *auto*
ultimately have *?case* **by** *metis*
}

```
      moreover {
        fix v tree1 tree2
        assume tree: tree = Node v tree1 tree2
        obtain
          tree' ψ' where inf: inference** ψ ψ' and
          part': partial-interps tree' {} (fst ψ') and
          decrease: sem-tree-size tree' < sem-tree-size tree ∨ sem-tree-size tree = 0
          using can-decrease-tree-size[of ψ] H(2,4,5) unfolding tautology-def by meson
        have sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto
        moreover have finite (fst ψ') using rtranclp-inference-preserves-finite inf H(4) by metis
        moreover have unsatisfiable (fst ψ')
          using inference-preserves-unsat inf bigger.prems(2) by blast
        moreover have already-used-inv ψ'
          using H(5) inf rtranclp-inference-preserves-already-used-inv[of ψ ψ'] by auto
        ultimately have ?case using inf rtranclp-trans part' H(1) by fastforce
      }
      ultimately show ?case by (cases tree, auto)
  qed
qed

lemma inference-completeness:
  fixes ψ :: 'v ::linorder state
  assumes unsat: ¬satisfiable (fst ψ)
  and finite: finite (fst ψ)
  and snd ψ = {}
  shows ∃ψ'. (rtranclp inference ψ ψ' ∧ {#} ∈ fst ψ')
proof −
  have already-used-inv ψ unfolding assms by auto
  then show ?thesis using assms inference-completeness-inv by blast
qed

lemma inference-soundness:
  fixes ψ :: 'v ::linorder state
  assumes rtranclp inference ψ ψ' and {#} ∈ fst ψ'
  shows unsatisfiable (fst ψ)
  using assms by (meson rtranclp-inference-preserves-un-sat satisfiable-def true-cls-empty
    true-clss-def)

lemma inference-soundness-and-completeness:
fixes ψ :: 'v ::linorder state
assumes finite: finite (fst ψ)
and snd ψ = {}
shows (∃ψ'. (inference** ψ ψ' ∧ {#} ∈ fst ψ')) ⟷ unsatisfiable (fst ψ)
  using assms inference-completeness inference-soundness by metis
```

4.1.4 Lemma about the simplified state

```
abbreviation simplified ψ ≡ (no-step simplify ψ)

lemma simplified-count:
  assumes simp: simplified ψ and χ: χ ∈ ψ
  shows count χ L ≤ 1
proof −
  {
    let ?χ' = χ − {#L, L#}
    assume count χ L ≥ 2
```

then have *f1*: *count* $(\chi - \{\#L, L\#\} + \{\#L, L\#\})$ L = *count* χ L
by *simp*
then have $L \in\# \chi - \{\#L\#\}$
by (*metis* (*no-types*) *add.left-neutral add-diff-cancel-left′ count-union diff-diff-add diff-single-trivial insert-DiffM mem-Collect-eq multi-member-this not-gr0 set-mset-def*)
then have χ': $?\chi' + \{\#L\#\} + \{\#L\#\} = \chi$
using *f1* by (*metis diff-diff-add diff-single-eq-union in-diffD*)

have $\exists \psi'$. *simplify* ψ ψ'
by (*metis* (*no-types*, *hide-lams*) χ χ' *add.commute factoring-imp-simplify union-assoc*)
then have *False* using *simp* by *auto*
}
then show *?thesis* by *arith*
qed

lemma *simplified-no-both*:
assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
shows $\neg$ ($L \in\# \chi \wedge -L \in\# \chi$)
proof (*rule ccontr*)
assume $\neg\neg$ ($L \in\# \chi \wedge - L \in\# \chi$)
then have $L \in\# \chi \wedge - L \in\# \chi$ **by** *metis*
then obtain χ' **where** $\chi = \chi' + \{\#Pos\ (atm\text{-}of\ L)\#\} + \{\#Neg\ (atm\text{-}of\ L)\#\}$
by (*metis Neg-atm-of-iff Pos-atm-of-iff diff-union-swap insert-DiffM2 uminus-Neg uminus-Pos*)
then show *False* **using** χ *simp tautology-deletion* **by** *fastforce*
qed

lemma *simplified-not-tautology*:
assumes *simplified* $\{\psi\}$
shows ~*tautology* ψ
proof (*rule ccontr*)
assume ~ *?thesis*
then obtain p **where** $Pos\ p \in\# \psi \wedge Neg\ p \in\# \psi$ **using** *tautology-decomp* **by** *metis*
then obtain χ **where** $\psi = \chi + \{\#Pos\ p\#\} + \{\#Neg\ p\#\}$
by (*metis insert-noteq-member literal.distinct*(*1*) *multi-member-split*)
then have ~ *simplified* $\{\psi\}$ **by** (*auto intro*: *tautology-deletion*)
then show *False* **using** *assms* **by** *auto*
qed

lemma *simplified-remove*:
assumes *simplified* $\{\psi\}$
shows *simplified* $\{\psi - \{\#l\#\}\}$
proof (*rule ccontr*)
assume *ns*: $\neg$ *simplified* $\{\psi - \{\#l\#\}\}$
{
assume $l \notin\# \psi$
then have $\psi - \{\#l\#\} = \psi$ **by** *simp*
then have *False* **using** *ns assms* **by** *auto*
}
moreover {
assume $l\psi$: $l \in\# \psi$
have A: $\bigwedge A.\ A \in \{\psi - \{\#l\#\}\} \longleftrightarrow A + \{\#l\#\} \in \{\psi\}$ **by** (*auto simp add*: $l\psi$)
obtain l' **where** l': *simplify* $\{\psi - \{\#l\#\}\}$ l' **using** *ns* **by** *metis*
then have $\exists l'$. *simplify* $\{\psi\}$ l'
proof (*induction rule*: *simplify.induct*)
case (*tautology-deletion* A P)
have $\{\#Neg\ P\#\} + (\{\#Pos\ P\#\} + (A + \{\#l\#\})) \in \{\psi\}$

```
        by (metis (no-types) A add.commute tautology-deletion.hyps union-lcomm)
      then show ?thesis
         by (metis simplify.tautology-deletion[of A+{#l#} P {ψ}] add.commute)
    next
      case (condensation A L)
      have A + {#L#} + {#L#} + {#l#} ∈ {ψ}
        using A condensation.hyps by blast
      then have {#L, L#} + (A + {#l#}) ∈ {ψ}
        by (metis (no-types) union-assoc union-commute)
      then show ?case
        using factoring-imp-simplify by blast
    next
      case (subsumption A B)
      then show ?case by blast
    qed
  then have False using assms(1) by blast
 }
 ultimately show False by auto
qed


lemma in-simplified-simplified:
 assumes simp: simplified ψ and incl: ψ' ⊆ ψ
 shows simplified ψ'
proof (rule ccontr)
 assume ¬ ?thesis
 then obtain ψ'' where simplify ψ' ψ''  by metis
  then have ∃ l'. simplify ψ l'
    proof (induction rule: simplify.induct)
      case (tautology-deletion A P)
      then show ?thesis using simplify.tautology-deletion[of A P ψ] incl by blast
    next
      case (condensation A L)
      then show ?case using simplify.condensation[of A L ψ] incl by blast
    next
      case (subsumption A B)
      then show ?case using simplify.subsumption[of A ψ B] incl by auto
    qed
 then show False using assms(1) by blast
qed

lemma simplified-in:
 assumes simplified ψ
 and N ∈ ψ
 shows simplified {N}
 using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

lemma subsumes-imp-formula:
 assumes ψ ≤# φ
 shows {ψ} ⊨p φ
 unfolding true-clss-cls-def apply auto
 using assms true-cls-mono-leD by blast

lemma simplified-imp-distinct-mset-tauto:
 assumes simp: simplified ψ'
 shows distinct-mset-set ψ' and  ∀ χ ∈ ψ'. ¬tautology χ
```

proof −
  show $\forall\,\chi \in \psi'$. $\neg$*tautology* χ
    using *simp* **by** (*auto simp add*: *simplified-in simplified-not-tautology*)

  show *distinct-mset-set* ψ'
    proof (*rule ccontr*)
      assume $\neg$*?thesis*
      then obtain χ **where** $\chi \in \psi'$ **and** $\neg$*distinct-mset* χ **unfolding** *distinct-mset-set-def* **by** *auto*
      then obtain L **where** *count* χ $L \geq 2$
        unfolding *distinct-mset-def*
        by (*meson count-greater-eq-one-iff le-antisym simp simplified-count*)
      then show *False* **by** (*metis Suc-1* ‹$\chi \in \psi'$› *not-less-eq-eq simp simplified-count*)
    qed
qed

lemma *simplified-no-more-full1-simplified*:
  assumes *simplified* ψ
  shows $\neg$*full1 simplify* ψ ψ'
  using *assms* **unfolding** *full1-def* **by** (*meson tranclpD*)

4.1.5 Resolution and Invariants

inductive *resolution* :: *'v state* $\Rightarrow$ *'v state* $\Rightarrow$ *bool* **where**
full1-simp: *full1 simplify* N N' $\Longrightarrow$ *resolution* (N, *already-used*) (N', *already-used*) |
inferring: *inference* (N, *already-used*) (N', *already-used'*) $\Longrightarrow$ *simplified* N
  $\Longrightarrow$ *full simplify* N' N'' $\Longrightarrow$ *resolution* (N, *already-used*) (N'', *already-used'*)

Invariants

lemma *resolution-finite*:
  assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  using *assms* **by** (*induct rule*: *resolution.induct*)
    (*auto simp add*: *full1-def full-def rtranclp-simplify-preserves-finite*
      dest: *tranclp-into-rtranclp inference-preserves-finite*)

lemma *rtranclp-resolution-finite*:
  assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *resolution-finite*)

lemma *resolution-finite-snd*:
  assumes *resolution* ψ ψ' **and** *finite* (*snd* ψ)
  shows *finite* (*snd* ψ')
  using *assms* **apply** (*induct rule*: *resolution.induct*, *auto simp add*: *inference-preserves-finite-snd*)
  using *inference-preserves-finite-snd snd-conv* **by** *metis*

lemma *rtranclp-resolution-finite-snd*:
  assumes *resolution*** ψ ψ' **and** *finite* (*snd* ψ)
  shows *finite* (*snd* ψ')
  using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *resolution-finite-snd*)

lemma *resolution-always-simplified*:
 assumes *resolution* ψ ψ'
 shows *simplified* (*fst* ψ')
 using *assms* **by** (*induct rule*: *resolution.induct*)

(*auto simp add*: *full1-def full-def*)

lemma *tranclp-resolution-always-simplified*:
assumes *tranclp resolution* ψ ψ'
shows *simplified* (*fst* ψ')
using *assms* **by** (*induct rule*: *tranclp.induct*, *auto simp add*: *resolution-always-simplified*)

lemma *resolution-atms-of*:
assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
using *assms* **apply** (*induct rule*: *resolution.induct*)
apply(*simp add*: *rtranclp-simplify-atms-of-ms tranclp-into-rtranclp full1-def*)
by (*metis* (*no-types*, *lifting*) *contra-subsetD fst-conv full-def*
inference-preserves-atms-of-ms rtranclp-simplify-atms-of-ms subsetI)

lemma *rtranclp-resolution-atms-of*:
assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
using *resolution-atms-of rtranclp-resolution-finite* **by** *blast+*

lemma *resolution-include*:
assumes *res*: *resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
proof −
have *finite'*: *finite* (*fst* ψ') **using** *local.finite res resolution-finite* **by** *blast*
have *simplified* (*fst* ψ') **using** *res finite' resolution-always-simplified* **by** *blast*
then have *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ'))
using *simplified-in-simple-clss finite' simplified-imp-distinct-mset-tauto*[*of fst* ψ'] **by** *auto*
moreover have *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
using *res finite resolution-atms-of*[*of* ψ ψ'] **by** *auto*
ultimately show *?thesis* **by** (*meson atms-of-ms-finite local.finite order.trans rev-finite-subset*
simple-clss-mono)
qed

lemma *rtranclp-resolution-include*:
assumes *res*: *tranclp resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
using *assms* **apply** (*induct rule*: *tranclp.induct*)
apply (*simp add*: *resolution-include*)
by (*meson simple-clss-mono order-class.le-trans resolution-include*
rtranclp-resolution-atms-of rtranclp-resolution-finite tranclp-into-rtranclp)

abbreviation *already-used-all-simple*
:: (*'a literal multiset* $\times$ *'a literal multiset*) *set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool* **where**
already-used-all-simple already-used vars $\equiv$
($\forall$ (*A*, *B*) $\in$ *already-used*. *simplified* {*A*} $\wedge$ *simplified* {*B*} $\wedge$ *atms-of A* $\subseteq$ *vars* $\wedge$ *atms-of B* $\subseteq$ *vars*)

lemma *already-used-all-simple-vars-incl*:
assumes *vars* $\subseteq$ *vars'*
shows *already-used-all-simple a vars* $\Longrightarrow$ *already-used-all-simple a vars'*
using *assms* **by** *fast*

lemma *inference-clause-preserves-already-used-all-simple*:
assumes *inference-clause S S'*
and *already-used-all-simple* (*snd S*) *vars*

```
  and simplified (fst S)
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd (fst S ∪ {fst S'}, snd S')) vars
  using assms
proof (induct rule: inference-clause.induct)
  case (factoring L C N already-used)
  then show ?case by (simp add: simplified-in factoring-imp-simplify)
next
  case (resolution P C N D already-used) note H = this
  show ?case apply clarify
    proof –
      fix A B v
      assume (A, B) ∈ snd (fst (N, already-used)
        ∪ {fst (C + D, already-used ∪ {({#Pos P#} + C, {#Neg P#} + D)})},
           snd (C + D, already-used ∪ {({#Pos P#} + C, {#Neg P#} + D)}))
      then have (A, B) ∈ already-used ∨ (A, B) = ({#Pos P#} + C, {#Neg P#} + D) by auto
      moreover {
        assume (A, B) ∈ already-used
        then have simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
          using H(4) by auto
      }
      moreover {
        assume eq: (A, B) = ({#Pos P#} + C, {#Neg P#} + D)
        then have simplified {A} using simplified-in H(1,5) by auto
        moreover have simplified {B} using eq simplified-in H(2,5) by auto
        moreover have atms-of A ⊆ atms-of-ms N
          using eq H(1)
          using atms-of-atms-of-ms-mono[of A N] by auto
        moreover have atms-of B ⊆ atms-of-ms N
          using eq H(2) atms-of-atms-of-ms-mono[of B N] by auto
        ultimately have simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
          using H(6) by auto
      }
      ultimately show simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
        by fast
    qed
qed

lemma inference-preserves-already-used-all-simple:
  assumes inference S S'
  and already-used-all-simple (snd S) vars
  and simplified (fst S)
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: inference.induct)
  case (inference-step S clause already-used)
  then show ?case
    using inference-clause-preserves-already-used-all-simple[of S (clause, already-used) vars]
    by auto
qed

lemma already-used-all-simple-inv:
  assumes resolution S S'
  and already-used-all-simple (snd S) vars
  and atms-of-ms (fst S) ⊆ vars
```

shows *already-used-all-simple* (*snd* S') *vars*
using *assms*
proof (*induct rule*: *resolution.induct*)
case (*full1-simp* N N')
then show *?case* **by** *simp*
next
case (*inferring* N *already-used* N' *already-used'* N'')
then show *already-used-all-simple* (*snd* (N'', *already-used'*)) *vars*
using *inference-preserves-already-used-all-simple*[*of* (N, *already-used*)] **by** *simp*
qed

lemma *rtranclp-already-used-all-simple-inv*:
assumes *resolution*** S S'
and *already-used-all-simple* (*snd* S) *vars*
and *atms-of-ms* (*fst* S) $\subseteq$ *vars*
and *finite* (*fst* S)
shows *already-used-all-simple* (*snd* S') *vars*
using *assms*
proof (*induct rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step* S' S'') **note** *infstar* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *res* = *this*(*2*) **and**
already = *this*(*4*) **and** *atms* = *this*(*5*) **and** *finite* = *this*(*6*)
have *already-used-all-simple* (*snd* S') *vars* **using** *IH already atms finite* **by** *simp*
moreover have *atms-of-ms* (*fst* S') $\subseteq$ *atms-of-ms* (*fst* S)
by (*simp add*: *infstar local.finite rtranclp-resolution-atms-of*)
then have *atms-of-ms* (*fst* S') $\subseteq$ *vars* **using** *atms* **by** *auto*
ultimately show *?case*
using *already-used-all-simple-inv*[*OF res*] **by** *simp*
qed

lemma *inference-clause-simplified-already-used-subset*:
assumes *inference-clause* S S'
and *simplified* (*fst* S)
shows *snd* S $\subset$ *snd* S'
using *assms* **apply** (*induct rule*: *inference-clause.induct*, *auto*)
using *factoring-imp-simplify* **by** *blast*

lemma *inference-simplified-already-used-subset*:
assumes *inference* S S'
and *simplified* (*fst* S)
shows *snd* S $\subset$ *snd* S'
using *assms* **apply** (*induct rule*: *inference.induct*)
by (*metis inference-clause-simplified-already-used-subset snd-conv*)

lemma *resolution-simplified-already-used-subset*:
assumes *resolution* S S'
and *simplified* (*fst* S)
shows *snd* S $\subset$ *snd* S'
using *assms* **apply** (*induct rule*: *resolution.induct*, *simp-all add*: *full1-def*)
apply (*meson tranclpD*)
by (*metis inference-simplified-already-used-subset fst-conv snd-conv*)

lemma *tranclp-resolution-simplified-already-used-subset*:
assumes *tranclp resolution* S S'

and *simplified* (*fst S*)
shows *snd S* ⊂ *snd S'*
using *assms* **apply** (*induct rule*: *tranclp.induct*)
using *resolution-simplified-already-used-subset* **apply** *metis*
by (*meson tranclp-resolution-always-simplified resolution-simplified-already-used-subset less-trans*)

abbreviation *already-used-top vars* ≡ *simple-clss vars* × *simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *s* ⊆ *already-used-top vars*
proof
fix *x*
assume *x-s*: *x* ∈ *s*
obtain *A B* **where** *x*: *x* = (*A*, *B*) **by** (*cases x*, *auto*)
then have *simplified* {*A*} **and** *atms-of A* ⊆ *vars* **using** *assms*(*1*) *x-s* **by** *fastforce+*
then have *A*: *A* ∈ *simple-clss vars*
using *simple-clss-mono*[*of atms-of A vars*] *x assms*(*2*)
simplified-imp-distinct-mset-tauto[*of* {*A*}]
distinct-mset-not-tautology-implies-in-simple-clss **by** *fast*
moreover have *simplified* {*B*} **and** *atms-of B* ⊆ *vars* **using** *assms*(*1*) *x-s x* **by** *fast+*
then have *B*: *B* ∈ *simple-clss vars*
using *simplified-imp-distinct-mset-tauto*[*of* {*B*}]
distinct-mset-not-tautology-implies-in-simple-clss
simple-clss-mono[*of atms-of B vars*] *x assms*(*2*) **by** *fast*
ultimately show *x* ∈ *simple-clss vars* × *simple-clss vars*
unfolding *x* **by** *auto*
qed

lemma *already-used-top-finite*:
assumes *finite vars*
shows *finite* (*already-used-top vars*)
using *simple-clss-finite assms* **by** *auto*

lemma *already-used-top-increasing*:
assumes *var* ⊆ *var'* **and** *finite var'*
shows *already-used-top var* ⊆ *already-used-top var'*
using *assms simple-clss-mono* **by** *auto*

lemma *already-used-all-simple-finite*:
fixes *s* :: (*'a literal multiset* × *'a literal multiset*) *set* **and** *vars* :: *'a set*
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *finite s*
using *assms already-used-all-simple-in-already-used-top*[*OF assms*(*1*)]
rev-finite-subset[*OF already-used-top-finite*[*of vars*]] **by** *auto*

abbreviation *card-simple vars* ψ ≡ *card* (*already-used-top vars* − ψ)

lemma *resolution-card-simple-decreasing*:
assumes *res*: *resolution* ψ ψ'
and *a-u-s*: *already-used-all-simple* (*snd* ψ) *vars*
and *finite-v*: *finite vars*
and *finite-fst*: *finite* (*fst* ψ)
and *finite-snd*: *finite* (*snd* ψ)
and *simp*: *simplified* (*fst* ψ)

and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
shows *card-simple vars* (*snd* ψ') < *card-simple vars* (*snd* ψ)
proof −
let *?vars* = *vars*
let *?top* = *simple-clss ?vars* × *simple-clss ?vars*
have *1*: *card-simple vars* (*snd* ψ) = *card ?top* − *card* (*snd* ψ)
using *card-Diff-subset finite-snd already-used-all-simple-in-already-used-top*[*OF a-u-s*]
finite-v **by** *metis*
have *a-u-s'*: *already-used-all-simple* (*snd* ψ') *vars*
using *already-used-all-simple-inv res a-u-s assms*(*7*) **by** *blast*
have *f*: *finite* (*snd* ψ') **using** *already-used-all-simple-finite a-u-s' finite-v* **by** *auto*
have *2*: *card-simple vars* (*snd* ψ') = *card ?top* − *card* (*snd* ψ')
using *card-Diff-subset*[*OF f*] *already-used-all-simple-in-already-used-top*[*OF a-u-s' finite-v*]
by *auto*
have *card* (*already-used-top vars*) ≥ *card* (*snd* ψ')
using *already-used-all-simple-in-already-used-top*[*OF a-u-s' finite-v*]
card-mono[*of already-used-top vars snd* ψ'] *already-used-top-finite*[*OF finite-v*] **by** *metis*
then show *?thesis*
using *psubset-card-mono*[*OF f resolution-simplified-already-used-subset*[*OF res simp*]]
unfolding *1 2* **by** *linarith*
qed

lemma *tranclp-resolution-card-simple-decreasing*:
assumes *tranclp resolution* ψ ψ' **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
and *finite-v*: *finite vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') < *card-simple vars* (*snd* ψ)
using *assms*
proof (*induct rule*: *tranclp-induct*)
case (*base* ψ')
then show *?case* **by** (*simp add*: *resolution-card-simple-decreasing*)
next
case (*step* ψ' ψ'') **note** *res* = *this*(*1*) **and** *res'* = *this*(*2*) **and** *a-u-s* = *this*(*5*) **and**
atms = *this*(*6*) **and** *f-v* = *this*(*7*) **and** *f-fst* = *this*(*4*) **and** *H* = *this*
then have *card-simple vars* (*snd* ψ') < *card-simple vars* (*snd* ψ) **by** *auto*
moreover have *a-u-s'*: *already-used-all-simple* (*snd* ψ') *vars*
using *rtranclp-already-used-all-simple-inv*[*OF tranclp-into-rtranclp*[*OF res*] *a-u-s atms f-fst*] **.**
have *finite* (*fst* ψ')
by (*meson finite-fst res rtranclp-resolution-finite tranclp-into-rtranclp*)
moreover have *finite* (*snd* ψ') **using** *already-used-all-simple-finite*[*OF a-u-s' f-v*] **.**
moreover have *simplified* (*fst* ψ') **using** *res tranclp-resolution-always-simplified* **by** *blast*
moreover have *atms-of-ms* (*fst* ψ') $\subseteq$ *vars*
by (*meson atms f-fst order.trans res rtranclp-resolution-atms-of tranclp-into-rtranclp*)
ultimately show *?case*
using *resolution-card-simple-decreasing*[*OF res' a-u-s' f-v*] *f-v*
less-trans[*of card-simple vars* (*snd* ψ'') *card-simple vars* (*snd* ψ')
card-simple vars (*snd* ψ)]
by *blast*
qed

lemma *tranclp-resolution-card-simple-decreasing-2*:

assumes *tranclp resolution* ψ ψ'
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* ψ = {}
and *simplified* (*fst* ψ)
shows *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ') < *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ)
proof −
let *?vars* = *atms-of-ms* (*fst* ψ)
have *already-used-all-simple* (*snd* ψ) *?vars* **unfolding** *empty-snd* **by** *auto*
moreover have *atms-of-ms* (*fst* ψ) ⊆ *?vars* **by** *auto*
moreover have *finite-v*: *finite ?vars* **using** *finite-fst* **by** *auto*
moreover have *finite-snd*: *finite* (*snd* ψ) **unfolding** *empty-snd* **by** *auto*
ultimately show *?thesis*
using *assms*(*1*,*2*,*4*) *tranclp-resolution-card-simple-decreasing*[*of* ψ ψ'] **by** *presburger*
qed

well-foundness if the relation

lemma *wf-simplified-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: '*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ *simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
proof −
{
fix *a b* :: '*v*::*linorder state*
assume (*b*, *a*) ∈ {(*y*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ *simplified* (*fst x*) ∧ *finite* (*snd x*)
∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
then have
atms-of-ms (*fst a*) ⊆ *vars* **and**
simp: *simplified* (*fst a*) **and**
finite (*snd a*) **and**
finite (*fst a*) **and**
a-u-v: *already-used-all-simple* (*snd a*) *vars* **and**
res: *resolution a b* **by** *auto*
have *finite* (*already-used-top vars*) **using** *f-vars already-used-top-finite* **by** *blast*
moreover have *already-used-top vars* ⊆ *already-used-top vars* **by** *auto*
moreover have *snd b* ⊆ *already-used-top vars*
using *already-used-all-simple-in-already-used-top*[*of snd b vars*]
a-u-v already-used-all-simple-inv[*OF res*] ‹*finite* (*fst a*)› ‹*atms-of-ms* (*fst a*) ⊆ *vars*› *f-vars*
by *presburger*
moreover have *snd a* ⊂ *snd b* **using** *resolution-simplified-already-used-subset*[*OF res simp*] **.**
ultimately have *finite* (*already-used-top vars*) ∧ *already-used-top vars* ⊆ *already-used-top vars*
∧ *snd b* ⊆ *already-used-top vars* ∧ *snd a* ⊂ *snd b* **by** *metis*
}
then show *?thesis* **using** *wf-bounded-set*[*of* {(*y*:: '*v*:: *linorder state*, *x*).
(*atms-of-ms* (*fst x*) ⊆ *vars*
∧ *simplified* (*fst x*) ∧ *finite* (*snd x*) ∧ *finite* (*fst x*)∧ *already-used-all-simple* (*snd x*) *vars*)
∧ *resolution x y*} λ-. *already-used-top vars snd*] **by** *auto*
qed

lemma *wf-simplified-resolution'*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: '*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ ¬*simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
unfolding *wf-def*
apply (*simp add*: *resolution-always-simplified*)
by (*metis* (*mono-tags*, *hide-lams*) *fst-conv resolution-always-simplified*)

lemma *wf-resolution*:
  assumes *f-vars*: *finite vars*
  shows *wf* ({(*y*:: *'v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ *simplified* (*fst x*)
      $\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}
    $\cup$ {(*y*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ $\neg$ *simplified* (*fst x*) $\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*)
      $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}) (**is** *wf* (*?R* $\cup$ *?S*))
proof –
  have *Domain ?R Int Range ?S* = {} **using** *resolution-always-simplified* **by** *auto blast*
  then show *wf* (*?R* $\cup$ *?S*)
    using *wf-simplified-resolution*[*OF f-vars*] *wf-simplified-resolution'*[*OF f-vars*] *wf-Un*[*of ?R ?S*]
    by *fast*
qed

lemma *rtrancp-simplify-already-used-inv*:
  assumes *simplify*** *S S'*
  and *already-used-inv* (*S*, *N*)
  shows *already-used-inv* (*S'*, *N*)
  using *assms* **apply** *induction*
  using *simplify-preserves-already-used-inv* **by** *fast+*

lemma *full1-simplify-already-used-inv*:
  assumes *full1 simplify S S'*
  and *already-used-inv* (*S*, *N*)
  shows *already-used-inv* (*S'*, *N*)
  using *assms tranclp-into-rtranclp*[*of simplify S S'*] *rtrancp-simplify-already-used-inv*
  unfolding *full1-def* **by** *fast*

lemma *full-simplify-already-used-inv*:
  assumes *full simplify S S'*
  and *already-used-inv* (*S*, *N*)
  shows *already-used-inv* (*S'*, *N*)
  using *assms rtrancp-simplify-already-used-inv* **unfolding** *full-def* **by** *fast*
lemma *resolution-already-used-inv*:
  assumes *resolution S S'*
  and *already-used-inv S*
  shows *already-used-inv S'*
  using *assms*
proof *induction*
  case (*full1-simp N N' already-used*)
  then show *?case* **using** *full1-simplify-already-used-inv* **by** *fast*
next
  case (*inferring N already-used N' already-used' N''*) **note** *inf* = *this*(*1*) **and** *full* = *this*(*3*) **and**
    a-u-v = *this*(*4*)
  then show *?case*
    using *inference-preserves-already-used-inv*[*OF inf a-u-v*] *full-simplify-already-used-inv full*
    by *fast*
qed

lemma *rtranclp-resolution-already-used-inv*:
  assumes *resolution*** *S S'*
  and *already-used-inv S*
  shows *already-used-inv S'*
  using *assms* **apply** *induction*
  using *resolution-already-used-inv* **by** *fast+*

lemma *rtanclp-simplify-preserves-unsat*:
assumes $simplify^{**}\ \psi\ \psi'$
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
using *assms* **apply** *induction*
using *simplify-clause-preserves-sat* **by** *blast+*

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* $\psi\ \psi'$
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
using *assms rtanclp-simplify-preserves-unsat*[*of* $\psi\ \psi'$] *tranclp-into-rtranclp*
unfolding *full1-def* **by** *metis*

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* $\psi\ \psi'$
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
using *assms rtanclp-simplify-preserves-unsat*[*of* $\psi\ \psi'$] **unfolding** *full-def* **by** *metis*

lemma *resolution-preserves-unsat*:
assumes *resolution* $\psi\ \psi'$
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
using *assms* **apply** (*induct rule*: *resolution.induct*)
using *full1-simplify-preserves-unsat* **apply** (*metis fst-conv*)
using *full-simplify-preserves-unsat simplify-preserves-unsat* **by** *fastforce*

lemma *rtranclp-resolution-preserves-unsat*:
assumes $resolution^{**}\ \psi\ \psi'$
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
using *assms* **apply** *induction*
using *resolution-preserves-unsat* **by** *fast+*

lemma *rtranclp-simplify-preserve-partial-tree*:
assumes $simplify^{**}\ N\ N'$
and *partial-interps t I N*
shows *partial-interps t I* N'
using *assms* **apply** (*induction*, *simp*)
using *simplify-preserve-partial-tree* **by** *metis*

lemma *full1-simplify-preserve-partial-tree*:
assumes *full1 simplify* $N\ N'$
and *partial-interps t I N*
shows *partial-interps t I* N'
using *assms rtranclp-simplify-preserve-partial-tree*[*of* $N\ N'\ t\ I$] *tranclp-into-rtranclp*
unfolding *full1-def* **by** *fast*

lemma *full-simplify-preserve-partial-tree*:
assumes *full simplify* $N\ N'$
and *partial-interps t I N*
shows *partial-interps t I* N'
using *assms rtranclp-simplify-preserve-partial-tree*[*of* $N\ N'\ t\ I$] *tranclp-into-rtranclp*
unfolding *full-def* **by** *fast*

lemma *resolution-preserve-partial-tree*:
assumes *resolution* $S\ S'$
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst* S')
using *assms* **apply** *induction*

using *full1-simplify-preserve-partial-tree fst-conv* **apply** *metis*
using *full-simplify-preserve-partial-tree inference-preserve-partial-tree* **by** *fastforce*

lemma *rtranclp-resolution-preserve-partial-tree*:
assumes $resolution^{**}\ S\ S'$
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst S'*)
using *assms* **apply** *induction*
using *resolution-preserve-partial-tree* **by** *fast+*
thm *nat-less-induct nat.induct*

lemma *nat-ge-induct*[*case-names 0 Suc*]:
assumes *P 0*
and $\bigwedge n.\ (\bigwedge m.\ m<Suc\ n \Longrightarrow P\ m) \Longrightarrow P\ (Suc\ n)$
shows *P n*
using *assms* **apply** (*induct rule*: *nat-less-induct*)
by (*rename-tac n*, *case-tac n*) *auto*

lemma *wf-always-more-step-False*:
assumes *wf R*
shows $(\forall x.\ \exists z.\ (z, x)\in R) \Longrightarrow False$
using *assms* **unfolding** *wf-def* **by** (*meson Domain.DomainI assms wfE-min*)

lemma *finite-finite-mset-element-of-mset*[*simp*]:
assumes *finite N*
shows *finite* $\{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\}$
using *assms*
proof (*induction N rule*: *finite-induct*)
case *empty*
show *?case* **by** *auto*
next
case (*insert x N*) **note** *finite* = *this*(*1*) **and** *IH* = *this*(*3*)
have $\{f\ \varphi\ L\ |\varphi\ L.\ (\varphi = x \vee \varphi \in N) \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\} \subseteq \{f\ x\ L\ |\ L.\ L \in\#\ x \wedge P\ x\ L\}$
$\cup \{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\}$ **by** *auto*
moreover have *finite* $\{f\ x\ L\ |\ L.\ L \in\#\ x\}$ **by** *auto*
ultimately show *?case* **using** *IH finite-subset* **by** *fastforce*
qed

definition *sum-count-ge-2* :: *'a multiset set* ⇒ *nat* (Ξ) **where**
sum-count-ge-2 ≡ *folding.F* ($\lambda\varphi.$ *op* +(*msetsum* {#*count* φ *L* |*L* ∈# φ. *2* ≤ *count* φ *L*#})) *0*

interpretation *sum-count-ge-2*:
folding ($\lambda\varphi.$ *op* +(*msetsum* {#*count* φ *L* |*L* ∈# φ. *2* ≤ *count* φ *L*#})) *0*
rewrites
folding.F ($\lambda\varphi.$ *op* +(*msetsum* {#*count* φ *L* |*L* ∈# φ. *2* ≤ *count* φ *L*#})) *0* = *sum-count-ge-2*
proof −
show *folding* ($\lambda\varphi.$ *op* + (*msetsum* (*image-mset* (*count* φ) {# *L* ∈# φ. *2* ≤ *count* φ *L*#})))
by *standard auto*
then interpret *sum-count-ge-2*:
folding ($\lambda\varphi.$ *op* +(*msetsum* {#*count* φ *L* |*L* ∈# φ. *2* ≤ *count* φ *L*#})) *0* **.**
show *folding.F* ($\lambda\varphi.$ *op* + (*msetsum* (*image-mset* (*count* φ) {# *L* ∈# φ. *2* ≤ *count* φ *L*#}))) *0*
= *sum-count-ge-2* **by** (*auto simp add*: *sum-count-ge-2-def*)
qed

lemma *finite-incl-le-setsum*:
$finite\ (B{::}'a\ multiset\ set) \Longrightarrow A \subseteq B \Longrightarrow \Xi\ A \leq \Xi\ B$
proof (*induction arbitrary*:A *rule*: *finite-induct*)
  case *empty*
  then show *?case* **by** *simp*
next
  case (*insert a F*) **note** $finite = this(1)$ **and** $aF = this(2)$ **and** $IH = this(3)$ **and** $AF = this(4)$
  show *?case*
    proof ($cases\ a \in A$)
      assume $a \notin A$
      then have $A \subseteq F$ **using** *AF* **by** *auto*
      then show *?case* **using** *IH*[*of A*] **by** (*simp add*: *aF local.finite*)
    next
      assume *aA*: $a \in A$
      then have $A - \{a\} \subseteq F$ **using** *AF* **by** *auto*
      then have $\Xi\ (A - \{a\}) \leq \Xi\ F$ **using** *IH* **by** *blast*
      then show *?case*
        proof −
          obtain $nn :: nat \Rightarrow nat \Rightarrow nat$ **where**
            $\forall x0\ x1.\ (\exists v2.\ x0 = x1 + v2) = (x0 = x1 + nn\ x0\ x1)$
            by *moura*
          then have $\Xi\ F = \Xi\ (A - \{a\}) + nn\ (\Xi\ F)\ (\Xi\ (A - \{a\}))$
            by (*meson* ‹$\Xi\ (A - \{a\}) \leq \Xi\ F$› *le-iff-add*)
          then show *?thesis*
            by (*metis* (*no-types*) *le-iff-add aA aF add.assoc finite.insertI finite-subset*
              insert.prems local.finite sum-count-ge-2.insert sum-count-ge-2.remove)
        qed
    qed
qed

lemma *simplify-finite-measure-decrease*:
  $simplify\ N\ N' \Longrightarrow finite\ N \Longrightarrow card\ N' + \Xi\ N' < card\ N + \Xi\ N$
proof (*induction rule*: *simplify.induct*)
  case (*tautology-deletion A P*) **note** $an = this(1)$ **and** $fin = this(2)$
  let $?N' = N - \{A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}\}$
  have $card\ ?N' < card\ N$
    by (*meson card-Diff1-less tautology-deletion.hyps tautology-deletion.prems*)
  moreover have $?N' \subseteq N$ **by** *auto*
  then have $sum\text{-}count\text{-}ge\text{-}2\ ?N' \leq sum\text{-}count\text{-}ge\text{-}2\ N$ **using** *finite-incl-le-setsum*[*OF fin*] **by** *blast*
  ultimately show *?case* **by** *linarith*
next
  case (*condensation A L*) **note** $AN = this(1)$ **and** $fin = this(2)$
  let $?C' = A + \{\#L\#\}$
  let $?C = A + \{\#L\#\} + \{\#L\#\}$
  let $?N' = N - \{?C\} \cup \{?C'\}$
  have $card\ ?N' \leq card\ N$
    using *AN* **by** (*metis* (*no-types*, *lifting*) *Diff-subset Un-empty-right Un-insert-right card.remove*
      card-insert-if card-mono fin finite-Diff order-refl)
  moreover have $\Xi\ \{?C'\} < \Xi\ \{?C\}$
    proof −
      have *mset-decomp*:
        $\{\#\ La \in\#\ A.\ (L = La \longrightarrow La \in\#\ A) \wedge (L \neq La \longrightarrow 2 \leq count\ A\ La)\#\}$
        $= \{\#\ La \in\#\ A.\ L \neq La \wedge 2 \leq count\ A\ La\#\} +$
          $\{\#\ La \in\#\ A.\ L = La \wedge Suc\ 0 \leq count\ A\ L\#\}$
          by (*auto simp*: *multiset-eq-iff ac-simps*)
      have *mset-decomp2*: $\{\#\ La \in\#\ A.\ L \neq La \longrightarrow 2 \leq count\ A\ La\#\} =$

```
      {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
      by (auto simp: multiset-eq-iff)
    show ?thesis
      by (auto simp: mset-decomp mset-decomp2 filter-mset-eq ac-simps)
   qed
  have Ξ ?N' < Ξ N
   proof cases
    assume a1: ?C' ∈ N
    then show ?thesis
      proof −
       have f2: ⋀m M. insert (m::'a literal multiset) (M − {m}) = M ∪ {} ∨ m ∉ M
         using Un-empty-right insert-Diff by blast
       have f3: ⋀m M Ma. insert (m::'a literal multiset) M − insert m Ma = M − insert m Ma
         by simp
       then have f4: ⋀M m. M − {m::'a literal multiset} = M ∪ {} ∨ m ∈ M
         using Diff-insert-absorb Un-empty-right by fastforce
       have f5: insert (A + {#L#} + {#L#}) N = N
         using f3 f2 Un-empty-right condensation.hyps insert-iff by fastforce
       have ⋀m M. insert (m::'a literal multiset) M = M ∪ {} ∨ m ∉ M
         using f3 f2 Un-empty-right add.right-neutral insert-iff by fastforce
       then have Ξ (N − {A + {#L#} + {#L#}}) < Ξ N
         using f5 f4 by (metis Un-empty-right ‹Ξ {A + {#L#}} < Ξ {A + {#L#} + {#L#}}›
           add.right-neutral add-diff-cancel-left' add-gr-0 diff-less fin finite.emptyI not-le
           sum-count-ge-2.empty sum-count-ge-2.insert-remove trans-le-add2)
       then show ?thesis
         using f3 f2 a1 by (metis (no-types) Un-empty-right Un-insert-right condensation.hyps
           insert-iff multi-self-add-other-not-self)
      qed
   next
    assume ?C' ∉ N
    have mset-decomp:
      {# La ∈# A. (L = La ⟶ Suc 0 ≤ count A La) ∧ (L ≠ La ⟶ 2 ≤ count A La)#}
      = {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} +
        {# La ∈# A. L = La ∧ Suc 0 ≤ count A L#}
         by (auto simp: multiset-eq-iff ac-simps)
    have mset-decomp2: {# La ∈# A. L ≠ La ⟶ 2 ≤ count A La#} =
      {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
      by (auto simp: multiset-eq-iff)

    show ?thesis
      using ‹Ξ {A + {#L#}} < Ξ {A + {#L#} + {#L#}}› condensation.hyps fin
      sum-count-ge-2.remove[of - A + {#L#} + {#L#}] ‹?C' ∉ N›
      by (auto simp: mset-decomp mset-decomp2 filter-mset-eq)
   qed
  ultimately show ?case by linarith
next
  case (subsumption A B) note AN = this(1) and AB = this(2) and BN = this(3) and fin = this(4)
  have card (N − {B}) < card N using BN by (meson card-Diff1-less subsumption.prems)
  moreover have Ξ (N − {B}) ≤ Ξ N
   by (simp add: Diff-subset finite-incl-le-setsum subsumption.prems)
  ultimately show ?case by linarith
qed

lemma simplify-terminates:
  wf {(N', N). finite N ∧ simplify N N'}
  using assms apply (rule wfP-if-measure[of finite simplify λN. card N + Ξ N])
```

using *simplify-finite-measure-decrease* **by** *blast*

lemma *wf-terminates*:
assumes *wf r*
shows $\exists N'.(N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r)$
proof −
let $?P = \lambda N. (\exists N'.(N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r))$
have $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P\ y) \longrightarrow ?P\ x)$
proof *clarify*
fix x
assume H: $\forall y. (y, x) \in r \longrightarrow ?P\ y$
{ assume $\exists y. (y, x) \in r$
then obtain y **where** y: $(y, x) \in r$ **by** *blast*
then have $?P\ y$ **using** H **by** *blast*
then have $?P\ x$ **using** y **by** (*meson rtrancl.rtrancl-into-rtrancl*)
}
moreover {
assume $\neg(\exists y. (y, x) \in r)$
then have $?P\ x$ **by** *auto*
}
ultimately show $?P\ x$ **by** *blast*
qed
moreover have $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P\ y) \longrightarrow ?P\ x) \longrightarrow All\ ?P$
using *assms* **unfolding** *wf-def* **by** (*rule allE*)
ultimately have $All\ ?P$ **by** *blast*
then show $?P\ N$ **by** *blast*
qed

lemma *rtranclp-simplify-terminates*:
assumes *fin*: *finite N*
shows $\exists N'.\ simplify^{**}\ N\ N' \wedge simplified\ N'$
proof −
have H: $\{(N', N).\ finite\ N \wedge simplify\ N\ N'\} = \{(N', N).\ simplify\ N\ N' \wedge finite\ N\}$ **by** *auto*
then have *wf*: $wf\ \{(N', N).\ simplify\ N\ N' \wedge finite\ N\}$
using *simplify-terminates* **by** (*simp add*: H)
obtain N' **where** N': $(N', N) \in \{(b, a).\ simplify\ a\ b \wedge finite\ a\}^*$ **and**
more: $(\forall N''. (N'', N') \notin \{(b, a).\ simplify\ a\ b \wedge finite\ a\})$
using *Prop-Resolution.wf-terminates*[*OF wf*, *of N*] **by** *blast*
have *1*: $simplify^{**}\ N\ N'$
using N' **by** (*induction rule*: *rtrancl.induct*) *auto*
then have *finite* N' **using** *fin rtranclp-simplify-preserves-finite* **by** *blast*
then have *2*: $\forall N''. \neg simplify\ N'\ N''$ **using** *more* **by** *auto*

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

lemma *finite-simplified-full1-simp*:
assumes *finite N*
shows $simplified\ N \vee (\exists N'.\ full1\ simplify\ N\ N')$
using *rtranclp-simplify-terminates*[*OF assms*] **unfolding** *full1-def*
by (*metis Nitpick.rtranclp-unfold*)

lemma *finite-simplified-full-simp*:
assumes *finite N*

shows $\exists N'$. *full simplify N N'*
using *rtranclp-simplify-terminates*[*OF assms*] **unfolding** *full-def* **by** *metis*

lemma *can-decrease-tree-size-resolution*:
fixes ψ :: *'v state* **and** *tree* :: *'v sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
and *simplified* (*fst* ψ)
shows $\exists$(*tree'*:: *'v sem-tree*) ψ'. *resolution*** ψ ψ' $\wedge$ *partial-interps tree' I* (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* $<$ *sem-tree-size tree* $\vee$ *sem-tree-size tree* $=$ *0*)
using *assms*
proof (*induct arbitrary*: *I rule*: *sem-tree-size*)
case (*bigger xs I*) **note** *IH* = *this*(*1*) **and** *finite* = *this*(*2*) **and** *a-u-i* = *this*(*3*) **and** *part* = *this*(*4*)
and *simp* = *this*(*5*)

{ assume *sem-tree-size xs* = *0*
then have *?case* **using** *part* **by** *blast*
}

moreover {
assume *sn0*: *sem-tree-size xs* > *0*
obtain *ag ad v* **where** *xs*: *xs* = *Node v ag ad* **using** *sn0* **by** (*cases xs*, *auto*)
{
assume *sem-tree-size ag* = *0* $\wedge$ *sem-tree-size ad* = *0*
then have *ag*: *ag* = *Leaf* **and** *ad*: *ad* = *Leaf* **by** (*cases ag*, *auto*, *cases ad*, *auto*)

then obtain χ χ' **where**
χ: $\neg$ *I* $\cup$ {*Pos v*} $\models$ χ **and**
totχ: *total-over-m* (*I* $\cup$ {*Pos v*}) {χ} **and**
$\chi\psi$: $\chi \in$ *fst* ψ **and**
χ': $\neg$ *I* $\cup$ {*Neg v*} $\models$ χ' **and**
totχ': *total-over-m* (*I* $\cup$ {*Neg v*}) {χ'} **and** $\chi'\psi$: $\chi' \in$ *fst* ψ
using *part* **unfolding** *xs* **by** *auto*
have *Posv*: *Pos v* $\notin\#$ χ **using** χ **unfolding** *true-cls-def true-lit-def* **by** *auto*
have *Negv*: *Neg v* $\notin\#$ χ' **using** χ' **unfolding** *true-cls-def true-lit-def* **by** *auto*
{
assume *Neg*χ: *Neg v* $\notin\#$ χ
then have $\neg$ *I* $\models$ χ **using** χ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m I* {χ}
using *Posv Neg*χ *atm-imp-pos-or-neg-lit tot*χ **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*
ultimately have *partial-interps Leaf I* (*fst* ψ)
and *sem-tree-size Leaf* < *sem-tree-size xs*
and *resolution*** ψ ψ
unfolding *xs* **by** (*auto simp add*: $\chi\psi$)
}
moreover {
assume *Pos*χ: *Pos v* $\notin\#$ χ'
then have *I*χ: $\neg$ *I* $\models$ χ' **using** χ' *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m I* {χ'}
using *Negv Pos*χ *atm-imp-pos-or-neg-lit tot*χ'
unfolding *total-over-m-def total-over-set-def* **by** *fastforce*
ultimately have *partial-interps Leaf I* (*fst* ψ)
and *sem-tree-size Leaf* < *sem-tree-size xs*
and *resolution*** ψ ψ
using $\chi'\psi$ *I*χ **unfolding** *xs* **by** *auto*

}
moreover {
assume *neg*: $Neg\ v \in\#\ \chi$ **and** *pos*: $Pos\ v \in\#\ \chi'$
have $count\ \chi\ (Neg\ v) = 1$
using *simplified-count*[*OF simp* $\chi\psi$] *neg*
by (*simp add*: *dual-order.antisym*)
have $count\ \chi'\ (Pos\ v) = 1$
using *simplified-count*[*OF simp* $\chi'\psi$] *pos*
by (*simp add*: *dual-order.antisym*)

obtain C **where** χC: $\chi = C + \{\#Neg\ v\#\}$ **and** *negC*: $Neg\ v \notin\#\ C$ **and** *posC*: $Pos\ v \notin\#\ C$
by (*metis* (*no-types*, *lifting*) *One-nat-def Posv Suc-eq-plus1-left* ‹$count\ \chi\ (Neg\ v) = 1$›
add-diff-cancel-left′ count-diff count-greater-eq-one-iff count-single insert-DiffM
insert-DiffM2 less-numeral-extra(*3*) *multi-member-skip not-le not-less-eq-eq*)

obtain C' **where**
$\chi C'$: $\chi' = C' + \{\#Pos\ v\#\}$ **and**
posC′: $Pos\ v \notin\#\ C'$ **and**
negC′: $Neg\ v \notin\#\ C'$
by (*metis* (*no-types*, *lifting*) *One-nat-def Negv Suc-eq-plus1-left* ‹$count\ \chi'\ (Pos\ v) = 1$›
add-diff-cancel-left′ count-diff count-greater-eq-one-iff count-single insert-DiffM
insert-DiffM2 less-numeral-extra(*3*) *multi-member-skip not-le not-less-eq-eq*)

have *totC*: *total-over-m* $I\ \{C\}$
using *tot*χ *tot-over-m-remove*[*of I Pos v C*] *negC posC* **unfolding** χC
by (*metis total-over-m-sum uminus-Neg uminus-of-uminus-id*)
have *totC′*: *total-over-m* $I\ \{C'\}$
using *tot*χ' *total-over-m-sum tot-over-m-remove*[*of I Neg v C′*] *negC′ posC′*
unfolding $\chi C'$ **by** (*metis total-over-m-sum uminus-Neg*)
have $\neg\ I \models C + C'$
using $\chi\ \chi'\ \chi C\ \chi C'$ **by** *auto*
then have *part-I-*ψ''': *partial-interps Leaf I* (*fst* $\psi \cup \{C + C'\}$)
using *totC totC′* ‹$\neg\ I \models C + C'$› **by** (*metis Un-insert-right insertI1*
partial-interps.simps(*1*) *total-over-m-sum*)
{
assume $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \notin$ *snd* ψ
then have *inf′′*: *inference* ψ (*fst* $\psi \cup \{C + C'\}$, *snd* $\psi \cup \{(\chi', \chi)\}$)
by (*metis* $\chi'\psi\ \chi C\ \chi C'\ \chi\psi$ *add.commute inference-step prod.collapse resolution*)
obtain N' **where** *full*: *full simplify* (*fst* $\psi \cup \{C + C'\}$) N'
by (*metis finite-simplified-full-simp fst-conv inf′′ inference-preserves-finite*
local.finite)
have *resolution* ψ (N', *snd* $\psi \cup \{(\chi', \chi)\}$)
using *resolution.intros*(*2*)[*OF - simp full*, *of snd* ψ *snd* $\psi \cup \{(\chi', \chi)\}$] *inf′′*
by (*metis surjective-pairing*)
moreover have *partial-interps Leaf I* N'
using *full-simplify-preserve-partial-tree*[*OF full part-I-*ψ''] **.**
moreover have *sem-tree-size Leaf* < *sem-tree-size xs* **unfolding** *xs* **by** *auto*
ultimately have *?case*
by (*metis* (*no-types*) *prod.sel*(*1*) *rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl*)
}
moreover {
assume *a*: $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \in$ *snd* ψ
then have $(\exists \chi \in$ *fst* ψ. $(\forall I.$ *total-over-m* $I\ \{C+C'\} \longrightarrow$ *total-over-m* $I\ \{\chi\})$
$\wedge\ (\forall I.$ *total-over-m* $I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)) \vee$ *tautology* $(C' + C)$
proof −
obtain p **where** p: $Pos\ p \in\#\ (\{\#Pos\ v\#\} + C') \wedge Neg\ p \in\#\ (\{\#Neg\ v\#\} + C)$

$\wedge((\exists \chi \in fst\ \psi.\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\}) \wedge (\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))) \vee tautology\ ((\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})))$

using *a* **by** (*blast intro*: *allE*[*OF a-u-i*[*unfolded subsumes-def Ball-def*],
of $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C)$]])
{ assume $p \neq v$
then have $Pos\ p \in\#\ C' \wedge Neg\ p \in\#\ C$ **using** *p* **by** *force*
then have *?thesis* **by** *auto*
}
moreover {
assume $p = v$
then have *?thesis* **using** *p* **by** (*metis add.commute add-diff-cancel-left'*)
}
ultimately show *?thesis* **by** *auto*
qed
moreover {
assume $\exists \chi \in fst\ \psi.\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\})$
$\wedge\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$
then obtain ϑ **where**
ϑ: $\vartheta \in fst\ \psi$ **and**
tot-ϑ-CC': $\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\vartheta\}$ **and**
ϑ-inv: $\forall I.\ total\text{-}over\text{-}m\ I\ \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$ **by** *blast*
have *partial-interps Leaf I* (*fst* ψ)
using *tot-ϑ-CC' ϑ ϑ-inv totC totC'* $\langle\neg\ I \models C + C'\rangle$ *total-over-m-sum* **by** *fastforce*
moreover have *sem-tree-size Leaf* < *sem-tree-size xs* **unfolding** *xs* **by** *auto*
ultimately have *?case* **by** *blast*
}
moreover {
assume *tautCC'*: *tautology* $(C' + C)$
have *total-over-m I* $\{C'+C\}$ **using** *totC totC' total-over-m-sum* **by** *auto*
then have $\neg$*tautology* $(C' + C)$
using $\langle\neg\ I \models C + C'\rangle$ **unfolding** *add.commute*[*of C C'*] *total-over-m-def*
unfolding *tautology-def* **by** *auto*
then have *False* **using** *tautCC'* **unfolding** *tautology-def* **by** *auto*
}
ultimately have *?case* **by** *auto*
}
ultimately have *?case* **by** *auto*
}
ultimately have *?case* **using** *part* **by** (*metis* (*no-types*) *sem-tree-size.simps*(*1*))
}
moreover {
assume *size-ag*: *sem-tree-size ag* > *0*
have *sem-tree-size ag* < *sem-tree-size xs* **unfolding** *xs* **by** *auto*
moreover have *partial-interps ag* $(I \cup \{Pos\ v\})$ (*fst* ψ)
and *partad*: *partial-interps ad* $(I \cup \{Neg\ v\})$ (*fst* ψ)
using *part partial-interps.simps*(*2*) **unfolding** *xs* **by** *metis+*
moreover
have *sem-tree-size ag* < *sem-tree-size xs* $\Longrightarrow$ *finite* (*fst* ψ) $\Longrightarrow$ *already-used-inv* ψ
$\Longrightarrow$ *partial-interps ag* $(I \cup \{Pos\ v\})$ (*fst* ψ) $\Longrightarrow$ *simplified* (*fst* ψ)
$\Longrightarrow \exists\, tree'\ \psi'.\ resolution^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ (I \cup \{Pos\ v\})\ (fst\ \psi')$
$\wedge\ (sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ ag \vee sem\text{-}tree\text{-}size\ ag = 0)$
using *IH*[*of ag I* $\cup$ $\{Pos\ v\}$] **by** *auto*
ultimately obtain ψ' :: *'v state* **and** *tree'* :: *'v sem-tree* **where**
inf: $resolution^{**}\ \psi\ \psi'$

```
      and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
      and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
      using finite part rtranclp.rtrancl-refl a-u-i simp by blast

    have partial-interps ad (I ∪ {Neg v}) (fst ψ')
      using rtranclp-resolution-preserve-partial-tree inf partad by fast
    then have partial-interps (Node v tree' ad) I (fst ψ') using part by auto
    then have ?case using inf size size-ag part unfolding xs by fastforce
  }
  moreover {
    assume size-ad: sem-tree-size ad > 0
    have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
    moreover
      have
        partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
        partial-interps ad (I ∪ {Neg v}) (fst ψ)
        using part partial-interps.simps(2) unfolding xs by metis+
    moreover have sem-tree-size ad < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
      ⟶ ( partial-interps ad (I ∪ {Neg v}) (fst ψ) ⟶ simplified (fst ψ)
      ⟶ (∃ tree' ψ'. resolution** ψ ψ' ∧ partial-interps tree' (I ∪ {Neg v}) (fst ψ')
          ∧ (sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0)))
      using IH by blast
    ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
      inf: resolution** ψ ψ'
      and part: partial-interps tree' (I ∪ {Neg v}) (fst ψ')
      and size: sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0
      using finite part rtranclp.rtrancl-refl a-u-i simp by blast

    have partial-interps ag (I ∪ {Pos v}) (fst ψ')
      using rtranclp-resolution-preserve-partial-tree inf partag by fast
    then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
    then have ?case using inf size size-ad unfolding xs by fastforce
  }
   ultimately have ?case by auto
 }
 ultimately show ?case by auto
qed

lemma resolution-completeness-inv:
 fixes ψ :: 'v ::linorder state
 assumes
   unsat: ¬satisfiable (fst ψ) and
   finite: finite (fst ψ) and
   a-u-v: already-used-inv ψ
 shows ∃ψ'. (resolution** ψ ψ' ∧ {#} ∈ fst ψ')
proof −
 obtain tree where partial-interps tree {} (fst ψ)
   using partial-interps-build-sem-tree-atms assms by metis
 then show ?thesis
   using unsat finite a-u-v
   proof (induct tree arbitrary: ψ rule: sem-tree-size)
     case (bigger tree ψ) note H = this
     {
       fix χ
       assume tree: tree = Leaf
       obtain χ where χ: ¬ {} ⊨ χ and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
```

using *H* **unfolding** *tree* **by** *auto*
moreover have $\{\#\} = \chi$
using *H atms-empty-iff-empty totχ*
unfolding *true-cls-def total-over-m-def total-over-set-def* **by** *fastforce*
moreover have *resolution*** ψ ψ **by** *auto*
ultimately have *?case* **by** *metis*
}
moreover {
fix *v tree1 tree2*
assume *tree*: *tree* = *Node v tree1 tree2*
obtain ψ_0 **where** ψ_0: *resolution*** ψ ψ_0 **and** *simp*: *simplified* (*fst* ψ_0)
proof −
{ assume *simplified* (*fst* ψ)
moreover have *resolution*** ψ ψ **by** *auto*
ultimately have *thesis* **using** *that* **by** *blast*
}
moreover {
assume ¬*simplified* (*fst* ψ)
then have $\exists\psi'$. *full1 simplify* (*fst* ψ) ψ'
by (*metis Nitpick.rtranclp-unfold bigger.prems*(*3*) *full1-def*
rtranclp-simplify-terminates)
then obtain *N* **where** *full1 simplify* (*fst* ψ) *N* **by** *metis*
then have *resolution* ψ (*N*, *snd* ψ)
using *resolution.intros*(*1*)[*of fst* ψ *N snd* ψ] **by** *auto*
moreover have *simplified N*
using ‹*full1 simplify* (*fst* ψ) *N*› **unfolding** *full1-def* **by** *blast*
ultimately have *?thesis* **using** *that* **by** *force*
}
ultimately show *?thesis* **by** *auto*
qed

have *p*: *partial-interps tree* {} (*fst* ψ_0)
and *uns*: *unsatisfiable* (*fst* ψ_0)
and *f*: *finite* (*fst* ψ_0)
and *a-u-v*: *already-used-inv* ψ_0
using ψ_0 *bigger.prems*(*1*) *rtranclp-resolution-preserve-partial-tree* **apply** *blast*
using ψ_0 *bigger.prems*(*2*) *rtranclp-resolution-preserves-unsat* **apply** *blast*
using ψ_0 *bigger.prems*(*3*) *rtranclp-resolution-finite* **apply** *blast*
using *rtranclp-resolution-already-used-inv*[*OF* ψ_0 *bigger.prems*(*4*)] **by** *blast*
obtain *tree′* ψ' **where**
inf: *resolution*** ψ_0 ψ' **and**
part′: *partial-interps tree′* {} (*fst* ψ') **and**
decrease: *sem-tree-size tree′* < *sem-tree-size tree* ∨ *sem-tree-size tree* = *0*
using *can-decrease-tree-size-resolution*[*OF f a-u-v p simp*] **unfolding** *tautology-def*
by *meson*
have *s*: *sem-tree-size tree′* < *sem-tree-size tree* **using** *decrease* **unfolding** *tree* **by** *auto*
have *fin*: *finite* (*fst* ψ')
using *f inf rtranclp-resolution-finite* **by** *blast*
have *unsat*: *unsatisfiable* (*fst* ψ')
using *rtranclp-resolution-preserves-unsat inf uns* **by** *metis*
have *a-u-i′*: *already-used-inv* ψ'
using *a-u-v inf rtranclp-resolution-already-used-inv*[*of* ψ_0 ψ'] **by** *auto*
have *?case*
using *inf rtranclp-trans*[*of resolution*] *H*(*1*)[*OF s part′ unsat fin a-u-i′*] ψ_0 **by** *blast*
}

ultimately show *?case* **by** (*cases tree*, *auto*)
qed
qed

lemma *resolution-preserves-already-used-inv*:
assumes *resolution S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms*
apply (*induct rule*: *resolution.induct*)
apply (*rule full1-simplify-already-used-inv*; *simp*)
apply (*rule full-simplify-already-used-inv*, *simp*)
apply (*rule inference-preserves-already-used-inv*, *simp*)
apply *blast*
done

lemma *rtranclp-resolution-preserves-already-used-inv*:
assumes *resolution*** *S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms*
apply (*induct rule*: *rtranclp-induct*)
apply *simp*
using *resolution-preserves-already-used-inv* **by** *fast*

lemma *resolution-completeness*:
fixes ψ :: *'v* ::*linorder state*
assumes *unsat*: ¬*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows $\exists \psi'$. (*resolution*** ψ ψ' ∧ {#} ∈ *fst* ψ')
proof −
have *already-used-inv* ψ **unfolding** *assms* **by** *auto*
then show *?thesis* **using** *assms resolution-completeness-inv* **by** *blast*
qed

lemma *rtranclp-preserves-sat*:
assumes *simplify*** *S S'*
and *satisfiable S*
shows *satisfiable S'*
using *assms* **apply** *induction*
apply *simp*
by (*meson satisfiable-carac satisfiable-def simplify-preserves-un-sat-eq*)

lemma *resolution-preserves-sat*:
assumes *resolution S S'*
and *satisfiable* (*fst S*)
shows *satisfiable* (*fst S'*)
using *assms* **apply** (*induction rule*: *resolution.induct*)
using *rtranclp-preserves-sat tranclp-into-rtranclp* **unfolding** *full1-def* **apply** *fastforce*
by (*metis fst-conv full-def inference-preserves-un-sat rtranclp-preserves-sat satisfiable-carac' satisfiable-def*)

lemma *rtranclp-resolution-preserves-sat*:
assumes *resolution*** *S S'*
and *satisfiable* (*fst S*)

shows *satisfiable* (*fst* S')
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
apply *simp*
using *resolution-preserves-sat* **by** *blast*

lemma *resolution-soundness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *resolution*** ψ ψ' **and** $\{\#\} \in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
using *assms* **by** (*meson rtranclp-resolution-preserves-sat satisfiable-def true-cls-empty true-clss-def*)

lemma *resolution-soundness-and-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd*: *snd* $\psi = \{\}$
shows $(\exists\,\psi'.\ (\mathit{resolution}^{**}\ \psi\ \psi' \wedge \{\#\} \in \mathit{fst}\ \psi')) \longleftrightarrow \mathit{unsatisfiable}\ (\mathit{fst}\ \psi)$
using *assms resolution-completeness resolution-soundness* **by** *metis*

lemma *simplified-falsity*:
assumes *simp*: *simplified* ψ
and $\{\#\} \in \psi$
shows $\psi = \{\{\#\}\}$
proof (*rule ccontr*)
assume *H*: $\neg$ *?thesis*
then obtain χ **where** $\chi \in \psi$ **and** $\chi \neq \{\#\}$ **using** *assms*(*2*) **by** *blast*
then have $\{\#\} \subset\# \chi$ **by** (*simp add*: *mset-less-empty-nonempty*)
then have *simplify* ψ $(\psi - \{\chi\})$
using *simplify.subsumption*[*OF assms*(*2*) ‹$\{\#\} \subset\# \chi$› ‹$\chi \in \psi$›] **by** *blast*
then show *False* **using** *simp* **by** *blast*
qed

lemma *simplify-falsity-in-preserved*:
assumes *simplify* χs $\chi s'$
and $\{\#\} \in \chi s$
shows $\{\#\} \in \chi s'$
using *assms*
by *induction auto*

lemma *rtranclp-simplify-falsity-in-preserved*:
assumes *simplify*** χs $\chi s'$
and $\{\#\} \in \chi s$
shows $\{\#\} \in \chi s'$
using *assms*
by *induction* (*auto intro*: *simplify-falsity-in-preserved*)

lemma *resolution-falsity-get-falsity-alone*:
assumes *finite* (*fst* ψ)
shows $(\exists\,\psi'.\ (\mathit{resolution}^{**}\ \psi\ \psi' \wedge \{\#\} \in \mathit{fst}\ \psi')) \longleftrightarrow (\exists\, a\text{-}u\text{-}v.\ \mathit{resolution}^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))$
(**is** *?A* $\longleftrightarrow$ *?B*)
proof
assume *?B*
then show *?A* **by** *auto*
next
assume *?A*

then obtain χs *a-u-v* **where** χs: *resolution*** ψ (χs, *a-u-v*) **and** *F*: {#} ∈ χs **by** *auto*
{ assume *simplified* χs
then have *?B* **using** *simplified-falsity*[*OF - F*] χs **by** *blast*
}
moreover {
assume ¬ *simplified* χs
then obtain $\chi s'$ **where** *full1 simplify* χs $\chi s'$
by (*metis* χs *assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite*)
then have {#} ∈ $\chi s'$
unfolding *full1-def* **by** (*meson F rtranclp-simplify-falsity-in-preserved tranclp-into-rtranclp*)
then have *?B*
by (*metis* χs ⟨*full1 simplify* χs $\chi s'$⟩ *fst-conv full1-simp resolution-always-simplified rtranclp.rtrancl-into-rtrancl simplified-falsity*)
}
ultimately show *?B* **by** *blast*
qed

lemma *resolution-soundness-and-completeness'*:
fixes ψ :: *'v* ::*linorder state*
assumes
finite: *finite* (*fst* ψ)**and**
snd: *snd* ψ = {}
shows (∃ *a-u-v*. (*resolution*** ψ ({{#}}, *a-u-v*))) ⟷ *unsatisfiable* (*fst* ψ)
using *assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone*
by *metis*

end
theory *Prop-Superposition*
imports *Partial-Clausal-Logic ../lib/Herbrand-Interpretation*
begin

4.2 Superposition

no-notation *Herbrand-Interpretation.true-cls* (**infix** ⊨ *50*)
notation *Herbrand-Interpretation.true-cls* (**infix** ⊨*h 50*)

no-notation *Herbrand-Interpretation.true-clss* (**infix** ⊨*s 50*)
notation *Herbrand-Interpretation.true-clss* (**infix** ⊨*hs 50*)

lemma *herbrand-interp-iff-partial-interp-cls*:
S ⊨*h C* ⟷ {*Pos P*|*P*. *P*∈*S*} ∪ {*Neg P*|*P*. *P*∉*S*} ⊨ *C*
unfolding *Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def*
by *auto*

lemma *herbrand-consistent-interp*:
consistent-interp ({*Pos P*|*P*. *P*∈*S*} ∪ {*Neg P*|*P*. *P*∉*S*})
unfolding *consistent-interp-def* **by** *auto*

lemma *herbrand-total-over-set*:
total-over-set ({*Pos P*|*P*. *P*∈*S*} ∪ {*Neg P*|*P*. *P*∉*S*}) *T*
unfolding *total-over-set-def* **by** *auto*

lemma *herbrand-total-over-m*:
total-over-m ({*Pos P*|*P*. *P*∈*S*} ∪ {*Neg P*|*P*. *P*∉*S*}) *T*

unfolding *total-over-m-def* **by** (*auto simp add*: *herbrand-total-over-set*)

lemma *herbrand-interp-iff-partial-interp-clss*:
$S \models hs\ C \longleftrightarrow \{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\} \models s\ C$
unfolding *true-clss-def Ball-def herbrand-interp-iff-partial-interp-cls Partial-Clausal-Logic.true-clss-def* **by** *auto*

definition *clss-lt* :: *'a::wellorder clauses* ⇒ *'a clause* ⇒ *'a clauses* **where**
$clss\text{-}lt\ N\ C = \{D \in N.\ D \#{\subset}\#\ C\}$

notation (*latex* **output**)
clss-lt (-<^bsup>-<^esup>)

locale *selection* =
fixes *S* :: *'a clause* ⇒ *'a clause*
assumes
S-selects-subseteq: $\bigwedge C.\ S\ C \leq\#\ C$ **and**
S-selects-neg-lits: $\bigwedge C\ L.\ L \in\#\ S\ C \Longrightarrow is\text{-}neg\ L$

locale *ground-resolution-with-selection* =
selection S **for** *S* :: (*'a* :: *wellorder*) *clause* ⇒ *'a clause*
begin

context
fixes *N* :: *'a clause set*
begin

We do not create an equivalent of δ, but we directly defined N_C by inlining the definition.

function
production :: *'a clause* ⇒ *'a interp*
where
production C =
$\{A.\ C \in N \wedge C \neq \{\#\} \wedge Max\ (set\text{-}mset\ C) = Pos\ A \wedge count\ C\ (Pos\ A) \leq 1$
$\wedge \neg\ (\bigcup D \in \{D.\ D \#{\subset}\#\ C\}.\ production\ D) \models h\ C \wedge S\ C = \{\#\}\}$
by *auto*
termination by (*relation* $\{(D,\ C).\ D \#{\subset}\#\ C\}$) (*auto simp*: *wf-less-multiset*)

declare *production.simps*[*simp del*]

definition *interp* :: *'a clause* ⇒ *'a interp* **where**
$interp\ C = (\bigcup D \in \{D.\ D \#{\subset}\#\ C\}.\ production\ D)$

lemma *production-unfold*:
$production\ C = \{A.\ C \in N \wedge C \neq \{\#\} \wedge Max\ (set\text{-}mset\ C) = Pos\ A {\wedge}\ count\ C\ (Pos\ A) \leq 1 \wedge \neg\ interp\ C \models h\ C \wedge S\ C = \{\#\}\}$
unfolding *interp-def* **by** (*rule production.simps*)

abbreviation *productive* $A \equiv (production\ A \neq \{\})$

abbreviation *produces* :: *'a clause* ⇒ *'a* ⇒ *bool* **where**
$produces\ C\ A \equiv production\ C = \{A\}$

lemma *producesD*:
$produces\ C\ A \Longrightarrow C \in N \wedge C \neq \{\#\} \wedge Pos\ A = Max\ (set\text{-}mset\ C) \wedge count\ C\ (Pos\ A) \leq 1 \wedge$
$\neg\ interp\ C \models h\ C \wedge S\ C = \{\#\}$
unfolding *production-unfold* **by** *auto*

lemma *produces C A* $\Longrightarrow$ *Pos A* $\in\#$ *C*
by (*simp add*: *Max-in-lits producesD*)

lemma *interp′-def-in-set*:
interp C = ($\bigcup$ *D* $\in$ {*D* $\in$ *N*. *D* $\#\subset\#$ *C*}. *production D*)
unfolding *interp-def* **apply** *auto*
unfolding *production-unfold* **apply** *auto*
done

lemma *production-iff-produces*:
produces D A $\longleftrightarrow$ *A* $\in$ *production D*
unfolding *production-unfold* **by** *auto*

definition *Interp* :: *′a clause* $\Rightarrow$ *′a interp* **where**
Interp C = *interp C* $\cup$ *production C*

lemma
assumes *produces C P*
shows *Interp C* $\models$*h C*
unfolding *Interp-def assms* **using** *producesD*[*OF assms*]
by (*metis Max-in-lits Un-insert-right insertI1 pos-literal-in-imp-true-cls*)

definition *INTERP* :: *′a interp* **where**
INTERP = ($\bigcup$ *D* $\in$*N*. *production D*)

lemma *interp-subseteq-Interp*[*simp*]: *interp C* $\subseteq$ *Interp C*
unfolding *Interp-def* **by** *simp*

lemma *Interp-as-UNION*: *Interp C* = ($\bigcup$ *D* $\in$ {*D*. *D* $\#\subseteq\#$ *C*}. *production D*)
unfolding *Interp-def interp-def le-multiset-def* **by** *fast*

lemma *productive-not-empty*: *productive C* $\Longrightarrow$ *C* $\neq$ {#}
unfolding *production-unfold* **by** *auto*

lemma *productive-imp-produces-Max-literal*: *productive C* $\Longrightarrow$ *produces C* (*atm-of* (*Max* (*set-mset C*)))
unfolding *production-unfold* **by** (*auto simp del*: *atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom*: *productive C* $\Longrightarrow$ *produces C* (*Max* (*atms-of C*))
unfolding *atms-of-def Max-atm-of-set-mset-commute*[*OF productive-not-empty*]
by (*rule productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-literal*: *produces C A* $\Longrightarrow$ *A* = *atm-of* (*Max* (*set-mset C*))
by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-atom*: *produces C A* $\Longrightarrow$ *A* = *Max* (*atms-of C*)
by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-atom*)

lemma *produces-imp-Pos-in-lits*: *produces C A* $\Longrightarrow$ *Pos A* $\in\#$ *C*
by (*auto intro*: *Max-in-lits dest*!: *producesD*)

lemma *productive-in-N*: *productive C* $\Longrightarrow$ *C* $\in$ *N*
unfolding *production-unfold* **by** *auto*

lemma *produces-imp-atms-leq*: *produces C A* $\Longrightarrow$ *B* $\in$ *atms-of C* $\Longrightarrow$ *B* $\leq$ *A*

by (*metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom singleton-inject*)

lemma *produces-imp-neg-notin-lits*: *produces C A* $\Longrightarrow$ *Neg A* $\notin\#$ *C*
by (*rule pos-Max-imp-neg-notin*) (*auto dest*: *producesD*)

lemma *less-eq-imp-interp-subseteq-interp*: *C* $\#\subseteq\#$ *D* $\Longrightarrow$ *interp C* $\subseteq$ *interp D*
unfolding *interp-def* **by** *auto* (*metis multiset-order.order.strict-trans2*)

lemma *less-eq-imp-interp-subseteq-Interp*: *C* $\#\subseteq\#$ *D* $\Longrightarrow$ *interp C* $\subseteq$ *Interp D*
unfolding *Interp-def* **using** *less-eq-imp-interp-subseteq-interp* **by** *blast*

lemma *less-imp-production-subseteq-interp*: *C* $\#\subset\#$ *D* $\Longrightarrow$ *production C* $\subseteq$ *interp D*
unfolding *interp-def* **by** *fast*

lemma *less-eq-imp-production-subseteq-Interp*: *C* $\#\subseteq\#$ *D* $\Longrightarrow$ *production C* $\subseteq$ *Interp D*
unfolding *Interp-def* **using** *less-imp-production-subseteq-interp*
by (*metis multiset-order.le-imp-less-or-eq le-supI1 sup-ge2*)

lemma *less-imp-Interp-subseteq-interp*: *C* $\#\subset\#$ *D* $\Longrightarrow$ *Interp C* $\subseteq$ *interp D*
unfolding *Interp-def*
by (*auto simp*: *less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)

lemma *less-eq-imp-Interp-subseteq-Interp*: *C* $\#\subseteq\#$ *D* $\Longrightarrow$ *Interp C* $\subseteq$ *Interp D*
using *less-imp-Interp-subseteq-interp*
unfolding *Interp-def* **by** (*metis multiset-order.le-imp-less-or-eq le-supI2 subset-refl sup-commute*)

lemma *false-Interp-to-true-interp-imp-less-multiset*: *A* $\notin$ *Interp C* $\Longrightarrow$ *A* $\in$ *interp D* $\Longrightarrow$ *C* $\#\subset\#$ *D*
using *less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-interp-imp-less-multiset*: *A* $\notin$ *interp C* $\Longrightarrow$ *A* $\in$ *interp D* $\Longrightarrow$ *C* $\#\subset\#$ *D*
using *less-eq-imp-interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

lemma *false-Interp-to-true-Interp-imp-less-multiset*: *A* $\notin$ *Interp C* $\Longrightarrow$ *A* $\in$ *Interp D* $\Longrightarrow$ *C* $\#\subset\#$ *D*
using *less-eq-imp-Interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-Interp-imp-le-multiset*: *A* $\notin$ *interp C* $\Longrightarrow$ *A* $\in$ *Interp D* $\Longrightarrow$ *C* $\#\subseteq\#$ *D*
using *less-imp-Interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

lemma *interp-subseteq-INTERP*: *interp C* $\subseteq$ *INTERP*
unfolding *interp-def INTERP-def* **by** (*auto simp*: *production-unfold*)

lemma *production-subseteq-INTERP*: *production C* $\subseteq$ *INTERP*
unfolding *INTERP-def* **using** *production-unfold* **by** *blast*

lemma *Interp-subseteq-INTERP*: *Interp C* $\subseteq$ *INTERP*
unfolding *Interp-def* **by** (*auto intro*!: *interp-subseteq-INTERP production-subseteq-INTERP*)

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: *Neg A* $\in\#$ *C* **and** *d*: *produces D A*
shows *A* $\in$ *interp C*
proof −
from *d* **have** *Max* (*set-mset D*) = *Pos A*
using *production-unfold* **by** *blast*
then have *D* $\#\subset\#$ $\{\#$*Neg A*$\#\}$

by (*auto intro*: *Max-pos-neg-less-multiset*)
moreover have $\{\#Neg\ A\#\}\ \#\subseteq\#\ C$
by (*rule less-eq-imp-le-multiset*) (*rule mset-le-single*[*OF a-in-c*])
ultimately show *?thesis*
using *d* **by** (*blast dest*: *less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)
qed

lemma *neg-notin-Interp-not-produce*: $Neg\ A \in\#\ C \Longrightarrow A \notin Interp\ D \Longrightarrow C\ \#\subseteq\#\ D \Longrightarrow \neg\ produces\ D''\ A$
by (*auto dest*: *produces-imp-in-interp less-eq-imp-interp-subseteq-Interp*)

lemma *in-production-imp-produces*: $A \in production\ C \Longrightarrow produces\ C\ A$
by (*metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq'*)

lemma *not-produces-imp-notin-production*: $\neg\ produces\ C\ A \Longrightarrow A \notin production\ C$
by (*metis in-production-imp-produces*)

lemma *not-produces-imp-notin-interp*: $(\bigwedge D.\ \neg\ produces\ D\ A) \Longrightarrow A \notin interp\ C$
unfolding *interp-def* **by** (*fast intro*!: *in-production-imp-produces*)

The results below corresponds to Lemma 3.4.

Nitpicking: If $D = D'$ and D is productive, $I^D \subseteq I_{D'}$ does not hold.

lemma *true-Interp-imp-general*:
assumes
c-le-d: $C\ \#\subseteq\#\ D$ **and**
d-lt-d': $D\ \#\subset\#\ D'$ **and**
c-at-d: $Interp\ D \models h\ C$ **and**
subs: $interp\ D' \subseteq (\bigcup C \in CC.\ production\ C)$
shows $(\bigcup C \in CC.\ production\ C) \models h\ C$
proof (*cases* $\exists A.\ Pos\ A \in\#\ C \wedge A \in Interp\ D$)
case *True*
then obtain A **where** *a-in-c*: $Pos\ A \in\#\ C$ **and** *a-at-d*: $A \in Interp\ D$
by *blast*
from *a-at-d* **have** $A \in interp\ D'$
using *d-lt-d' less-imp-Interp-subseteq-interp* **by** *blast*
then show *?thesis*
using *subs a-in-c* **by** (*blast dest*: *contra-subsetD*)
next
case *False*
then obtain A **where** *a-in-c*: $Neg\ A \in\#\ C$ **and** $A \notin Interp\ D$
using *c-at-d* **unfolding** *true-cls-def* **by** *blast*
then have $\bigwedge D''.\ \neg\ produces\ D''\ A$
using *c-le-d neg-notin-Interp-not-produce* **by** *simp*
then show *?thesis*
using *a-in-c subs not-produces-imp-notin-production* **by** *auto*
qed

lemma *true-Interp-imp-interp*: $C\ \#\subseteq\#\ D \Longrightarrow D\ \#\subset\#\ D' \Longrightarrow Interp\ D \models h\ C \Longrightarrow interp\ D' \models h\ C$
using *interp-def true-Interp-imp-general* **by** *simp*

lemma *true-Interp-imp-Interp*: $C\ \#\subseteq\#\ D \Longrightarrow D\ \#\subset\#\ D' \Longrightarrow Interp\ D \models h\ C \Longrightarrow Interp\ D' \models h\ C$
using *Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general* **by** *simp*

lemma *true-Interp-imp-INTERP*: $C\ \#\subseteq\#\ D \Longrightarrow Interp\ D \models h\ C \Longrightarrow INTERP \models h\ C$
using *INTERP-def interp-subseteq-INTERP*

true-Interp-imp-general[*OF* - *less-multiset-right-total*]
by *simp*

lemma *true-interp-imp-general*:
assumes
c-le-d: $C \#\subseteq\# D$ **and**
d-lt-d': $D \#\subset\# D'$ **and**
c-at-d: *interp* $D \models h\ C$ **and**
subs: *interp* $D' \subseteq (\bigcup C \in CC.\ production\ C)$
shows $(\bigcup C \in CC.\ production\ C) \models h\ C$
proof (*cases* $\exists A.\ Pos\ A \in\# C \wedge A \in interp\ D$)
case *True*
then obtain A **where** *a-in-c*: $Pos\ A \in\# C$ **and** *a-at-d*: $A \in interp\ D$
by *blast*
from *a-at-d* **have** $A \in interp\ D'$
using *d-lt-d'* *less-eq-imp-interp-subseteq-interp*[*OF multiset-order.less-imp-le*] **by** *blast*
then show *?thesis*
using *subs a-in-c* **by** (*blast dest*: *contra-subsetD*)
next
case *False*
then obtain A **where** *a-in-c*: $Neg\ A \in\# C$ **and** $A \notin interp\ D$
using *c-at-d* **unfolding** *true-cls-def* **by** *blast*
then have $\bigwedge D''.\ \neg\ produces\ D''\ A$
using *c-le-d* **by** (*auto dest*: *produces-imp-in-interp less-eq-imp-interp-subseteq-interp*)
then show *?thesis*
using *a-in-c subs not-produces-imp-notin-production* **by** *auto*
qed

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *true-interp-imp-interp*: $C \#\subseteq\# D \Longrightarrow D \#\subset\# D' \Longrightarrow interp\ D \models h\ C \Longrightarrow interp\ D' \models h\ C$
using *interp-def true-interp-imp-general* **by** *simp*

lemma *true-interp-imp-Interp*: $C \#\subseteq\# D \Longrightarrow D \#\subset\# D' \Longrightarrow interp\ D \models h\ C \Longrightarrow Interp\ D' \models h\ C$
using *Interp-as-UNION interp-subseteq-Interp*[*of* D'] *true-interp-imp-general* **by** *simp*

lemma *true-interp-imp-INTERP*: $C \#\subseteq\# D \Longrightarrow interp\ D \models h\ C \Longrightarrow INTERP \models h\ C$
using *INTERP-def interp-subseteq-INTERP*
true-interp-imp-general[*OF* - *less-multiset-right-total*]
by *simp*

lemma *productive-imp-false-interp*: $productive\ C \Longrightarrow \neg\ interp\ C \models h\ C$
unfolding *production-unfold* **by** *auto*

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:
assumes D: $\{\#Pos\ P,\ Pos\ P\#\} \#\subset\# C$
shows $\neg produces\ C\ P$
proof −
let $?D = \{\#Pos\ P,\ Pos\ P\#\}$
note $D' = D$[*unfolded less-multiset*$_{HO}$]
consider
(P) *count* $C\ (Pos\ P) \geq 2$
| (Q) Q **where** $Q > Pos\ P$ **and** $Q \in\# C$

using *HOL.spec*[*OF HOL.conjunct2*[*OF D'*], *of Pos P*] **by** (*auto split*: *if-split-asm*)
then show *?thesis*
proof *cases*
case *Q*
have $Q \in$ *set-mset C*
using *Q*(*2*) **by** (*auto split*: *if-split-asm*)
then have *Max* (*set-mset C*) > *Pos P*
using *Q*(*1*) *Max-gr-iff* **by** *blast*
then show *?thesis*
unfolding *production-unfold* **by** *auto*
next
case *P*
then show *?thesis*
unfolding *production-unfold* **by** *auto*
qed
qed

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma
assumes *D*: *C*+{#*Neg P*#} #⊂# *D*
shows *production D* ≠ {*P*}
proof −
note $D' = D$[*unfolded less-multiset*$_{HO}$]
consider
(*P*) *Neg P* ∈# *D*
| (*Q*) *Q* **where** *Q* > *Neg P* **and** *count D Q* > *count* (*C* + {#*Neg P*#}) *Q*
using *HOL.spec*[*OF HOL.conjunct2*[*OF D'*], *of Neg P*] *count-greater-zero-iff* **by** *fastforce*
then show *?thesis*
proof *cases*
case *Q*
have $Q \in$ *set-mset D*
using *Q*(*2*) *gr-implies-not0* **by** *fastforce*
then have *Max* (*set-mset D*) > *Neg P*
using *Q*(*1*) *Max-gr-iff* **by** *blast*
then have *Max* (*set-mset D*) > *Pos P*
using *less-trans*[*of Pos P Neg P Max* (*set-mset D*)] **by** *auto*
then show *?thesis*
unfolding *production-unfold* **by** *auto*
next
case *P*
then have *Max* (*set-mset D*) > *Pos P*
by (*meson Max-ge finite-set-mset le-less-trans linorder-not-le pos-less-neg*)
then show *?thesis*
unfolding *production-unfold* **by** *auto*
qed
qed

lemma *in-interp-is-produced*:
assumes *P* ∈ *INTERP*
shows ∃ *D*. *D* +{#*Pos P*#} ∈ *N* ∧ *produces* (*D* +{#*Pos P*#}) *P*
using *assms* **unfolding** *INTERP-def UN-iff production-iff-produces Ball-def*
by (*metis ground-resolution-with-selection.produces-imp-Pos-in-lits insert-DiffM2 ground-resolution-with-selection-axioms not-produces-imp-notin-production*)

end

end

abbreviation *MMax M* ≡ *Max* (*set-mset M*)

4.2.1 We can now define the rules of the calculus

inductive *superposition-rules* :: *'a clause* ⇒ *'a clause* ⇒ *'a clause* ⇒ *bool* **where**
factoring: *superposition-rules* (*C* + {#*Pos P*#} + {#*Pos P*#}) *B* (*C* + {#*Pos P*#}) |
superposition-l: *superposition-rules* (C_1 + {#*Pos P*#}) (C_2 + {#*Neg P*#}) (C_1+ C_2)

inductive *superposition* :: *'a clauses* ⇒ *'a clauses* ⇒ *bool* **where**
superposition: *A* ∈ *N* ⟹ *B* ∈ *N* ⟹ *superposition-rules A B C*
⟹ *superposition N* (*N* ∪ {*C*})

definition *abstract-red* :: *'a::wellorder clause* ⇒ *'a clauses* ⇒ *bool* **where**
abstract-red C N = (*clss-lt N C* ⊨*p C*)

lemma *less-multiset*[*iff*]: *M* < *N* ⟷ *M* #⊂# *N*
unfolding *less-multiset-def* **by** *auto*

lemma *less-eq-multiset*[*iff*]: *M* ≤ *N* ⟷ *M* #⊆# *N*
unfolding *less-eq-multiset-def* **by** *auto*

lemma *herbrand-true-clss-true-clss-cls-herbrand-true-clss*:
assumes
AB: *A* ⊨*hs B* **and**
BC: *B* ⊨*p C*
shows *A* ⊨*h C*
proof −
let *?I* = {*Pos P* |*P*. *P* ∈ *A*} ∪ {*Neg P* |*P*. *P* ∉ *A*}
have *B*: *?I* ⊨*s B* **using** *AB*
by (*auto simp add*: *herbrand-interp-iff-partial-interp-clss*)

have *IH*: ⋀*I*. *total-over-set I* (*atms-of C*) ⟹ *total-over-m I B* ⟹ *consistent-interp I*
⟹ *I* ⊨*s B* ⟹ *I* ⊨ *C* **using** *BC*
by (*auto simp add*: *true-clss-cls-def*)
show *?thesis*
unfolding *herbrand-interp-iff-partial-interp-cls*
by (*auto intro*: *IH*[*of ?I*] *simp add*: *herbrand-total-over-set herbrand-total-over-m*
herbrand-consistent-interp B)
qed

lemma *abstract-red-subset-mset-abstract-red*:
assumes
abstr: *abstract-red C N* **and**
c-lt-d: *C* ⊆# *D*
shows *abstract-red D N*
proof −
have {*D* ∈ *N*. *D* #⊂# *C*} ⊆ {*D'* ∈ *N*. *D'* #⊂# *D*}
using *c-lt-d less-eq-imp-le-multiset* **by** *fastforce*
then show *?thesis*
using *abstr* **unfolding** *abstract-red-def clss-lt-def*
by (*metis* (*no-types*, *lifting*) *c-lt-d subset-mset.diff-add true-clss-cls-mono-r'*
true-clss-cls-subset)
qed

lemma *true-clss-cls-extended*:
  assumes
    $A \models p\ B$ **and**
    tot: *total-over-m I A* **and**
    cons: *consistent-interp I* **and**
    I-A: $I \models s\ A$
  shows $I \models B$
proof −
  let $?I = I \cup \{Pos\ P|P.\ P \in atms\text{-}of\ B \wedge P \notin atms\text{-}of\text{-}s\ I\}$
  have *consistent-interp ?I*
    using *cons* **unfolding** *consistent-interp-def atms-of-s-def atms-of-def*
      apply (*auto 1 5 simp add*: *image-iff*)
    by (*metis atm-of-uminus literal.sel*(*1*))
  moreover have *total-over-m ?I* $(A \cup \{B\})$
    proof −
      obtain $aa :: 'a\ set \Rightarrow 'a\ literal\ set \Rightarrow 'a$ **where**
        $f2: \forall x0\ x1.\ (\exists v2.\ v2 \in x0 \wedge Pos\ v2 \notin x1 \wedge Neg\ v2 \notin x1)$
          $\longleftrightarrow (aa\ x0\ x1 \in x0 \wedge Pos\ (aa\ x0\ x1) \notin x1 \wedge Neg\ (aa\ x0\ x1) \notin x1)$
        by *moura*
      have $\forall a.\ a \notin atms\text{-}of\text{-}ms\ A \vee Pos\ a \in I \vee Neg\ a \in I$
        using *tot* **by** (*simp add*: *total-over-m-def total-over-set-def*)
      then have $aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})\ (I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s$
$I\})$
        $\notin atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\} \vee Pos\ (aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
          $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})) \in I$
            $\cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\}$
          $\vee Neg\ (aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
            $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})) \in I$
            $\cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\}$
        by *auto*
      then have *total-over-set* $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})$
        $(atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
        using *f2* **by** (*meson total-over-set-def*)
      then show *?thesis*
        by (*simp add*: *total-over-m-def*)
    qed
  moreover have $?I \models s\ A$
    using *I-A* **by** *auto*
  ultimately have $?I \models B$
    using ‹$A \models pB$› **unfolding** *true-clss-cls-def* **by** *auto*
  then show *?thesis*
oops
lemma
  assumes
    CP: $\neg$ *clss-lt* $N\ (\{\#C\#\} + \{\#E\#\}) \models p\ \{\#C\#\} + \{\#Neg\ P\#\}$ **and**
    clss-lt $N\ (\{\#C\#\} + \{\#E\#\}) \models p\ \{\#E\#\} + \{\#Pos\ P\#\} \vee$ *clss-lt* $N\ (\{\#C\#\} + \{\#E\#\}) \models p$
$\{\#C\#\} + \{\#Neg\ P\#\}$
  shows *clss-lt* $N\ (\{\#C\#\} + \{\#E\#\}) \models p\ \{\#E\#\} + \{\#Pos\ P\#\}$

oops

locale *ground-ordered-resolution-with-redundancy* =
  ground-resolution-with-selection +
  fixes *redundant* :: $'a::wellorder\ clause \Rightarrow 'a\ clauses \Rightarrow bool$
  assumes

redundant-iff-abstract: *redundant A N* ⟷ *abstract-red A N*
begin
definition *saturated* :: *'a clauses* ⇒ *bool* **where**
saturated N ⟷ (∀ *A B C*. *A* ∈ *N* ⟶ *B* ∈ *N* ⟶ ¬*redundant A N* ⟶ ¬*redundant B N*
⟶ *superposition-rules A B C* ⟶ *redundant C N* ∨ *C* ∈ *N*)

lemma
assumes
saturated: *saturated N* **and**
finite: *finite N* **and**
empty: {#} ∉ *N*
shows *INTERP N* ⊨*hs N*
proof (*rule ccontr*)
let $?N_{\mathcal{I}}$ = *INTERP N*
assume ¬ *?thesis*
then have *not-empty*: {*E*∈*N*. ¬$?N_{\mathcal{I}}$ ⊨*h E*} ≠ {}
unfolding *true-clss-def Ball-def* **by** *auto*
def *D* ≡ *Min* {*E*∈*N*. ¬$?N_{\mathcal{I}}$ ⊨*h E*}
have [*simp*]: *D* ∈ *N*
unfolding *D-def*
by (*metis* (*mono-tags*, *lifting*) *Min-in not-empty finite mem-Collect-eq rev-finite-subset subsetI*)
have *not-d-interp*: ¬$?N_{\mathcal{I}}$ ⊨*h D*
unfolding *D-def*
by (*metis* (*mono-tags*, *lifting*) *Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI*)
have *cls-not-D*: ⋀*E*. *E* ∈ *N* ⟹ *E* ≠ *D* ⟹ ¬$?N_{\mathcal{I}}$ ⊨*h E* ⟹ *D* ≤ *E*
using *finite D-def* **by** (*auto simp del*: *less-eq-multiset*)
obtain *C L* **where** *D*: *D* = *C* + {#*L*#} **and** *LSD*: *L* ∈# *S D* ∨ (*S D* = {#} ∧ *Max* (*set-mset D*) = *L*)
proof (*cases S D* = {#})
case *False*
then obtain *L* **where** *L* ∈# *S D*
using *Max-in-lits* **by** *blast*
moreover
then have *L* ∈# *D*
using *S-selects-subseteq*[*of D*] **by** *auto*
then have *D* = (*D* − {#*L*#}) + {#*L*#}
by *auto*
ultimately show *?thesis* **using** *that* **by** *blast*
next
let *?L* = *MMax D*
case *True*
moreover
have *?L* ∈# *D*
by (*metis* (*no-types*, *lifting*) *Max-in-lits* ‹*D* ∈ *N*› *empty*)
then have *D* = (*D* − {#*?L*#}) + {#*?L*#}
by *auto*
ultimately show *?thesis* **using** *that* **by** *blast*
qed
have *red*: ¬*redundant D N*
proof (*rule ccontr*)
assume *red*[*simplified*]: ~~*redundant D N*
have ∀ *E* < *D*. *E* ∈ *N* ⟶ $?N_{\mathcal{I}}$ ⊨*h E*
using *cls-not-D not-le* **by** *fastforce*
then have $?N_{\mathcal{I}}$ ⊨*hs clss-lt N D*
unfolding *clss-lt-def true-clss-def Ball-def* **by** *blast*
then show *False*

using *red not-d-interp* **unfolding** *abstract-red-def redundant-iff-abstract*
using *herbrand-true-clss-true-clss-cls-herbrand-true-clss* **by** *fast*
qed

consider
(*L*) *P* **where** *L* = *Pos P* **and** *S D* = {#} **and** *Max* (*set-mset D*) = *Pos P*
| (*Lneg*) *P* **where** *L* = *Neg P*
using *LSD S-selects-neg-lits*[*of L D*] **by** (*cases L*) *auto*
then show *False*
proof *cases*
case *L* **note** *P* = *this*(*1*) **and** *S* = *this*(*2*) **and** *max* = *this*(*3*)
have *count D L* > *1*
proof (*rule ccontr*)
assume ~ *?thesis*
then have *count*: *count D L* = *1*
unfolding *D* **by** (*auto simp*: *not-in-iff*)
have ¬*?*$N_{\mathcal{I}}$⊨*h D*
using *not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms*
by *blast*
then have *produces N D P*
using *not-empty empty finite* ‹*D* ∈ *N*› *count L*
true-interp-imp-INTERP **unfolding** *production-iff-produces* **unfolding** *production-unfold*
by (*auto simp add*: *max not-empty*)
then have *INTERP N* ⊨*h D*
unfolding *D*
by (*metis pos-literal-in-imp-true-cls produces-imp-Pos-in-lits*
production-subseteq-INTERP singletonI subsetCE)
then show *False*
using *not-d-interp* **by** *blast*
qed
then have *Pos P* ∈# *C*
by (*simp add*: *P D*)
then obtain *C′* **where** *C′*:*D* = *C′* + {#*Pos P*#} + {#*Pos P*#}
unfolding *D* **by** (*metis* (*full-types*) *P insert-DiffM2*)
have *sup*: *superposition-rules D D* (*D* − {#*L*#})
unfolding *C′ L* **by** (*auto simp add*: *superposition-rules.simps*)
have *C′* + {#*Pos P*#} #⊂# *C′* + {#*Pos P*#} + {#*Pos P*#}
by *auto*
moreover have ¬*?*$N_{\mathcal{I}}$ ⊨*h* (*D* − {#*L*#})
using *not-d-interp* **unfolding** *C′ L* **by** *auto*
ultimately have *C′* + {#*Pos P*#} ∉ *N*
by (*metis* (*no-types*, *lifting*) *C′ P add-diff-cancel-right′ cls-not-D less-multiset*
multi-self-add-other-not-self not-le)
have *D* − {#*L*#} #⊂# *D*
unfolding *C′ L* **by** *auto*
have *c′-p-p*: *C′* + {#*Pos P*#} + {#*Pos P*#} − {#*Pos P*#} = *C′* + {#*Pos P*#}
by *auto*
have *redundant* (*C′* + {#*Pos P*#}) *N*
using *saturated red sup* ‹*D* ∈ *N*›‹*C′* + {#*Pos P*#} ∉ *N*› **unfolding** *saturated-def C′ L c′-p-p*
by *blast*
moreover have *C′* + {#*Pos P*#} ⊆# *C′* + {#*Pos P*#} + {#*Pos P*#}
by *auto*
ultimately show *False*
using *red* **unfolding** *C′ redundant-iff-abstract* **by** (*blast dest*:
abstract-red-subset-mset-abstract-red)
next

case *Lneg* **note** $L = this(1)$
have $P \in ?N_{\mathcal{I}}$
  using *not-d-interp* **unfolding** *D true-cls-def L* **by** (*auto split*: *if-split-asm*)
then obtain E **where**
  DPN: $E + \{\#Pos\ P\#\} \in N$ **and**
  prod: *production* N $(E + \{\#Pos\ P\#\}) = \{P\}$
  using *in-interp-is-produced* **by** *blast*
have *sup-EC*: *superposition-rules* $(E + \{\#Pos\ P\#\})$ $(C + \{\#Neg\ P\#\})$ $(E + C)$
  using *superposition-l* **by** *fast*
then have *superposition* N $(N \cup \{E+C\})$
  using *DPN* ‹$D \in N$› **unfolding** *D L* **by** (*auto simp add*: *superposition.simps*)
have
  PMax: $Pos\ P = MMax\ (E + \{\#Pos\ P\#\})$ **and**
  count $(E + \{\#Pos\ P\#\})$ $(Pos\ P) \leq 1$ **and**
  $S\ (E + \{\#Pos\ P\#\}) = \{\#\}$ **and**
  $\neg$*interp* N $(E + \{\#Pos\ P\#\}) \models h\ E + \{\#Pos\ P\#\}$
  using *prod* **unfolding** *production-unfold* **by** *auto*
have $Neg\ P \notin\#\ E$
  using *prod produces-imp-neg-notin-lits* **by** *force*
then have $\bigwedge y.\ y \in\#\ (E + \{\#Pos\ P\#\})$
  $\Longrightarrow$ *count* $(E + \{\#Pos\ P\#\})$ $(Neg\ P) <$ *count* $(C + \{\#Neg\ P\#\})$ $(Neg\ P)$
  using *count-greater-zero-iff* **by** *fastforce*
moreover have $\bigwedge y.\ y \in\#\ (E + \{\#Pos\ P\#\}) \Longrightarrow y < Neg\ P$
  using *PMax* **by** (*metis DPN Max-less-iff empty finite-set-mset pos-less-neg*
    set-mset-eq-empty-iff)
moreover have $E + \{\#Pos\ P\#\} \neq C + \{\#Neg\ P\#\}$
  using *prod produces-imp-neg-notin-lits* **by** *force*
ultimately have $E + \{\#Pos\ P\#\}\ \#\subset\#\ C + \{\#Neg\ P\#\}$
  unfolding *less-multiset*$_{HO}$ **by** (*metis count-greater-zero-iff less-iff-Suc-add zero-less-Suc*)
have *ce-lt-d*: $C + E\ \#\subset\#\ D$
 unfolding *D L* **by** (*simp add*: ‹$\bigwedge y.\ y \in\#\ E + \{\#Pos\ P\#\} \Longrightarrow y < Neg\ P$› *ex-gt-imp-less-multiset*)
have $?N_{\mathcal{I}} \models h\ E + \{\#Pos\ P\#\}$
  using ‹$P \in ?N_{\mathcal{I}}$› **by** *blast*
have $?N_{\mathcal{I}} \models h\ C{+}E \vee C{+}E \notin N$
  using *ce-lt-d cls-not-D* **unfolding** *D-def* **by** *fastforce*
have $Pos\ P \notin\#\ C{+}E$
  using *D* ‹$P \in$ *ground-resolution-with-selection.INTERP* $S\ N$›
    ‹*count* $(E + \{\#Pos\ P\#\})$ $(Pos\ P) \leq 1$› *multi-member-skip not-d-interp*
    by (*auto simp*: *not-in-iff*)
then have $\bigwedge y.\ y \in\#\ C{+}E$
  $\Longrightarrow$ *count* $(C{+}E)$ $(Pos\ P) <$ *count* $(E + \{\#Pos\ P\#\})$ $(Pos\ P)$
  using *set-mset-def* **by** *fastforce*

have $\neg$*redundant* $(C + E)$ N
  proof (*rule ccontr*)
    assume *red'*[*simplified*]: $\neg$ *?thesis*
    have *abs*: *clss-lt* N $(C + E) \models p\ C + E$
      using *redundant-iff-abstract red'* **unfolding** *abstract-red-def* **by** *auto*
    have *clss-lt* N $(C + E) \models p\ E + \{\#Pos\ P\#\} \vee$ *clss-lt* N $(C + E) \models p\ C + \{\#Neg\ P\#\}$
      proof *clarify*
        assume *CP*: $\neg$ *clss-lt* N $(C + E) \models p\ C + \{\#Neg\ P\#\}$
        { fix I
          assume
            total-over-m I (*clss-lt* N $(C + E) \cup \{E + \{\#Pos\ P\#\}\}$) **and**
            consistent-interp I **and**
            $I \models s$ *clss-lt* N $(C + E)$

```
            then have I ⊨ C + E
              using  abs sorry
            moreover have ¬ I ⊨ C + {#Neg P#}
              using CP unfolding true-clss-cls-def
              sorry
            ultimately have  I ⊨ E + {#Pos P#} by auto
          }
          then show clss-lt N (C + E) ⊨p E + {#Pos P#}
            unfolding true-clss-cls-def by auto
        qed
      moreover have clss-lt N (C + E) ⊆ clss-lt N (C + {#Neg P#})
        using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
      ultimately have redundant (C + {#Neg P#}) N ∨ clss-lt N (C + E) ⊨p E + {#Pos P#}
        unfolding redundant-iff-abstract abstract-red-def using true-clss-cls-subset by blast
      show False sorry
    qed
  moreover have ¬ redundant (E + {#Pos P#}) N
    sorry
  ultimately have CEN: C + E ∈ N
    using ‹D∈N› ‹E + {#Pos P#}∈N› saturated sup-EC red unfolding saturated-def D L
    by (metis union-commute)
  have CED: C + E ≠ D
    using D ce-lt-d by auto
  have interp: ¬ INTERP N ⊨h C + E
  sorry
  show False
     using cls-not-D[OF CEN CED interp] ce-lt-d unfolding INTERP-def less-eq-multiset-def by
auto
  qed
qed

end

lemma tautology-is-redundant:
  assumes tautology C
  shows abstract-red C N
  using assms unfolding abstract-red-def true-clss-cls-def tautology-def by auto

lemma subsumed-is-redundant:
  assumes AB: A ⊂# B
  and AN: A ∈ N
  shows abstract-red B N
proof −
  have A ∈ clss-lt N B using AN AB unfolding clss-lt-def
    by (auto dest: less-eq-imp-le-multiset simp add: multiset-order.dual-order.order-iff-strict)
  then show ?thesis
    using AB unfolding abstract-red-def true-clss-cls-def Partial-Clausal-Logic.true-clss-def
    by blast
qed

inductive redundant :: 'a clause ⇒ 'a clauses ⇒ bool where
subsumption: A ∈ N ⟹ A ⊂# B ⟹ redundant B N

lemma redundant-is-redundancy-criterion:
  fixes A :: 'a :: wellorder clause and N :: 'a :: wellorder clauses
  assumes redundant A N
```

shows *abstract-red A N*
using *assms*
proof (*induction rule*: *redundant.induct*)
case (*subsumption A B N*)
then show *?case*
using *subsumed-is-redundant*[*of A N B*] **unfolding** *abstract-red-def clss-lt-def* **by** *auto*
qed

lemma *redundant-mono*:
redundant A N $\Longrightarrow$ *A* $\subseteq$# *B* $\Longrightarrow$ *redundant B N*
apply (*induction rule*: *redundant.induct*)
by (*meson subset-mset.less-le-trans subsumption*)

locale *truc* =
selection S **for** *S* :: *nat clause* $\Rightarrow$ *nat clause*
begin

end

end

4.3 Partial Clausal Logic

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

theory *Partial-Annotated-Clausal-Logic*
imports *Partial-Clausal-Logic*

begin

4.3.1 Decided Literals

Definition

datatype (*'v*, *'mark*) *ann-lit* =
is-decided: *Decided* (*lit-of*: *'v literal*) |
is-proped: *Propagated* (*lit-of*: *'v literal*) (*mark-of*: *'mark*)

lemma *ann-lit-list-induct*[*case-names Nil Decided Propagated*]:
assumes *P* [] **and**
$\bigwedge$*L xs. P xs* $\Longrightarrow$ *P* (*Decided L* # *xs*) **and**
$\bigwedge$*L m xs. P xs* $\Longrightarrow$ *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 $\Longrightarrow$ ($\bigwedge$*K. L* = *Decided K* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by (*cases L*) *auto*

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

definition *lits-of* :: (*'a*, *'b*) *ann-lit set* $\Rightarrow$ *'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*)

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 clauses* ⇒ *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*]:
$I \models as\ \{\}$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-single-true-annot*[*iff*]:
$I \models as\ \{C\} \longleftrightarrow I \models a\ C$
unfolding *true-annots-def* **by** *auto*

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

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

lemma *true-annots-union*[*iff*]:
$M \models as\ A \cup B \longleftrightarrow (M \models as\ A \wedge M \models as\ B)$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-insert*[*iff*]:
$M \models as\ insert\ a\ A \longleftrightarrow (M \models a\ a \wedge M \models as\ A)$
unfolding *true-annots-def* **by** *auto*

Link between $\models as$ and $\models s$:

lemma *true-annots-true-cls*:
$I \models as\ CC \longleftrightarrow lits\text{-}of\text{-}l\ I \models s\ CC$
unfolding *true-annots-def Ball-def true-annot-def true-clss-def* **by** *auto*

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

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

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

lemma *true-annot-true-clss-cls*:
$MLs \models a\ \psi \Longrightarrow set\ (map\ unmark\ MLs) \models p\ \psi$
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*:
$MLs \models as\ \psi \Longrightarrow set\ (map\ unmark\ MLs) \models ps\ \psi$
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* $A \models ps\ \Psi$
  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*, *'m*) *ann-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*, *'m*) *ann-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 ann-lit.exhaust-sel* **by** *metis*

lemma *literal-is-lit-of-decided*:
assumes $L = \textit{lit-of } x$
shows $(x = \textit{Decided } L) \lor (\exists l'.\ x = \textit{Propagated } L\ l')$
using *assms* **by** (*cases x*) *auto*

lemma *true-annot-iff-decided-or-true-lit*:
$\textit{defined-lit } I\ L \longleftrightarrow (\textit{lits-of-l } I \models l\ L \lor \textit{lits-of-l } I \models l\ -L)$
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*:
$\textit{consistent-interp } (\textit{lits-of-l } I) \Longrightarrow I \models as\ N \Longrightarrow \textit{satisfiable } N$
by (*simp add*: *true-annots-true-cls*)

lemma *defined-lit-map*:
$\textit{defined-lit } Ls\ L \longleftrightarrow \textit{atm-of } L \in (\lambda l.\ \textit{atm-of } (\textit{lit-of } l))\ `\ \textit{set } Ls$
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*]:
$\textit{defined-lit } I\ (-L) \longleftrightarrow \textit{defined-lit } I\ L$
unfolding *defined-lit-def* **by** *auto*

lemma *Decided-Propagated-in-iff-in-lits-of-l*:
$\textit{defined-lit } I\ L \longleftrightarrow (L \in \textit{lits-of-l } I \lor -L \in \textit{lits-of-l } I)$
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*]:
$\neg\textit{defined-lit } []\ L$
unfolding *defined-lit-def* **by** *simp*

4.3.2 Backtracking

fun *backtrack-split* :: $('v, 'm)\ \textit{ann-lits}$
$\Rightarrow ('v, 'm)\ \textit{ann-lits} \times ('v, 'm)\ \textit{ann-lits}$ **where**
$\textit{backtrack-split } [] = ([], [])\ |$
$\textit{backtrack-split } (\textit{Propagated } L\ P\ \#\ \textit{mlits}) = \textit{apfst } ((\textit{op } \#)\ (\textit{Propagated } L\ P))\ (\textit{backtrack-split mlits})\ |$
$\textit{backtrack-split } (\textit{Decided } L\ \#\ \textit{mlits}) = ([], \textit{Decided } L\ \#\ \textit{mlits})$

lemma *backtrack-split-fst-not-decided*: $a \in \textit{set } (\textit{fst } (\textit{backtrack-split } l)) \Longrightarrow \neg\textit{is-decided } a$
by (*induct l rule*: *ann-lit-list-induct*) *auto*

lemma *backtrack-split-snd-hd-decided*:
$\textit{snd } (\textit{backtrack-split } l) \neq [] \Longrightarrow \textit{is-decided } (\textit{hd } (\textit{snd } (\textit{backtrack-split } l)))$
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'', []) $\Longrightarrow \forall l \in$ *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*:
  $\exists l \in$ *set M*. *is-decided l* $\Longrightarrow \exists M'\ L'\ M''$. *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*

4.3.3 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*, '*m*) *ann-lits*
  ⇒ (('*a*, '*m*) *ann-lits* × ('*a*, '*m*) *ann-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* ((*op* #) (*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 = [] $\longleftrightarrow$ *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* $\longleftrightarrow$ *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*) $\Longrightarrow$ *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'*) $\longleftrightarrow$ *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*)] $\Longrightarrow$ *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* = [] $\vee$ (*length a* = *1* $\wedge$ *is-decided* (*hd a*) $\wedge$ *hd a* $\in$ *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* = [] $\vee$ (*is-decided* (*hd a*) $\wedge$ *hd a* $\in$ *set M*)
using *assms* **apply** (*induct M arbitrary*: *a b rule*: *ann-lit-list-induct*)
apply *auto*[*2*]
by (*metis UnCI backtrack-split-snd-hd-decided get-all-ann-decomposition-backtrack-split get-all-ann-decomposition-decomp hd-in-set list.sel*(*1*) *set-append snd-conv*)

lemma *get-all-ann-decomposition-snd-not-decided*:
assumes (*a*, *b*) $\in$ *set* (*get-all-ann-decomposition M*)
and *L* $\in$ *set b*
shows $\neg$*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* $\in$ *set* (*tl* (*get-all-ann-decomposition M1*))
shows *x* $\in$ *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 $(x, y) \in set$ (*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*:
$(a, b) \in set$ (*get-all-ann-decomposition M'*) $\Longrightarrow$
$\exists b'. (a, b' @ b) \in set$ (*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 $(a, b) \in set$ (*get-all-ann-decomposition M*)
shows $a = [] \vee$ (*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 $\forall l \in set\ M'. \neg$*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 $\forall l \in set\ M'. \neg$*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*)) $\neq []$
and $\forall l \in set\ M'. \neg$*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 $\forall l \in$ *set* M'. $\neg$*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 = $M0'$ @ $M0$ $\wedge$ *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 *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 = $M0'$ @ $M0$ $\wedge$ *is-decided* (*hd* M) **using** g a **by** *auto*
}
moreover {
fix L P
assume a: a = *Propagated* L P
have *tl* M = $M0'$ @ $M0$ $\wedge$ *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) $\in$ *set* (*get-all-ann-decomposition* M)
shows $\exists$ 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) $\in$ *set* (*get-all-ann-decomposition* M)
shows *set* b $\subseteq$ *set* M **and** *set* a $\subseteq$ *set* M
using *assms* *get-all-ann-decomposition-exists-prepend* **by** *fastforce*+

lemma *get-all-ann-decomposition-exists-prepend'*:
assumes (a, b) $\in$ *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) $\in$ *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*:
∃ *M1 M2*. (*Decided K # M1*, *M2*) ∈ *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*

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 literal multiset 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 literal multiset 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 ΨN 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 $N \cup \{$*unmark L* |*L. is-decided L* $\wedge$ *L* $\in$ *set M*$\}$ $\models$*ps unmark-l M*
(**is** *?I* $\models$*ps ?A*)
proof −
have *?I* $\models$*ps unmark-s* $\{L$ |*L. is-decided L* $\wedge$ *L* $\in$ *set M*$\}$
by (*auto intro*: *all-in-true-clss-clss*)
moreover have *?I* $\models$*ps unmark* ‘ $\bigcup$(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*))
using *all-decomposition-implies-trail-is-implied assms* **by** *blast*
ultimately have $N \cup \{$*unmark m* |*m. is-decided m* $\wedge$ *m* $\in$ *set M*$\}$
$\models$*ps unmark* ‘ $\bigcup$(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition M*))
$\cup$ *unmark* ‘ $\{m$ |*m. is-decided m* $\wedge$ *m* $\in$ *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 $\Longrightarrow$ *all-decomposition-implies* (*insert C N*) *M*
unfolding *all-decomposition-implies-def* **by** *auto*

4.3.4 Negation of Clauses

We define the negation of a *'a Partial-Clausal-Logic.clause*: it converts it from the a single clause to a set of clauses, wherein each clause is a single negated literal.

definition *CNot* :: *'v clause* $\Rightarrow$ *'v clauses* **where**
CNot ψ = $\{$ $\{$#−*L*#$\}$ | *L. L* $\in$# ψ $\}$

lemma *in-CNot-uminus*[*iff*]:
shows $\{$#*L*#$\}$ $\in$ *CNot* ψ $\longleftrightarrow$ −*L* $\in$# ψ
unfolding *CNot-def* **by** *force*

lemma
shows
CNot-singleton[*simp*]: *CNot* $\{$#*L*#$\}$ = $\{\{$#−*L*#$\}\}$ **and**
CNot-empty[*simp*]: *CNot* $\{$#$\}$ = $\{\}$ **and**
CNot-plus[*simp*]: *CNot* (*A* + *B*) = *CNot A* $\cup$ *CNot B*
unfolding *CNot-def* **by** *auto*

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

lemma *in-CNot-implies-uminus*:
assumes *L* $\in$# *D* **and** *M* $\models$*as CNot D*
shows *M* $\models$*a* $\{$#−*L*#$\}$ **and** −*L* $\in$ *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*]:
$(\forall x \in CNot\ D.\ P\ x) \longleftrightarrow (\forall L \in\# D.\ P\ \{\#-L\#\})$
unfolding *CNot-def* **by** *auto*

lemma *consistent-CNot-not*:
assumes *consistent-interp I*
shows $I \models s\ CNot\ \varphi \Longrightarrow \neg I \models \varphi$
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* $\{\varphi\}$ **and** $\neg I \models \varphi$
shows $I \models s\ CNot\ \varphi$
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* $\{\varphi\}$ **and** $\neg I \models s\ CNot\ \varphi$
shows $I \models \varphi$
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*:
$C \in D \Longrightarrow D \models ps\ CNot\ C \Longrightarrow D \models p\ \{\#\}$
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 $M \models as\ CNot\ T$ **and** *a1*: $L \in\# T$
shows *atm-of* $L \in$ *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 $M \models as\ CNot\ T$ **and** *a1*: $-L \in\# T$
shows *atm-of* $L \in$ *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 $\{\{\#L\#\}\} \cup B \models p\ \{\#\}$
shows $B \models ps\ CNot\ \{\#L\#\}$
unfolding *true-clss-clss-def true-clss-cls-def*
proof (*intro allI impI*)
fix *I*
assume
tot: *total-over-m I* $(B \cup CNot\ \{\#L\#\})$ **and**
cons: *consistent-interp I* **and**
I: $I \models s\ B$
have *total-over-m I* $(\{\{\#L\#\}\} \cup B)$ **using** *tot* **by** *auto*
then have $\neg I \models s$ *insert* $\{\#L\#\}$ B
using *assms cons* **unfolding** *true-clss-cls-def* **by** *simp*
then show $I \models s\ CNot\ \{\#L\#\}$

using *tot I* **by** (*cases L*) *auto*
qed

lemma *true-annots-true-cls-def-iff-negation-in-model*:
$M \models as\ CNot\ C \longleftrightarrow (\forall L \in\# C.\ -L \in lits\text{-}of\text{-}l\ M)$
unfolding *CNot-def true-annots-true-cls true-clss-def* **by** *auto*

lemma *true-annot-CNot-diff*:
$I \models as\ CNot\ C \Longrightarrow I \models as\ CNot\ (C - C')$
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*:
$consistent\text{-}interp\ M \Longrightarrow M \models s\ CNot\ D \Longrightarrow \neg tautology\ D$
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*: $atms\text{-}of\text{-}ms\ (CNot\ CC) = atms\text{-}of\text{-}ms\ \{CC\}$
by *simp*

lemma *total-over-m-CNot-toal-over-m*[*simp*]:
$total\text{-}over\text{-}m\ I\ (CNot\ C) = total\text{-}over\text{-}set\ I\ (atms\text{-}of\ C)$
unfolding *total-over-m-def total-over-set-def* **by** *auto*

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 *uminus-lit-swap*: $-(a::'a\ literal) = i \longleftrightarrow a = -i$
by *auto*

lemma *true-clss-cls-plus-CNot*:
assumes
CC-L: $A \models p\ CC + \{\#L\#\}$ **and**
CNot-CC: $A \models ps\ CNot\ CC$
shows $A \models p\ \{\#L\#\}$
unfolding *true-clss-clss-def true-clss-cls-def CNot-def total-over-m-def*
proof (*intro allI impI*)
fix *I*
assume
tot: $total\text{-}over\text{-}set\ I\ (atms\text{-}of\text{-}ms\ (A \cup \{\{\#L\#\}\}))$ **and**
cons: $consistent\text{-}interp\ I$ **and**
I: $I \models s\ A$
let $?I = I \cup \{Pos\ P|P.\ P \in atms\text{-}of\ CC \land P \notin atm\text{-}of\ `\ I\}$
have *cons′*: $consistent\text{-}interp\ ?I$
using *cons* **unfolding** *consistent-interp-def*
by (*auto simp*: *uminus-lit-swap atms-of-def rev-image-eqI*)
have *I′*: $?I \models s\ A$
using *I true-clss-union-increase* **by** *blast*
have *tot-CNot*: $total\text{-}over\text{-}m\ ?I\ (A \cup CNot\ CC)$
using *tot atms-of-s-def* **by** (*fastforce simp*: *total-over-m-def total-over-set-def*)

then have *tot-I-A-CC-L*: $total\text{-}over\text{-}m\ ?I\ (A \cup \{CC + \{\#L\#\}\})$
using *tot* **unfolding** *total-over-m-def total-over-set-atm-of* **by** *auto*

then have *?I* ⊨ *CC* + {#*L*#} **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** ¬ *lits-of-l A* ⊨*l* − *L*
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 $L \in$ *lits-of-l* $A \vee -L \in$ *lits-of-l* A
using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* **by** *metis*
then have $L \in$ *lits-of-l* A **using** ‹ ¬ *lits-of-l* $A \models l - L$› **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*

4.3.5 Other

abbreviation *no-dup* $L \equiv$ *distinct* (*map* ($\lambda l.$ *atm-of* (*lit-of* l)) L)

lemma *no-dup-rev*[*simp*]:
no-dup (*rev* M) $\longleftrightarrow$ *no-dup* M
by (*auto simp*: *rev-map*[*symmetric*])

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*)

lemma *distinct-consistent-interp*:
no-dup $M \Longrightarrow$ *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 *a2*: *atm-of* (*lit-of* L) $\notin$ ($\lambda l.$ *atm-of* (*lit-of* l)) ‘ *set* M **using** *Cons.prems* **by** *auto*
have *undefined-lit* M (*lit-of* L)
using *a2* **unfolding** *defined-lit-map* **by** *fastforce*
then show *?case*
using *a1* **by** *simp*
qed

lemma *distinct-get-all-ann-decomposition-no-dup*:
assumes (a, b) $\in$ *set* (*get-all-ann-decomposition* M)
and *no-dup* M
shows *no-dup* (a @ b)
using *assms* **by** *force*

lemma *true-annots-lit-of-notin-skip*:
assumes L # M $\models$*as CNot* A
and $-$*lit-of* L ∉# A
and *no-dup* (L # M)
shows M $\models$*as CNot* A
proof $-$
have $\forall\, l$ ∈# A. $-l \in$ *lits-of-l* (L # M)
using *assms*(*1*) *in-CNot-implies-uminus*(*2*) **by** *blast*
moreover
have *atm-of* (*lit-of* L) $\notin$ *atm-of* ‘ *lits-of-l* M

using *assms(3)* **unfolding** *lits-of-def* **by** *force*
then have $-$ *lit-of L* $\notin$ *lits-of-l M* **unfolding** *lits-of-def*
by (*metis* (*no-types*) *atm-of-uminus imageI*)
ultimately have $\forall$ *l* $\in\#$ *A*. $-l \in$ *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

4.3.6 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** $\models$*asm 50*) **where**
I $\models$*asm C* $\equiv$ *I* $\models$*as* (*set-mset C*)

abbreviation *true-clss-clss-m*:: *'v clause multiset* $\Rightarrow$ *'v clause multiset* $\Rightarrow$ *bool* (**infix** $\models$*psm 50*) **where**
I $\models$*psm C* $\equiv$ *set-mset I* $\models$*ps* (*set-mset C*)

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

lemma *true-clss-clssm-subsetE*: *N* $\models$*psm B* $\Longrightarrow$ *A* $\subseteq\#$ *B* $\Longrightarrow$ *N* $\models$*psm A*
using *set-mset-mono true-clss-clss-subsetE* **by** *blast*

abbreviation *true-clss-cls-m*:: *'a clause multiset* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool* (**infix** $\models$*pm 50*) **where**
I $\models$*pm C* $\equiv$ *set-mset I* $\models$*p C*

abbreviation *distinct-mset-mset* :: *'a multiset multiset* $\Rightarrow$ *bool* **where**
distinct-mset-mset $\Sigma \equiv$ *distinct-mset-set* (*set-mset* Σ)

abbreviation *all-decomposition-implies-m* **where**
all-decomposition-implies-m A B $\equiv$ *all-decomposition-implies* (*set-mset A*) *B*

abbreviation *atms-of-mm* :: *'a literal multiset multiset* $\Rightarrow$ *'a set* **where**
atms-of-mm U $\equiv$ *atms-of-ms* (*set-mset U*)

Other definition using *Union-mset*

lemma *atms-of-mm U* $\equiv$ *set-mset* ($\bigcup\#$ *image-mset* (*image-mset atm-of*) *U*)
unfolding *atms-of-ms-def* **by** (*auto simp*: *atms-of-def*)

abbreviation *true-clss-m*:: *'a interp* $\Rightarrow$ *'a clause multiset* $\Rightarrow$ *bool* (**infix** $\models$*sm 50*) **where**
I $\models$*sm C* $\equiv$ *I* $\models$*s set-mset C*

abbreviation *true-clss-ext-m* (**infix** $\models$*sextm 49*) **where**
I $\models$*sextm C* $\equiv$ *I* $\models$*sext set-mset C*

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

Chapter 5

NOT's CDCL and DPLL

theory *CDCL-WNOT-Measure*
imports *Main List-More*
begin

The organisation of the development is the following:

- `CDCL_WNOT_Measure.thy` contains the measure used to show the termination the core of CDCL.
- `CDCL_NOT.thy` contains the specification of the rules: the rules are defined, and we proof the correctness and termination for some strategies CDCL.
- `DPLL_NOT.thy` contains the DPLL calculus based on the CDCL version.
- `DPLL_W.thy` contains Weidenbach's version of DPLL and the proof of equivalence between the two DPLL versions.

5.1 Measure

This measure show the termination of the core of CDCL: each step improves the number of literals we know for sure.

This measure can also be seen as the increasing lexicographic order: it is an order on bounded sequences, when each element is bounded. The proof involves a measure like the one defined here (the same?).

definition μ_C :: $nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat$ **where**
$\mu_C\ s\ b\ M \equiv (\sum i{=}0..{<}length\ M.\ M!i * b\hat{}\ (s + i - length\ M))$

lemma μ_C*-Nil*[*simp*]:
  $\mu_C\ s\ b\ [] = 0$
  unfolding μ_C*-def* **by** *auto*

lemma μ_C*-single*[*simp*]:
  $\mu_C\ s\ b\ [L] = L * b\ \hat{}\ (s - Suc\ 0)$
  unfolding μ_C*-def* **by** *auto*

lemma *set-sum-atLeastLessThan-add*:
  $(\sum i{=}k..{<}k{+}(b{::}nat).\ f\ i) = (\sum i{=}0..{<}b.\ f\ (k{+}\ i))$
  by (*induction b*) *auto*

lemma *set-sum-atLeastLessThan-Suc*:
  $(\sum i{=}1..{<}Suc\ j.\ f\ i) = (\sum i{=}0..{<}j.\ f\ (Suc\ i))$
  using *set-sum-atLeastLessThan-add*[*of - 1 j*] **by** *force*

lemma μ_C*-cons*:
  $\mu_C\ s\ b\ (L\ \#\ M) = L * b\ \hat{}\ (s - 1 - length\ M) + \mu_C\ s\ b\ M$
proof −
  have $\mu_C\ s\ b\ (L\ \#\ M) = (\sum i{=}0..{<}length\ (L\#M).\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M)))$
    unfolding μ_C*-def* **by** *blast*
  also have $\ldots = (\sum i{=}0..{<}1.\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M)))$
              $+ (\sum i{=}1..{<}length\ (L\#M).\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M)))$
     by (*rule setsum-add-nat-ivl*[*symmetric*]) *simp-all*
  finally have $\mu_C\ s\ b\ (L\ \#\ M) = L * b\ \hat{}\ (s - 1 - length\ M)$
              $+ (\sum i{=}1..{<}length\ (L\#M).\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M)))$
     by *auto*
  moreover {
    have $(\sum i{=}1..{<}length\ (L\#M).\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M))) =$
          $(\sum i{=}0..{<}length\ (M).\ (L\#M)!(Suc\ i) * b\hat{}\ (s + (Suc\ i) - length\ (L\#M)))$
     unfolding *length-Cons set-sum-atLeastLessThan-Suc* **by** *blast*
    also have $\ldots = (\sum i{=}0..{<}length\ (M).\ M!i * b\hat{}\ (s + i - length\ M))$
      by *auto*
    finally have $(\sum i{=}1..{<}length\ (L\#M).\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M))) = \mu_C\ s\ b\ M$
      unfolding μ_C*-def* **.**
    }
  ultimately show *?thesis* **by** *presburger*
qed

lemma μ_C*-append*:
  assumes $s \geq length\ (M@M')$
  shows $\mu_C\ s\ b\ (M@M') = \mu_C\ (s - length\ M')\ b\ M + \mu_C\ s\ b\ M'$
proof −
  have $\mu_C\ s\ b\ (M@M') = (\sum i{=}0..{<}length\ (M@M').\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
    unfolding μ_C*-def* **by** *blast*
  moreover then have $\ldots = (\sum i{=}0..{<}\ length\ M.\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
              $+ (\sum i{=}length\ M..{<}length\ (M@M').\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
    by (*auto intro*!: *setsum-add-nat-ivl*[*symmetric*])
  moreover
   have $\forall\, i{\in}\{0..{<}\ length\ M\}.\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')) = M\ !\ i * b\ \hat{}\ (s - length\ M' + i - length\ M)$
      using ‹$s \geq length\ (M@M')$› **by** (*auto simp add*: *nth-append ac-simps*)
     then have $\mu_C\ (s - length\ M')\ b\ M = (\sum i{=}0..{<}\ length\ M.\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
      unfolding μ_C*-def* **by** *auto*
  ultimately have $\mu_C\ s\ b\ (M@M') = \mu_C\ (s - length\ M')\ b\ M$
              $+ (\sum i{=}length\ M..{<}length\ (M@M').\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
     by *auto*
  moreover {
    have $(\sum i{=}length\ M..{<}length\ (M@M').\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M'))) =$
          $(\sum i{=}0..{<}length\ M'.\ M'!i * b\hat{}\ (s + i - length\ M'))$
     unfolding *length-append set-sum-atLeastLessThan-add* **by** *auto*
    then have $(\sum i{=}length\ M..{<}length\ (M@M').\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M'))) = \mu_C\ s\ b\ M'$
      unfolding μ_C*-def* **.**
    }
  ultimately show *?thesis* **by** *presburger*

qed

lemma μ_C*-cons-non-empty-inf*:
  assumes *M-ge-1*: $\forall$ *i*$\in$*set M*. $i \geq 1$ **and** *M*: $M \neq []$
  shows μ_C *s b M* $\geq b$ ^ ($s -$ *length M*)
  using *assms* **by** (*cases M*) (*auto simp*: *mult-eq-if* μ_C*-cons*)

Copy of `~~/src/HOL/ex/NatSum.thy` (but generalized to $0 \leq k$)

lemma *sum-of-powers*: $0 \leq k \Longrightarrow (k - 1) * (\sum i=0..<n.\ k\hat{}i) = k\hat{}n - (1::nat)$
  apply (*cases* $k = 0$)
    apply (*cases n*; *simp*)
  by (*induct n*) (*auto simp*: *Nat.nat-distrib*)

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

lemma μ_C*-bounded-non-degenerated*:
  fixes *b* ::*nat*
  assumes
    $b > 0$ **and**
    $M \neq []$ **and**
    M-le: $\forall$ *i* < *length M*. *M*!*i* < *b* **and**
    $s \geq$ *length M*
  shows μ_C *s b M* $< b\hat{}s$
proof −
  consider (*b1*) *b*= *1* | (*b*) *b*>*1* **using** ‹*b*>*0*› **by** (*cases b*) *auto*
  then show *?thesis*
    proof *cases*
      case *b1*
      then have $\forall$ *i* < *length M*. *M*!*i* = *0* **using** *M-le* **by** *auto*
      then have μ_C *s b M* = *0* **unfolding** μ_C*-def* **by** *auto*
      then show *?thesis* **using** ‹*b* > *0*› **by** *auto*
    next
      case *b*
      have $\forall$ *i* $\in$ {*0*..<*length M*}. *M*!*i* ∗ *b*^ (*s* +*i* − *length M*) ≤ (*b*−*1*) ∗ *b*^ (*s* +*i* − *length M*)
        using *M-le* ‹*b* > *1*› **by** *auto*
      then have μ_C *s b M* ≤ ($\sum$ *i*=*0*..<*length M*. (*b*−*1*) ∗ *b*^ (*s* +*i* − *length M*))
         using ‹*M*≠[]› ‹*b*>*0*› **unfolding** μ_C*-def* **by** (*auto intro*: *setsum-mono*)
      also
       have $\forall$ *i* $\in$ {*0*..<*length M*}. (*b*−*1*) ∗ *b*^ (*s* +*i* − *length M*) = (*b*−*1*) ∗ *b*^ *i* ∗ *b*^(*s* − *length M*)
         by (*metis Nat.add-diff-assoc2 add.commute assms*(*4*) *mult.assoc power-add*)
       then have ($\sum$ *i*=*0*..<*length M*. (*b*−*1*) ∗ *b*^ (*s* +*i* − *length M*))
         = ($\sum$ *i*=*0*..<*length M*. (*b*−*1*)∗ *b*^ *i* ∗ *b*^(*s* − *length M*))
         by (*auto simp add*: *ac-simps*)
      also have ... = ($\sum$ *i*=*0*..<*length M*. *b*^ *i*) ∗ *b*^(*s* − *length M*) ∗ (*b*−*1*)
         by (*simp add*: *setsum-left-distrib setsum-right-distrib ac-simps*)
      finally have μ_C *s b M* ≤ ($\sum$ *i*=*0*..<*length M*. *b*^ *i*) ∗ (*b*−*1*) ∗ *b*^(*s* − *length M*)
        by (*simp add*: *ac-simps*)

      also
        have ($\sum$ *i*=*0*..<*length M*. *b*^ *i*)∗ (*b*−*1*) = *b* ^ (*length M*) − *1*
          using *sum-of-powers*[*of b length M*] ‹*b*>*1*›
          by (*auto simp add*: *ac-simps*)
      finally have μ_C *s b M* ≤ (*b* ^ (*length M*) − *1*) ∗ *b* ^ (*s* − *length M*)
        by *auto*
      also have ... < *b* ^ (*length M*) ∗ *b* ^ (*s* − *length M*)

using ‹*b*>*1*› **by** *auto*
also have … = $b \,\hat{}\, s$
by (*metis assms*(*4*) *le-add-diff-inverse power-add*)
finally show *?thesis* **unfolding** μ_C*-def* **by** (*auto simp add*: *ac-simps*)
qed
qed

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

lemma μ_C*-bounded*:
fixes b :: *nat*
assumes
M-le: $\forall\, i < \mathit{length}\ M.\ M!i < b$ **and**
$s \geq \mathit{length}\ M$
$b > 0$
shows $\mu_C\ s\ b\ M < b \,\hat{}\, s$
proof −
consider (*M0*) $M = []$ | (*M*) $b > 0$ **and** $M \neq []$
using *M-le* **by** (*cases b*, *cases M*) *auto*
then show *?thesis*
proof *cases*
case *M0*
then show *?thesis* **using** *M-le* ‹$b > 0$› **by** *auto*
next
case *M*
show *?thesis* **using** μ_C*-bounded-non-degenerated*[*OF M assms*(*1*,*2*)] **by** *arith*
qed
qed

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

lemma μ_C*-base-0*:
assumes $\mathit{length}\ M \leq s$
shows $\mu_C\ s\ 0\ M \leq M!0$
proof −
{
assume $s = \mathit{length}\ M$
moreover {
fix n
have $(\sum i{=}0..{<}n.\ M\ !\ i * (0{::}nat) \,\hat{}\, i) \leq M\ !\ 0$
apply (*induction n rule*: *nat-induct*)
by *simp* (*rename-tac n*, *case-tac n*, *auto*)
}
ultimately have *?thesis* **unfolding** μ_C*-def* **by** *auto*
}
moreover
{
assume $\mathit{length}\ M < s$
then have $\mu_C\ s\ 0\ M = 0$ **unfolding** μ_C*-def* **by** *auto***}**
ultimately show *?thesis* **using** *assms* **unfolding** μ_C*-def* **by** *linarith*
qed

lemma *finite-bounded-pair-list*:
fixes b :: *nat*
shows *finite* $\{(ys, xs).\ \mathit{length}\ xs < s \wedge \mathit{length}\ ys < s \wedge$
$(\forall\, i{<}\ \mathit{length}\ xs.\ xs\ !\ i < b) \wedge (\forall\, i{<}\ \mathit{length}\ ys.\ ys\ !\ i < b)\}$

proof −
  have H: $\{(ys, xs).$ *length xs* $< s \wedge$ *length ys* $< s \wedge$
    $(\forall i<$ *length xs. xs* $! \; i < b) \wedge (\forall i<$ *length ys. ys* $! \; i < b)\}$
    $\subseteq$
    $\{xs.$ *length xs* $< s \wedge (\forall i<$ *length xs. xs* $! \; i < b)\} \times$
    $\{xs.$ *length xs* $< s \wedge (\forall i<$ *length xs. xs* $! \; i < b)\}$
    by *auto*
  moreover have *finite* $\{xs.$ *length xs* $< s \wedge (\forall i<$ *length xs. xs* $! \; i < b)\}$
    by (*rule finite-bounded-list*)
  ultimately show *?thesis* **by** (*auto simp*: *finite-subset*)
qed

definition νNOT :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ (*nat list* $\times$ *nat list*) *set* **where**
νNOT *s base* $= \{(ys, xs).$ *length xs* $< s \wedge$ *length ys* $< s \wedge$
  $(\forall i<$ *length xs. xs* $! \; i <$ *base*$) \wedge (\forall i<$ *length ys. ys* $! \; i <$ *base*$) \wedge$
  $(ys, xs) \in$ *lenlex less-than*$\}$

lemma *finite-*νNOT[*simp*]:
  finite (νNOT *s base*)
proof −
  have νNOT *s base* $\subseteq \{(ys, xs).$ *length xs* $< s \wedge$ *length ys* $< s \wedge$
    $(\forall i<$ *length xs. xs* $! \; i <$ *base*$) \wedge (\forall i<$ *length ys. ys* $! \; i <$ *base*$)\}$
    by (*auto simp*: νNOT*-def*)
  moreover have *finite* $\{(ys, xs).$ *length xs* $< s \wedge$ *length ys* $< s \wedge$
    $(\forall i<$ *length xs. xs* $! \; i <$ *base*$) \wedge (\forall i<$ *length ys. ys* $! \; i <$ *base*$)\}$
      by (*rule finite-bounded-pair-list*)
  ultimately show *?thesis* **by** (*auto simp*: *finite-subset*)
qed

lemma *acyclic-*νNOT: *acyclic* (νNOT *s base*)
  apply (*rule acyclic-subset*[*of lenlex less-than* νNOT *s base*])
    apply (*rule wf-acyclic*)
  by (*auto simp*: νNOT*-def*)

lemma *wf-*νNOT: *wf* (νNOT *s base*)
  by (*rule finite-acyclic-wf*) (*auto simp*: *acyclic-*νNOT)

end
theory *CDCL-NOT*
imports *List-More Wellfounded-More CDCL-WNOT-Measure Partial-Annotated-Clausal-Logic*
begin

5.2 NOT's CDCL

5.2.1 Auxiliary Lemmas and Measure

We define here some more simplification rules, or rules that have been useful as help for some tactic

lemma *no-dup-cannot-not-lit-and-uminus*:
  no-dup M $\Longrightarrow$ − *lit-of xa* $=$ *lit-of x* $\Longrightarrow$ $x \in$ *set M* $\Longrightarrow$ *xa* $\notin$ *set M*
  by (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id'*)

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 *atms-of-uminus-lit-atm-of-lit-of*:
atms-of {# −*lit-of x*. *x* ∈# *A*#} = *atm-of* ‘ (*lit-of* ‘ (*set-mset A*))
unfolding *atms-of-def* **by** (*auto simp add*: *Fun.image-comp*)

lemma *atms-of-ms-single-image-atm-of-lit-of*:
atms-of-ms (*unmark-s A*) = *atm-of* ‘ (*lit-of* ‘ *A*)
unfolding *atms-of-ms-def* **by** *auto*

5.2.2 Initial definitions

The state

We define here an abstraction over operation on the state we are manipulating.

locale *dpll-state-ops* =
fixes
trail :: ′*st* ⇒ (′*v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: ′*st* ⇒ ′*v clauses* **and**
prepend-trail :: (′*v*, *unit*) *ann-lit* ⇒ ′*st* ⇒ ′*st* **and**
tl-trail :: ′*st* ⇒′*st* **and**
$add\text{-}cls_{NOT}$ :: ′*v clause* ⇒ ′*st* ⇒ ′*st* **and**
$remove\text{-}cls_{NOT}$ :: ′*v clause* ⇒ ′*st* ⇒ ′*st*
begin
abbreviation $state_{NOT}$ :: ′*st* ⇒ (′*v*, *unit*) *ann-lit list* × ′*v clauses* **where**
$state_{NOT}$ *S* ≡ (*trail S*, $clauses_{NOT}$ *S*)
end

NOT's state is basically a pair composed of the trail (i.e. the candidate model) and the set of clauses. We abstract this state to convert this state to other states. like Weidenbach's five-tuple.

locale *dpll-state* =
dpll-state-ops
trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ — related to the state
for
trail :: ′*st* ⇒ (′*v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: ′*st* ⇒ ′*v clauses* **and**
prepend-trail :: (′*v*, *unit*) *ann-lit* ⇒ ′*st* ⇒ ′*st* **and**
tl-trail :: ′*st* ⇒′*st* **and**
$add\text{-}cls_{NOT}$ :: ′*v clause* ⇒ ′*st* ⇒ ′*st* **and**
$remove\text{-}cls_{NOT}$ :: ′*v clause* ⇒ ′*st* ⇒ ′*st* +
assumes
$prepend\text{-}trail_{NOT}$:
$state_{NOT}$ (*prepend-trail L st*) = (*L* # *trail st*, $clauses_{NOT}$ *st*) **and**
$tl\text{-}trail_{NOT}$:
$state_{NOT}$ (*tl-trail st*) = (*tl* (*trail st*), $clauses_{NOT}$ *st*) **and**
$add\text{-}cls_{NOT}$:
$state_{NOT}$ ($add\text{-}cls_{NOT}$ *C st*) = (*trail st*, {#*C*#} + $clauses_{NOT}$ *st*) **and**
$remove\text{-}cls_{NOT}$:
$state_{NOT}$ ($remove\text{-}cls_{NOT}$ *C st*) = (*trail st*, *removeAll-mset C* ($clauses_{NOT}$ *st*))
begin
lemma
trail-prepend-trail[*simp*]:
trail (*prepend-trail L st*) = *L* # *trail st*
and
$trail\text{-}tl\text{-}trail_{NOT}$[*simp*]: *trail* (*tl-trail st*) = *tl* (*trail st*) **and**
$trail\text{-}add\text{-}cls_{NOT}$[*simp*]: *trail* ($add\text{-}cls_{NOT}$ *C st*) = *trail st* **and**

trail-remove-cls$_{NOT}$[*simp*]: *trail* (*remove-cls*$_{NOT}$ *C st*) = *trail st* **and**

clauses-prepend-trail[*simp*]:
clauses$_{NOT}$ (*prepend-trail L st*) = *clauses*$_{NOT}$ *st*
and
clauses-tl-trail[*simp*]: *clauses*$_{NOT}$ (*tl-trail st*) = *clauses*$_{NOT}$ *st* **and**
clauses-add-cls$_{NOT}$[*simp*]:
clauses$_{NOT}$ (*add-cls*$_{NOT}$ *C st*) = {#*C*#} + *clauses*$_{NOT}$ *st* **and**
clauses-remove-cls$_{NOT}$[*simp*]:
clauses$_{NOT}$ (*remove-cls*$_{NOT}$ *C st*) = *removeAll-mset C* (*clauses*$_{NOT}$ *st*)
using *prepend-trail*$_{NOT}$[*of L st*] *tl-trail*$_{NOT}$[*of st*] *add-cls*$_{NOT}$[*of C st*] *remove-cls*$_{NOT}$[*of C st*]
by (*cases state*$_{NOT}$ *st*; *auto*)+

We define the following function doing the backtrack in the trail:

function *reduce-trail-to*$_{NOT}$ :: *'a list* ⇒ *'st* ⇒ *'st* **where**
reduce-trail-to$_{NOT}$ *F S* =
(*if length* (*trail S*) = *length F* ∨ *trail S* = [] *then S else reduce-trail-to*$_{NOT}$ *F* (*tl-trail S*))
by *fast*+
termination by (*relation measure* (λ(-, *S*). *length* (*trail S*))) *auto*
declare *reduce-trail-to*$_{NOT}$.*simps*[*simp del*]

Then we need several lemmas about the *reduce-trail-to*$_{NOT}$.

lemma
shows
reduce-trail-to$_{NOT}$-*Nil*[*simp*]: *trail S* = [] ⟹ *reduce-trail-to*$_{NOT}$ *F S* = *S* **and**
reduce-trail-to$_{NOT}$-*eq-length*[*simp*]: *length* (*trail S*) = *length F* ⟹ *reduce-trail-to*$_{NOT}$ *F S* = *S*
by (*auto simp*: *reduce-trail-to*$_{NOT}$.*simps*)

lemma *reduce-trail-to*$_{NOT}$-*length-ne*[*simp*]:
length (*trail S*) ≠ *length F* ⟹ *trail S* ≠ [] ⟹
reduce-trail-to$_{NOT}$ *F S* = *reduce-trail-to*$_{NOT}$ *F* (*tl-trail S*)
by (*auto simp*: *reduce-trail-to*$_{NOT}$.*simps*)

lemma *trail-reduce-trail-to*$_{NOT}$-*length-le*:
assumes *length F* > *length* (*trail S*)
shows *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = []
using *assms* **by** (*induction F S rule*: *reduce-trail-to*$_{NOT}$.*induct*)
(*simp add*: *less-imp-diff-less reduce-trail-to*$_{NOT}$.*simps*)

lemma *trail-reduce-trail-to*$_{NOT}$-*Nil*[*simp*]:
trail (*reduce-trail-to*$_{NOT}$ [] *S*) = []
by (*induction* [] *S rule*: *reduce-trail-to*$_{NOT}$.*induct*)
(*simp add*: *less-imp-diff-less reduce-trail-to*$_{NOT}$.*simps*)

lemma *clauses-reduce-trail-to*$_{NOT}$-*Nil*:
clauses$_{NOT}$ (*reduce-trail-to*$_{NOT}$ [] *S*) = *clauses*$_{NOT}$ *S*
by (*induction* [] *S rule*: *reduce-trail-to*$_{NOT}$.*induct*)
(*simp add*: *less-imp-diff-less reduce-trail-to*$_{NOT}$.*simps*)

lemma *trail-reduce-trail-to*$_{NOT}$-*drop*:
trail (*reduce-trail-to*$_{NOT}$ *F S*) =
(*if length* (*trail S*) ≥ *length F*
then drop (*length* (*trail S*) − *length F*) (*trail S*)
else [])
apply (*induction F S rule*: *reduce-trail-to*$_{NOT}$.*induct*)
apply (*rename-tac F S*, *case-tac trail S*)

apply *auto*[]
apply (*rename-tac list*, *case-tac Suc* (*length list*) > *length F*)
prefer *2* **apply** *simp*
apply (*subgoal-tac Suc* (*length list*) − *length F* = *Suc* (*length list* − *length F*))
apply *simp*
apply *simp*
done

lemma *reduce-trail-to*$_{NOT}$*-skip-beginning*:
assumes *trail S* = *F′* @ *F*
shows *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = *F*
using *assms* **by** (*auto simp*: *trail-reduce-trail-to*$_{NOT}$*-drop*)

lemma *reduce-trail-to*$_{NOT}$*-clauses*[*simp*]:
clauses$_{NOT}$ (*reduce-trail-to*$_{NOT}$ *F S*) = *clauses*$_{NOT}$ *S*
by (*induction F S rule*: *reduce-trail-to*$_{NOT}$*.induct*)
(*simp add*: *less-imp-diff-less reduce-trail-to*$_{NOT}$*.simps*)

lemma *trail-eq-reduce-trail-to*$_{NOT}$*-eq*:
trail S = *trail T* ⟹ *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = *trail* (*reduce-trail-to*$_{NOT}$ *F T*)
apply (*induction F S arbitrary*: *T rule*: *reduce-trail-to*$_{NOT}$*.induct*)
by (*metis trail-tl-trail*$_{NOT}$ *reduce-trail-to*$_{NOT}$*-eq-length reduce-trail-to*$_{NOT}$*-length-ne*
reduce-trail-to$_{NOT}$*-Nil*)

lemma *trail-reduce-trail-to*$_{NOT}$*-add-cls*$_{NOT}$[*simp*]:
no-dup (*trail S*) ⟹
trail (*reduce-trail-to*$_{NOT}$ *F* (*add-cls*$_{NOT}$ *C S*)) = *trail* (*reduce-trail-to*$_{NOT}$ *F S*)
by (*rule trail-eq-reduce-trail-to*$_{NOT}$*-eq*) *simp*

lemma *reduce-trail-to*$_{NOT}$*-trail-tl-trail-decomp*[*simp*]:
trail S = *F′* @ *Decided K* # *F* ⟹
trail (*reduce-trail-to*$_{NOT}$ *F* (*tl-trail S*)) = *F*
apply (*rule reduce-trail-to*$_{NOT}$*-skip-beginning*[*of - tl* (*F′* @ *Decided K* # [])])
by (*cases F′*) (*auto simp add*:*tl-append reduce-trail-to*$_{NOT}$*-skip-beginning*)

lemma *reduce-trail-to*$_{NOT}$*-length*:
length M = *length M′* ⟹ *reduce-trail-to*$_{NOT}$ *M S* = *reduce-trail-to*$_{NOT}$ *M′ S*
apply (*induction M S rule*: *reduce-trail-to*$_{NOT}$*.induct*)
by (*simp add*: *reduce-trail-to*$_{NOT}$*.simps*)

abbreviation *trail-weight* **where**
trail-weight S ≡ *map* ((λ*l*. *1* + *length l*) *o snd*) (*get-all-ann-decomposition* (*trail S*))

As we are defining abstract states, the Isabelle equality about them is too strong: we want the weaker equivalence stating that two states are equal if they cannot be distinguished, i.e. given the getter *trail* and *clauses*$_{NOT}$ do not distinguish them.

definition *state-eq*$_{NOT}$ :: *′st* ⇒ *′st* ⇒ *bool* (**infix** ∼ *50*) **where**
S ∼ *T* ⟷ *trail S* = *trail T* ∧ *clauses*$_{NOT}$ *S* = *clauses*$_{NOT}$ *T*

lemma *state-eq*$_{NOT}$*-ref*[*simp*]:
S ∼ *S*
unfolding *state-eq*$_{NOT}$*-def* **by** *auto*

lemma *state-eq*$_{NOT}$*-sym*:
S ∼ *T* ⟷ *T* ∼ *S*

unfolding *state-eq*$_{NOT}$*-def* **by** *auto*

lemma *state-eq*$_{NOT}$*-trans*:
$S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$
unfolding *state-eq*$_{NOT}$*-def* **by** *auto*

lemma
shows
state-eq$_{NOT}$*-trail*: $S \sim T \Longrightarrow$ *trail* $S =$ *trail* T **and**
state-eq$_{NOT}$*-clauses*: $S \sim T \Longrightarrow$ *clauses*$_{NOT}$ $S =$ *clauses*$_{NOT}$ T
unfolding *state-eq*$_{NOT}$*-def* **by** *auto*

lemmas *state-simp*$_{NOT}$[*simp*] = *state-eq*$_{NOT}$*-trail* *state-eq*$_{NOT}$*-clauses*

lemma *reduce-trail-to*$_{NOT}$*-state-eq*$_{NOT}$*-compatible*:
assumes *ST*: $S \sim T$
shows *reduce-trail-to*$_{NOT}$ F $S \sim$ *reduce-trail-to*$_{NOT}$ F T
proof −
have *clauses*$_{NOT}$ (*reduce-trail-to*$_{NOT}$ F S) = *clauses*$_{NOT}$ (*reduce-trail-to*$_{NOT}$ F T)
using *ST* **by** *auto*
moreover have *trail* (*reduce-trail-to*$_{NOT}$ F S) = *trail* (*reduce-trail-to*$_{NOT}$ F T)
using *trail-eq-reduce-trail-to*$_{NOT}$*-eq*[*of* S T F] *ST* **by** *auto*
ultimately show *?thesis* **by** (*auto simp del*: *state-simp*$_{NOT}$ *simp*: *state-eq*$_{NOT}$*-def*)
qed

end

Definition of the operation

Each possible is in its own locale.

locale *propagate-ops* =
dpll-state trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
for
trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
clauses$_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls$_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ +
fixes
propagate-cond :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow bool$
begin
inductive *propagate*$_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
propagate$_{NOT}$[*intro*]: $C + \{\#L\#\} \in\#$ *clauses*$_{NOT}$ $S \Longrightarrow$ *trail* $S \models as$ *CNot* C
$\Longrightarrow$ *undefined-lit* (*trail* S) L
$\Longrightarrow$ *propagate-cond* (*Propagated* L ()) S
$\Longrightarrow T \sim$ *prepend-trail* (*Propagated* L ()) S
$\Longrightarrow$ *propagate*$_{NOT}$ S T
inductive-cases *propagate*$_{NOT}E$[*elim*]: *propagate*$_{NOT}$ S T

end

locale *decide-ops* =
dpll-state trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
for

trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st*
begin
inductive *decide*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **where**
decide$_{NOT}$[*intro*]: *undefined-lit* (*trail S*) *L* ⟹ *atm-of L* ∈ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
⟹ *T* ∼ *prepend-trail* (*Decided L*) *S*
⟹ *decide*$_{NOT}$ *S T*

inductive-cases *decide*$_{NOT}$*E*[*elim*]: *decide*$_{NOT}$ *S S'*
end

locale *backjumping-ops* =
dpll-state trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* +
fixes
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool*
begin

inductive *backjump* **where**
trail S = *F'* @ *Decided K*# *F*
⟹ *T* ∼ *prepend-trail* (*Propagated L* ()) (*reduce-trail-to*$_{NOT}$ *F S*)
⟹ *C* ∈# *clauses*$_{NOT}$ *S*
⟹ *trail S* ⊨*as CNot C*
⟹ *undefined-lit F L*
⟹ *atm-of L* ∈ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
⟹ *clauses*$_{NOT}$ *S* ⊨*pm C'* + {#*L*#}
⟹ *F* ⊨*as CNot C'*
⟹ *backjump-conds C C' L S T*
⟹ *backjump S T*
inductive-cases *backjumpE*: *backjump S T*

The condition *atm-of L* ∈ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ *lits-of-l* (*trail S*) is not implied by the the condition *clauses*$_{NOT}$ *S* ⊨*pm C'* + {#*L*#} (no negation).

end

5.2.3 DPLL with backjumping

locale *dpll-with-backjumping-ops* =
propagate-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *propagate-conds* +
decide-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ +
backjumping-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *backjump-conds*
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**

$add\text{-}cls_{NOT}$ :: $'v$ $clause \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v$ $clause \Rightarrow 'st \Rightarrow 'st$ **and**
inv :: $'st \Rightarrow bool$ **and**
$backjump\text{-}conds$ :: $'v$ $clause \Rightarrow 'v$ $clause \Rightarrow 'v$ $literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **and**
$propagate\text{-}conds$ :: $('v,\ unit)$ $ann\text{-}lit \Rightarrow 'st \Rightarrow bool$ +
assumes
bj-can-jump:
$\bigwedge S\ C\ F'\ K\ F\ L.$
$inv\ S \Longrightarrow$
$no\text{-}dup\ (trail\ S) \Longrightarrow$
$trail\ S = F'\ @\ Decided\ K\ \#\ F \Longrightarrow$
$C \in\#\ clauses_{NOT}\ S \Longrightarrow$
$trail\ S \models as\ CNot\ C \Longrightarrow$
$undefined\text{-}lit\ F\ L \Longrightarrow$
$atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F)) \Longrightarrow$
$clauses_{NOT}\ S \models pm\ C' + \{\#L\#\} \Longrightarrow$
$F \models as\ CNot\ C' \Longrightarrow$
$\neg no\text{-}step\ backjump\ S$
begin

We cannot add a like condition $atms\text{-}of\ C' \subseteq atms\text{-}of\text{-}ms\ N$ to ensure that we can backjump even if the last decision variable has disappeared from the set of clauses.

The part of the condition $atm\text{-}of\ L \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F)$ is important, otherwise you are not sure that you can backtrack.

Definition

We define dpll with backjumping:

inductive $dpll\text{-}bj$ :: $'st \Rightarrow 'st \Rightarrow bool$ **for** S :: $'st$ **where**
$bj\text{-}decide_{NOT}$: $decide_{NOT}\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$ |
$bj\text{-}propagate_{NOT}$: $propagate_{NOT}\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$ |
$bj\text{-}backjump$: $backjump\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$

lemmas $dpll\text{-}bj\text{-}induct = dpll\text{-}bj.induct[split\text{-}format(complete)]$
thm $dpll\text{-}bj\text{-}induct[OF\ dpll\text{-}with\text{-}backjumping\text{-}ops\text{-}axioms]$
lemma $dpll\text{-}bj\text{-}all\text{-}induct[consumes\ 2,\ case\text{-}names\ decide_{NOT}\ propagate_{NOT}\ backjump]$:
fixes $S\ T$ :: $'st$
assumes
$dpll\text{-}bj\ S\ T$ **and**
$inv\ S$
$\bigwedge L\ T.\ undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
$\Longrightarrow T \sim prepend\text{-}trail\ (Decided\ L)\ S$
$\Longrightarrow P\ S\ T$ **and**
$\bigwedge C\ L\ T.\ C + \{\#L\#\} \in\#\ clauses_{NOT}\ S \Longrightarrow trail\ S \models as\ CNot\ C \Longrightarrow undefined\text{-}lit\ (trail\ S)\ L$
$\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ S$
$\Longrightarrow P\ S\ T$ **and**
$\bigwedge C\ F'\ K\ F\ L\ C'\ T.\ C \in\#\ clauses_{NOT}\ S \Longrightarrow F'\ @\ Decided\ K\ \#\ F \models as\ CNot\ C$
$\Longrightarrow trail\ S = F'\ @\ Decided\ K\ \#\ F$
$\Longrightarrow undefined\text{-}lit\ F\ L$
$\Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F))$
$\Longrightarrow clauses_{NOT}\ S \models pm\ C' + \{\#L\#\}$
$\Longrightarrow F \models as\ CNot\ C'$
$\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ S)$
$\Longrightarrow P\ S\ T$

shows *P S T*
apply (*induct T rule*: *dpll-bj-induct*[*OF local.dpll-with-backjumping-ops-axioms*])
apply (*rule assms*(*1*))
using *assms*(*3*) **apply** *blast*
apply (*elim propagate*$_{NOT}$*E*) **using** *assms*(*4*) **apply** *blast*
apply (*elim backjumpE*) **using** *assms*(*5*) ‹*inv S*› **by** *simp*

Basic properties

First, some better suited induction principle **lemma** *dpll-bj-clauses*:
assumes *dpll-bj S T* **and** *inv S*
shows *clauses*$_{NOT}$ *S* = *clauses*$_{NOT}$ *T*
using *assms* **by** (*induction rule*: *dpll-bj-all-induct*) *auto*

No duplicates in the trail **lemma** *dpll-bj-no-dup*:
assumes *dpll-bj S T* **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
using *assms* **by** (*induction rule*: *dpll-bj-all-induct*)
(*auto simp add*: *defined-lit-map reduce-trail-to*$_{NOT}$*-skip-beginning*)

Valuations **lemma** *dpll-bj-sat-iff*:
assumes *dpll-bj S T* **and** *inv S*
shows *I* ⊨*sm clauses*$_{NOT}$ *S* ⟷ *I* ⊨*sm clauses*$_{NOT}$ *T*
using *assms* **by** (*induction rule*: *dpll-bj-all-induct*) *auto*

Clauses **lemma** *dpll-bj-atms-of-ms-clauses-inv*:
assumes
dpll-bj S T **and**
inv S
shows *atms-of-mm* (*clauses*$_{NOT}$ *S*) = *atms-of-mm* (*clauses*$_{NOT}$ *T*)
using *assms* **by** (*induction rule*: *dpll-bj-all-induct*) *auto*

lemma *dpll-bj-atms-in-trail*:
assumes
dpll-bj S T **and**
inv S **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
using *assms* **by** (*induction rule*: *dpll-bj-all-induct*)
(*auto simp*: *in-plus-implies-atm-of-on-atms-of-ms reduce-trail-to*$_{NOT}$*-skip-beginning*)

lemma *dpll-bj-atms-in-trail-in-set*:
assumes *dpll-bj S T***and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *A*
using *assms* **by** (*induction rule*: *dpll-bj-all-induct*)
(*auto simp*: *in-plus-implies-atm-of-on-atms-of-ms*)

lemma *dpll-bj-all-decomposition-implies-inv*:
assumes
dpll-bj S T **and**
inv: *inv S* **and**

decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
using *assms*(*1*,*2*)
proof (*induction rule*:*dpll-bj-all-induct*)
case *decide*$_{NOT}$
then show *?case* **using** *decomp* **by** *auto*
next
case (*propagate*$_{NOT}$ *C L T*) **note** *propa* = *this*(*1*) **and** *undef* = *this*(*3*) **and** *T* = *this*(*4*)
let *?M′* = *trail* (*prepend-trail* (*Propagated L* ()) *S*)
let *?N* = *clauses*$_{NOT}$ *S*
obtain *a y l* **where** *ay*: *get-all-ann-decomposition ?M′* = (*a*, *y*) # *l*
by (*cases get-all-ann-decomposition ?M′*) *fastforce+*
then have *M′*: *?M′* = *y* @ *a* **using** *get-all-ann-decomposition-decomp*[*of ?M′*] **by** *auto*
have *M*: *get-all-ann-decomposition* (*trail S*) = (*a*, *tl y*) # *l*
using *ay undef* **by** (*cases get-all-ann-decomposition* (*trail S*)) *auto*
have y_0: *y* = (*Propagated L* ()) # (*tl y*)
using *ay undef* **by** (*auto simp add*: *M*)
from *arg-cong*[*OF this*, *of set*] **have** *y*[*simp*]: *set y* = *insert* (*Propagated L* ()) (*set* (*tl y*))
by *simp*
have *tr-S*: *trail S* = *tl y* @ *a*
using *arg-cong*[*OF M′*, *of tl*] y_0 *M get-all-ann-decomposition-decomp* **by** *force*
have *a-Un-N-M*: *unmark-l a* ∪ *set-mset ?N* ⊨*ps unmark-l* (*tl y*)
using *decomp ay* **unfolding** *all-decomposition-implies-def* **by** (*simp add*: *M*)+

moreover have *unmark-l a* ∪ *set-mset ?N* ⊨*p* {#*L*#} (**is** *?I* ⊨*p* -)
proof (*rule true-clss-cls-plus-CNot*)
show *?I* ⊨*p C* + {#*L*#}
using *propa propagate*$_{NOT}$*.prems* **by** (*auto dest*!: *true-clss-clss-in-imp-true-clss-cls*)
next
have *unmark-l ?M′* ⊨*ps CNot C*
using ‹*trail S* ⊨*as CNot C*› *undef* **by** (*auto simp add*: *true-annots-true-clss-clss*)
have *a1*: *unmark-l a* ∪ *unmark-l* (*tl y*) ⊨*ps CNot C*
using *propagate*$_{NOT}$*.hyps*(*2*) *tr-S true-annots-true-clss-clss*
by (*force simp add*: *image-Un sup-commute*)
then have *unmark-l a* ∪ *set-mset* (*clauses*$_{NOT}$ *S*) ⊨*ps unmark-l a* ∪ *unmark-l* (*tl y*)
using *a-Un-N-M true-clss-clss-def* **by** *blast*
then show *unmark-l a* ∪ *set-mset* (*clauses*$_{NOT}$ *S*) ⊨*ps CNot C*
using *a1* **by** (*meson true-clss-clss-left-right true-clss-clss-union-and*
true-clss-clss-union-l-r)
qed
ultimately have *unmark-l a* ∪ *set-mset ?N* ⊨*ps unmark-l ?M′*
unfolding *M′* **by** (*auto simp add*: *all-in-true-clss-clss image-Un*)
then show *?case*
using *decomp T M undef* **unfolding** *ay all-decomposition-implies-def* **by** (*auto simp add*: *ay*)
next
case (*backjump C F′ K F L D T*) **note** *confl* = *this*(*2*) **and** *tr* = *this*(*3*) **and** *undef* = *this*(*4*) **and**
L = *this*(*5*) **and** *N-C* = *this*(*6*) **and** *vars-D* = *this*(*5*) **and** *T* = *this*(*8*)
have *decomp*: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition F*)
using *decomp* **unfolding** *tr all-decomposition-implies-def*
by (*metis* (*no-types*, *lifting*) *get-all-ann-decomposition.simps*(*1*)
get-all-ann-decomposition-never-empty hd-Cons-tl insert-iff list.sel(*3*) *list.set*(*2*)
tl-get-all-ann-decomposition-skip-some)

obtain *a b li* **where** *F*: *get-all-ann-decomposition F* = (*a*, *b*) # *li*
by (*cases get-all-ann-decomposition F*) *auto*
have *F* = *b* @ *a*

using *get-all-ann-decomposition-decomp*[*of F a b*] *F* **by** *auto*
have *a-N-b*:*unmark-l a* ∪ *set-mset* (*clauses*$_{NOT}$ *S*) ⊨*ps unmark-l b*
using *decomp* **unfolding** *all-decomposition-implies-def* **by** (*auto simp add*: *F*)

have *F-D*: *unmark-l F* ⊨*ps CNot D*
using ‹*F* ⊨*as CNot D*› **by** (*simp add*: *true-annots-true-clss-clss*)
then have *unmark-l a* ∪ *unmark-l b* ⊨*ps CNot D*
unfolding ‹*F* = *b* @ *a*› **by** (*simp add*: *image-Un sup.commute*)
have *a-N-CNot-D*: *unmark-l a* ∪ *set-mset* (*clauses*$_{NOT}$ *S*) ⊨*ps CNot D* ∪ *unmark-l b*
apply (*rule true-clss-clss-left-right*)
using *a-N-b F-D* **unfolding** ‹*F* = *b* @ *a*› **by** (*auto simp add*: *image-Un ac-simps*)

have *a-N-D-L*: *unmark-l a* ∪ *set-mset* (*clauses*$_{NOT}$ *S*) ⊨*p D*+{#*L*#}
by (*simp add*: *N-C*)
have *unmark-l a* ∪ *set-mset* (*clauses*$_{NOT}$ *S*) ⊨*p* {#*L*#}
using *a-N-D-L a-N-CNot-D* **by** (*blast intro*: *true-clss-cls-plus-CNot*)
then show *?case*
using *decomp T tr undef* **unfolding** *all-decomposition-implies-def* **by** (*auto simp add*: *F*)
qed

Termination

Using a proper measure **lemma** *length-get-all-ann-decomposition-append-Decided*:
length (*get-all-ann-decomposition* (*F'* @ *Decided K* # *F*)) =
length (*get-all-ann-decomposition F'*)
+ *length* (*get-all-ann-decomposition* (*Decided K* # *F*))
− *1*
by (*induction F' rule*: *ann-lit-list-induct*) *auto*

lemma *take-length-get-all-ann-decomposition-decided-sandwich*:
take (*length* (*get-all-ann-decomposition F*))
(*map* (*f o snd*) (*rev* (*get-all-ann-decomposition* (*F'* @ *Decided K* # *F*))))
=
map (*f o snd*) (*rev* (*get-all-ann-decomposition F*))

proof (*induction F' rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Decided K*)
then show *?case* **by** (*simp add*: *length-get-all-ann-decomposition-append-Decided*)
next
case (*Propagated L m F'*) **note** *IH* = *this*(*1*)
obtain *a b l* **where** *F'*: *get-all-ann-decomposition* (*F'* @ *Decided K* # *F*) = (*a*, *b*) # *l*
by (*cases get-all-ann-decomposition* (*F'* @ *Decided K* # *F*)) *auto*
have *length* (*get-all-ann-decomposition F*) − *length l* = *0*
using *length-get-all-ann-decomposition-append-Decided*[*of F' K F*]
unfolding *F'* **by** (*cases get-all-ann-decomposition F'*) *auto*
then show *?case*
using *IH* **by** (*simp add*: *F'*)
qed

lemma *length-get-all-ann-decomposition-length*:
length (*get-all-ann-decomposition M*) ≤ *1* + *length M*
by (*induction M rule*: *ann-lit-list-induct*) *auto*

lemma *length-in-get-all-ann-decomposition-bounded*:
  assumes *i*:$i \in$ *set* (*trail-weight S*)
  shows $i \leq$ *Suc* (*length* (*trail S*))
proof −
  obtain *a b* **where**
    (*a*, *b*) $\in$ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
    ib: *i* = *Suc* (*length b*)
    using *i* **by** *auto*
  then obtain *c* **where** *trail S* = *c* @ *b* @ *a*
    using *get-all-ann-decomposition-exists-prepend'* **by** *metis*
  from *arg-cong*[*OF this*, *of length*] **show** *?thesis* **using** *i ib* **by** *auto*
qed

Well-foundedness The bounds are the following:

- *1* + *card* (*atms-of-ms A*): *card* (*atms-of-ms A*) is an upper bound on the length of the list. As *get-all-ann-decomposition* appends an possibly empty couple at the end, adding one is needed.
- *2* + *card* (*atms-of-ms A*): *card* (*atms-of-ms A*) is an upper bound on the number of elements, where adding one is necessary for the same reason as for the bound on the list, and one is needed to have a strict bound.

abbreviation *unassigned-lit* :: *'b literal multiset set* $\Rightarrow$ *'a list* $\Rightarrow$ *nat* **where**
  unassigned-lit N M $\equiv$ *card* (*atms-of-ms N*) − *length M*
lemma *dpll-bj-trail-mes-increasing-prop*:
  fixes *M* :: (*'v*, *unit*) *ann-lits* **and** *N* :: *'v clauses*
  assumes
    dpll-bj S T **and**
    inv S **and**
    NA: *atms-of-mm* ($clauses_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
    MA: *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* **and**
    n-d: *no-dup* (*trail S*) **and**
    finite: *finite A*
  shows μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*)
    > μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)
  using *assms*(*1*,*2*)
proof (*induction rule*: *dpll-bj-all-induct*)
  case ($propagate_{NOT}$ *C L*) **note** *CLN* = *this*(*1*) **and** *MC* = *this*(*2*) **and** *undef-L* = *this*(*3*) **and** *T* = *this*(*4*)
  have *incl*: *atm-of* ‘ *lits-of-l* (*Propagated L* () # *trail S*) $\subseteq$ *atms-of-ms A*
    using $propagate_{NOT}$ *dpll-bj-atms-in-trail-in-set* $bj\text{-}propagate_{NOT}$ *NA MA CLN*
    by (*auto simp*: *in-plus-implies-atm-of-on-atms-of-ms*)

  have *no-dup*: *no-dup* (*Propagated L* () # *trail S*)
    using *defined-lit-map n-d undef-L* **by** *auto*
  obtain *a b l* **where** *M*: *get-all-ann-decomposition* (*trail S*) = (*a*, *b*) # *l*
    by (*cases get-all-ann-decomposition* (*trail S*)) *auto*
  have *b-le-M*: *length b* $\leq$ *length* (*trail S*)
    using *get-all-ann-decomposition-decomp*[*of trail S*] **by** (*simp add*: *M*)
  have *finite* (*atms-of-ms A*) **using** *finite* **by** *simp*

  then have *length* (*Propagated L* () # *trail S*) $\leq$ *card* (*atms-of-ms A*)
    using *incl finite* **unfolding** *no-dup-length-eq-card-atm-of-lits-of-l*[*OF no-dup*]

by (*simp add*: *card-mono*)
then have *latm*: *unassigned-lit A b* = *Suc* (*unassigned-lit A* (*Propagated L d* # *b*))
using *b-le-M* **by** *auto*
then show *?case* **using** *T undef-L* **by** (*auto simp*: *latm M* μ_C-*cons*)
next
case ($decide_{NOT}$ *L*) **note** *undef-L* = *this*(*1*) **and** *MC* = *this*(*2*) **and** *T* = *this*(*3*)
have *incl*: *atm-of* ‘ *lits-of-l* (*Decided L* # (*trail S*)) ⊆ *atms-of-ms A*
using *dpll-bj-atms-in-trail-in-set bj-*$decide_{NOT}$ $decide_{NOT}$.$decide_{NOT}$[*OF* $decide_{NOT}$.*hyps*] *NA MA MC*
by *auto*

have *no-dup*: *no-dup* (*Decided L* # (*trail S*))
using *defined-lit-map n-d undef-L* **by** *auto*
obtain *a b l* **where** *M*: *get-all-ann-decomposition* (*trail S*) = (*a*, *b*) # *l*
by (*cases get-all-ann-decomposition* (*trail S*)) *auto*

then have *length* (*Decided L* # (*trail S*)) ≤ *card* (*atms-of-ms A*)
using *incl finite* **unfolding** *no-dup-length-eq-card-atm-of-lits-of-l*[*OF no-dup*]
by (*simp add*: *card-mono*)
show *?case* **using** *T undef-L* **by** (*simp add*: μ_C-*cons*)
next
case (*backjump C F′ K F L C′ T*) **note** *undef-L* = *this*(*4*) **and** *MC* = *this*(*1*) **and** *tr-S* = *this*(*3*) **and**
L = *this*(*5*) **and** *T* = *this*(*8*)
have *incl*: *atm-of* ‘ *lits-of-l* (*Propagated L* () # *F*) ⊆ *atms-of-ms A*
using *dpll-bj-atms-in-trail-in-set NA MA L* **by** (*auto simp*: *tr-S*)

have *no-dup*: *no-dup* (*Propagated L* () # *F*)
using *defined-lit-map n-d undef-L tr-S* **by** *auto*
obtain *a b l* **where** *M*: *get-all-ann-decomposition* (*trail S*) = (*a*, *b*) # *l*
by (*cases get-all-ann-decomposition* (*trail S*)) *auto*
have *b-le-M*: *length b* ≤ *length* (*trail S*)
using *get-all-ann-decomposition-decomp*[*of trail S*] **by** (*simp add*: *M*)
have *fin-atms-A*: *finite* (*atms-of-ms A*) **using** *finite* **by** *simp*

then have *F-le-A*: *length* (*Propagated L* () # *F*) ≤ *card* (*atms-of-ms A*)
using *incl finite* **unfolding** *no-dup-length-eq-card-atm-of-lits-of-l*[*OF no-dup*]
by (*simp add*: *card-mono*)
have *tr-S-le-A*: *length* (*trail S*) ≤ *card* (*atms-of-ms A*)
using *n-d MA* **by** (*metis fin-atms-A card-mono no-dup-length-eq-card-atm-of-lits-of-l*)
obtain *a b l* **where** *F*: *get-all-ann-decomposition F* = (*a*, *b*) # *l*
by (*cases get-all-ann-decomposition F*) *auto*
then have *F* = *b* @ *a*
using *get-all-ann-decomposition-decomp*[*of Propagated L* () # *F a*
Propagated L () # *b*] **by** *simp*
then have *latm*: *unassigned-lit A b* = *Suc* (*unassigned-lit A* (*Propagated L* () # *b*))
using *F-le-A* **by** *simp*
obtain *rem* **where**
rem:*map* (λ*a*. *Suc* (*length* (*snd a*))) (*rev* (*get-all-ann-decomposition* (*F′* @ *Decided K* # *F*)))
= *map* (λ*a*. *Suc* (*length* (*snd a*))) (*rev* (*get-all-ann-decomposition F*)) @ *rem*
using *take-length-get-all-ann-decomposition-decided-sandwich*[*of F* λ*a*. *Suc* (*length a*) *F′ K*]
unfolding *o-def* **by** (*metis append-take-drop-id*)
then have *rem*: *map* (λ*a*. *Suc* (*length* (*snd a*)))
(*get-all-ann-decomposition* (*F′* @ *Decided K* # *F*))
= *rev rem* @ *map* (λ*a*. *Suc* (*length* (*snd a*))) ((*get-all-ann-decomposition F*))
by (*simp add*: *rev-map*[*symmetric*] *rev-swap*)

have *length* (*rev rem* @ *map* (λ*a*. *Suc* (*length* (*snd a*))) (*get-all-ann-decomposition F*))
    ≤ *Suc* (*card* (*atms-of-ms A*))
  using *arg-cong*[*OF rem*, *of length*] *tr-S-le-A*
  length-get-all-ann-decomposition-length[*of F′* @ *Decided K* # *F*] *tr-S* **by** *auto*
moreover
  { fix *i* :: *nat* **and** *xs* :: *′a list*
    have *i* < *length xs* ⟹ *length xs* − *Suc i* < *length xs*
      by *auto*
    then have *H*: *i*<*length xs* ⟹ *rev xs* ! *i* ∈ *set xs*
      using *rev-nth*[*of i xs*] **unfolding** *in-set-conv-nth* **by** (*force simp add*: *in-set-conv-nth*)
  } note *H* = *this*
  have ∀ *i*<*length rem*. *rev rem* ! *i* < *card* (*atms-of-ms A*) + *2*
    using *tr-S-le-A length-in-get-all-ann-decomposition-bounded*[*of - S*] **unfolding** *tr-S*
    by (*force simp add*: *o-def rem dest*!: *H intro*: *length-get-all-ann-decomposition-length*)
ultimately show *?case*
  using μ_C-*bounded*[*of rev rem card* (*atms-of-ms A*)+*2 unassigned-lit A l*] *T undef-L*
  by (*simp add*: *rem* μ_C-*append* μ_C-*cons F tr-S*)
qed

lemma *dpll-bj-trail-mes-decreasing-prop*:
assumes *dpll*: *dpll-bj S T* **and** *inv*: *inv S* **and**
N-A: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
M-A: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
nd: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
      − μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*)
    < (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
      − μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)
proof −
let *?b* = *2*+*card* (*atms-of-ms A*)
let *?s* = *1*+*card* (*atms-of-ms A*)
let *?μ* = μ_C *?s ?b*
have *M′-A*: *atm-of* ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
  by (*meson M-A N-A dpll dpll-bj-atms-in-trail-in-set inv*)
have *nd′*: *no-dup* (*trail T*)
  using ‹*dpll-bj S T*› *dpll-bj-no-dup nd inv* **by** *blast*
{ fix *i* :: *nat* **and** *xs* :: *′a list*
  have *i* < *length xs* ⟹ *length xs* − *Suc i* < *length xs*
    by *auto*
  then have *H*: *i*<*length xs* ⟹ *xs* ! *i* ∈ *set xs*
    using *rev-nth*[*of i xs*] **unfolding** *in-set-conv-nth* **by** (*force simp add*: *in-set-conv-nth*)
} note *H* = *this*

have *l-M-A*: *length* (*trail S*) ≤ *card* (*atms-of-ms A*)
  by (*simp add*: *fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd*)
have *l-M′-A*: *length* (*trail T*) ≤ *card* (*atms-of-ms A*)
  by (*simp add*: *fin-A M′-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd′*)
have *l-trail-weight-M*: *length* (*trail-weight T*) ≤ *1*+*card* (*atms-of-ms A*)
  using *l-M′-A length-get-all-ann-decomposition-length*[*of trail T*] **by** *auto*
have *bounded-M*: ∀ *i*<*length* (*trail-weight T*). (*trail-weight T*)! *i* < *card* (*atms-of-ms A*) + *2*
  using *length-in-get-all-ann-decomposition-bounded*[*of - T*] *l-M′-A*
  by (*metis* (*no-types*, *lifting*) *H Nat.le-trans add-2-eq-Suc′ not-le not-less-eq-eq*)

from *dpll-bj-trail-mes-increasing-prop*[*OF dpll inv N-A M-A nd fin-A*]
have μ_C *?s ?b* (*trail-weight S*) < μ_C *?s ?b* (*trail-weight T*) **by** *simp*

moreover from μ_C-*bounded*[*OF bounded-M l-trail-weight-M*]
  have μ_C *?s ?b* (*trail-weight T*) $\leq$ *?b* ^ *?s* **by** *auto*
ultimately show *?thesis* **by** *linarith*
qed

lemma *wf-dpll-bj*:
assumes *fin*: *finite A*
shows *wf* {(*T*, *S*). *dpll-bj S T*
  $\wedge$ *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* $\wedge$ *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A*
  $\wedge$ *no-dup* (*trail S*) $\wedge$ *inv S*}
(**is** *wf ?A*)
proof (*rule wf-bounded-measure*[*of* -
    λ-. (*2* + *card* (*atms-of-ms A*))^(*1* + *card* (*atms-of-ms A*))
    λS. μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)])
fix *a b* :: *'st*
let *?b* = *2*+*card* (*atms-of-ms A*)
let *?s* = *1*+*card* (*atms-of-ms A*)
let *?μ* = μ_C *?s ?b*
assume *ab*: (*b*, *a*) $\in$ *?A*

have *fin-A*: *finite* (*atms-of-ms A*)
  using *fin* **by** *auto*
have
  dpll-bj: *dpll-bj a b* **and**
  N-A: *atms-of-mm* (*clauses*$_{NOT}$ *a*) $\subseteq$ *atms-of-ms A* **and**
  M-A: *atm-of* ' *lits-of-l* (*trail a*) $\subseteq$ *atms-of-ms A* **and**
  nd: *no-dup* (*trail a*) **and**
  inv: *inv a*
  using *ab* **by** *auto*

have *M'-A*: *atm-of* ' *lits-of-l* (*trail b*) $\subseteq$ *atms-of-ms A*
  by (*meson M-A N-A* ‹*dpll-bj a b*› *dpll-bj-atms-in-trail-in-set inv*)
have *nd'*: *no-dup* (*trail b*)
  using ‹*dpll-bj a b*› *dpll-bj-no-dup nd inv* **by** *blast*
{ fix *i* :: *nat* **and** *xs* :: *'a list*
  have *i* < *length xs* $\Longrightarrow$ *length xs* − *Suc i* < *length xs*
    by *auto*
  then have *H*: *i*<*length xs* $\Longrightarrow$ *xs* ! *i* $\in$ *set xs*
    using *rev-nth*[*of i xs*] **unfolding** *in-set-conv-nth* **by** (*force simp add*: *in-set-conv-nth*)
} note *H* = *this*

have *l-M-A*: *length* (*trail a*) $\leq$ *card* (*atms-of-ms A*)
  by (*simp add*: *fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd*)
have *l-M'-A*: *length* (*trail b*) $\leq$ *card* (*atms-of-ms A*)
  by (*simp add*: *fin-A M'-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd'*)
have *l-trail-weight-M*: *length* (*trail-weight b*) $\leq$ *1*+*card* (*atms-of-ms A*)
  using *l-M'-A length-get-all-ann-decomposition-length*[*of trail b*] **by** *auto*
have *bounded-M*: $\forall$ *i*<*length* (*trail-weight b*). (*trail-weight b*)! *i* < *card* (*atms-of-ms A*) + *2*
  using *length-in-get-all-ann-decomposition-bounded*[*of* - *b*] *l-M'-A*
  by (*metis* (*no-types*, *lifting*) *Nat.le-trans One-nat-def Suc-1 add.right-neutral add-Suc-right*
    le-imp-less-Suc less-eq-Suc-le nth-mem)

from *dpll-bj-trail-mes-increasing-prop*[*OF dpll-bj inv N-A M-A nd fin*]
have μ_C *?s ?b* (*trail-weight a*) < μ_C *?s ?b* (*trail-weight b*) **by** *simp*
moreover from μ_C-*bounded*[*OF bounded-M l-trail-weight-M*]
  have μ_C *?s ?b* (*trail-weight b*) $\leq$ *?b* ^ *?s* **by** *auto*

ultimately show *?b* ^ *?s* $\leq$ *?b* ^ *?s* $\wedge$
μ_C *?s ?b* (*trail-weight b*) $\leq$ *?b* ^ *?s* $\wedge$
μ_C *?s ?b* (*trail-weight a*) $<$ μ_C *?s ?b* (*trail-weight b*)
by *blast*
qed

Normal Forms

We prove that given a normal form of DPLL, with some structural invariants, then either N is satisfiable and the built valuation M is a model; or N is unsatisfiable.

Idea of the proof: We have to prove tat *satisfiable N*, $\neg$ *M* $\models$*as N* and there is no remaining step is incompatible.

1. The *decide* rule tells us that every variable in N has a value.
2. The assumption $\neg$ *M* $\models$*as N* implies that there is conflict.
3. There is at least one decision in the trail (otherwise, M would be a model of the set of clauses N).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

   The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj S*

theorem *dpll-backjump-final-state*:
fixes *A* :: *'v clause set* **and** *S T* :: *'st*
assumes
atms-of-mm (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* **and**
no-dup (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
n-s: *no-step dpll-bj S* **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
$\vee$ (*trail S* $\models$*asm clauses*$_{NOT}$ *S* $\wedge$ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))
proof −
let *?N* = *set-mset* (*clauses*$_{NOT}$ *S*)
let *?M* = *trail S*
consider
(*sat*) *satisfiable ?N* **and** *?M* $\models$*as ?N*
| (*sat'*) *satisfiable ?N* **and** $\neg$ *?M* $\models$*as ?N*
| (*unsat*) *unsatisfiable ?N*
by *auto*
then show *?thesis*
proof *cases*
case *sat'* **note** *sat* = *this*(*1*) **and** *M* = *this*(*2*)
obtain *C* **where** *C* $\in$ *?N* **and** $\neg$*?M* $\models$*a C* **using** *M* **unfolding** *true-annots-def* **by** *auto*
obtain *I* :: *'v literal set* **where**
I $\models$*s ?N* **and**
cons: *consistent-interp I* **and**
tot: *total-over-m I ?N* **and**
atm-I-N: *atm-of* ‘*I* $\subseteq$ *atms-of-ms ?N*

using *sat* **unfolding** *satisfiable-def-min* **by** *auto*
let *?I* = *I* ∪ {*P*| *P*. *P* ∈ *lits-of-l ?M* ∧ *atm-of P* ∉ *atm-of* ‘ *I*}
let *?O* = {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set ?M* ∧ *atm-of* (*lit-of L*) ∉ *atms-of-ms ?N*}
have *cons-I′*: *consistent-interp ?I*
using *cons* **using** ‹*no-dup ?M*› **unfolding** *consistent-interp-def*
by (*auto simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def*
dest!: *no-dup-cannot-not-lit-and-uminus*)
have *tot-I′*: *total-over-m ?I* (*?N* ∪ *unmark-l ?M*)
using *tot atm-I-N* **unfolding** *total-over-m-def total-over-set-def*
by (*fastforce simp*: *image-iff lits-of-def*)
have {*P* |*P*. *P* ∈ *lits-of-l ?M* ∧ *atm-of P* ∉ *atm-of* ‘ *I*} ⊨*s ?O*
using ‹*I*⊨*s ?N*› *atm-I-N* **by** (*auto simp add*: *atm-of-eq-atm-of true-clss-def lits-of-def*)
then have *I′-N*: *?I* ⊨*s ?N* ∪ *?O*
using ‹*I*⊨*s ?N*› *true-clss-union-increase* **by** *force*
have *tot′*: *total-over-m ?I* (*?N*∪*?O*)
using *atm-I-N tot* **unfolding** *total-over-m-def total-over-set-def*
by (*force simp*: *lits-of-def elim*!: *is-decided-ex-Decided*)

have *atms-N-M*: *atms-of-ms ?N* ⊆ *atm-of* ‘ *lits-of-l ?M*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *l* :: ′*v* **where**
l-N: *l* ∈ *atms-of-ms ?N* **and**
l-M: *l* ∉ *atm-of* ‘ *lits-of-l ?M*
by *auto*
have *undefined-lit ?M* (*Pos l*)
using *l-M* **by** (*metis Decided-Propagated-in-iff-in-lits-of-l*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(*1*))
from *bj-decide*$_{NOT}$[*OF decide*$_{NOT}$[*OF this*]] **show** *False*
using *l-N n-s* **by** (*metis literal.sel*(*1*) *state-eq*$_{NOT}$*-ref*)
qed
have *?M* ⊨*as CNot C*
apply (*rule all-variables-defined-not-imply-cnot*)
using ‹*C* ∈ *set-mset* (*clauses*$_{NOT}$ *S*)› ‹¬ *trail S* ⊨*a C*›
atms-N-M **by** (*auto dest*: *atms-of-atms-of-ms-mono*)
have ∃ *l* ∈ *set ?M*. *is-decided l*
proof (*rule ccontr*)
let *?O* = {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set ?M* ∧ *atm-of* (*lit-of L*) ∉ *atms-of-ms ?N*}
have *ϑ*[*iff*]: ⋀*I*. *total-over-m I* (*?N* ∪ *?O* ∪ *unmark-l ?M*)
⟷ *total-over-m I* (*?N* ∪*unmark-l ?M*)
unfolding *total-over-set-def total-over-m-def atms-of-ms-def* **by** *blast*
assume ¬ *?thesis*
then have [*simp*]:{*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set ?M*}
= {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set ?M* ∧ *atm-of* (*lit-of L*) ∉ *atms-of-ms ?N*}
by *auto*
then have *?N* ∪ *?O* ⊨*ps unmark-l ?M*
using *all-decomposition-implies-propagated-lits-are-implied*[*OF decomp*] **by** *auto*

then have *?I* ⊨*s unmark-l ?M*
using *cons-I′ I′-N tot-I′* ‹*?I* ⊨*s ?N* ∪ *?O*› **unfolding** *ϑ true-clss-clss-def* **by** *blast*
then have *lits-of-l ?M* ⊆ *?I*
unfolding *true-clss-def lits-of-def* **by** *auto*
then have *?M* ⊨*as ?N*
using *I′-N* ‹*C* ∈ *?N*› ‹¬ *?M* ⊨*a C*› *cons-I′ atms-N-M*
by (*meson* ‹*trail S* ⊨*as CNot C*› *consistent-CNot-not rev-subsetD sup-ge1 true-annot-def*
true-annots-def true-cls-mono-set-mset-l true-clss-def)

then show *False* **using** *M* **by** *fast*
qed
from *List.split-list-first-propE*[*OF this*] **obtain** *K* :: *'v literal* **and**
F F' :: (*'v*, *unit*) *ann-lits* **where**
M-K: *?M* = *F'* @ *Decided K* # *F* **and**
nm: ∀*f*∈*set F'*. ¬*is-decided f*
unfolding *is-decided-def* **by** *metis*
let *?K* = *Decided K* :: (*'v*, *unit*) *ann-lit*
have *?K* ∈ *set ?M*
unfolding *M-K* **by** *auto*
let *?C* = *image-mset lit-of* {#*L*∈#*mset ?M*. *is-decided L* ∧ *L*≠*?K*#} :: *'v clause*
let *?C'* = *set-mset* (*image-mset* (λ*L*::*'v literal*. {#*L*#}) (*?C* + *unmark ?K*))
have *?N* ∪ {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set ?M*} ⊨*ps unmark-l ?M*
using *all-decomposition-implies-propagated-lits-are-implied*[*OF decomp*] **.**
moreover have *C'*: *?C'* = {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set ?M*}
unfolding *M-K* **by** *standard force*+
ultimately have *N-C-M*: *?N* ∪ *?C'* ⊨*ps unmark-l ?M*
by *auto*
have *N-M-False*: *?N* ∪ (λ*L*. *unmark L*) ‘ (*set ?M*) ⊨*ps* {{#}}
using *M* ‹*?M* ⊨*as CNot C*› ‹*C*∈*?N*› **unfolding** *true-clss-clss-def true-annots-def Ball-def*
true-annot-def **by** (*metis consistent-CNot-not sup.orderE sup-commute true-clss-def*
true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have *undefined-lit F K* **using** ‹*no-dup ?M*› **unfolding** *M-K* **by** (*simp add*: *defined-lit-map*)
moreover
have *?N* ∪ *?C'* ⊨*ps* {{#}}
proof −
have *A*: *?N* ∪ *?C'* ∪ *unmark-l ?M* = *?N* ∪ *unmark-l ?M*
unfolding *M-K* **by** *auto*
show *?thesis*
using *true-clss-clss-left-right*[*OF N-C-M*, *of* {{#}}] *N-M-False* **unfolding** *A* **by** *auto*
qed
have *?N* ⊨*p image-mset uminus ?C* + {#−*K*#}
unfolding *true-clss-cls-def true-clss-clss-def total-over-m-def*
proof (*intro allI impI*)
fix *I*
assume
tot: *total-over-set I* (*atms-of-ms* (*?N* ∪ {*image-mset uminus ?C*+ {#− *K*#}})) **and**
cons: *consistent-interp I* **and**
I ⊨*s ?N*
have (*K* ∈ *I* ∧ −*K* ∉ *I*) ∨ (−*K* ∈ *I* ∧ *K* ∉ *I*)
using *cons tot* **unfolding** *consistent-interp-def* **by** (*cases K*) *auto*
have {*a* ∈ *set* (*trail S*). *is-decided a* ∧ *a* ≠ *Decided K*} =
set (*trail S*) ∩ {*L*. *is-decided L* ∧ *L* ≠ *Decided K*}
by *auto*
then have *tot'*: *total-over-set I*
(*atm-of* ‘ *lit-of* ‘ (*set ?M* ∩ {*L*. *is-decided L* ∧ *L* ≠ *Decided K*}))
using *tot* **by** (*auto simp add*: *atms-of-uminus-lit-atm-of-lit-of*)
{ **fix** *x* :: (*'v*, *unit*) *ann-lit*
assume
a3: *lit-of x* ∉ *I* **and**
a1: *x* ∈ *set ?M* **and**
a4: *is-decided x* **and**
a5: *x* ≠ *Decided K*
then have *Pos* (*atm-of* (*lit-of x*)) ∈ *I* ∨ *Neg* (*atm-of* (*lit-of x*)) ∈ *I*
using *a5 a4 tot' a1* **unfolding** *total-over-set-def atms-of-s-def* **by** *blast*

moreover have *f6*: $Neg\ (atm\text{-}of\ (lit\text{-}of\ x)) = -\ Pos\ (atm\text{-}of\ (lit\text{-}of\ x))$
by *simp*
ultimately have $-\ lit\text{-}of\ x \in I$
using *f6 a3* **by** (*metis* (*no-types*) *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel*(*1*))
} **note** $H = this$

have $\neg I \models s\ ?C'$
using ‹$?N \cup ?C' \models ps\ \{\{\#\}\}$› *tot cons* ‹$I \models s\ ?N$›
unfolding *true-clss-clss-def total-over-m-def*
by (*simp add*: *atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of*)
then show $I \models image\text{-}mset\ uminus\ ?C + \{\#-\ K\#\}$
unfolding *true-clss-def true-cls-def* **using** ‹$(K \in I \land -K \notin I) \lor (-K \in I \land K \notin I)$›
by (*auto dest*!: *H*)
qed
moreover have $F \models as\ CNot\ (image\text{-}mset\ uminus\ ?C)$
using *nm* **unfolding** *true-annots-def CNot-def M-K* **by** (*auto simp add*: *lits-of-def*)
ultimately have *False*
using *bj-can-jump*[*of* $S\ F'\ K\ F\ C\ -K$
image-mset uminus (*image-mset lit-of* $\{\#\ L :\#\ mset\ ?M.\ is\text{-}decided\ L \land L \neq Decided\ K\#\}$)]
‹$C \in ?N$› *n-s* ‹$?M \models as\ CNot\ C$› *bj-backjump inv* ‹*no-dup* (*trail S*)› **unfolding** *M-K* **by** *auto*
then show *?thesis* **by** *fast*
qed *auto*
qed

end — End of *dpll-with-backjumping-ops*

locale *dpll-with-backjumping* =
dpll-with-backjumping-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *inv*
backjump-conds propagate-conds
for
trail :: $'st \Rightarrow ('v,\ unit)\ ann\text{-}lits$ **and**
$clauses_{NOT}$ :: $'st \Rightarrow\ 'v\ clauses$ **and**
prepend-trail :: $('v,\ unit)\ ann\text{-}lit \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v\ clause \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
inv :: $'st \Rightarrow bool$ **and**
backjump-conds :: $'v\ clause \Rightarrow\ 'v\ clause \Rightarrow\ 'v\ literal \Rightarrow\ 'st \Rightarrow\ 'st \Rightarrow bool$ **and**
propagate-conds :: $('v,\ unit)\ ann\text{-}lit \Rightarrow\ 'st \Rightarrow bool$
+
assumes *dpll-bj-inv*: $\bigwedge S\ T.\ dpll\text{-}bj\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
begin

lemma *rtranclp-dpll-bj-inv*:
assumes $dpll\text{-}bj^{**}\ S\ T$ **and** *inv S*
shows *inv T*
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto simp add*: *dpll-bj-no-dup intro*: *dpll-bj-inv*)

lemma *rtranclp-dpll-bj-no-dup*:
assumes $dpll\text{-}bj^{**}\ S\ T$ **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto simp add*: *dpll-bj-no-dup dest*: *rtranclp-dpll-bj-inv dpll-bj-inv*)

lemma *rtranclp-dpll-bj-atms-of-ms-clauses-inv*:
assumes
dpll-bj** *S T* **and** *inv S*
shows *atms-of-mm* (*clauses*$_{NOT}$ *S*) = *atms-of-mm* (*clauses*$_{NOT}$ *T*)
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto dest*: *rtranclp-dpll-bj-inv dpll-bj-atms-of-ms-clauses-inv*)

lemma *rtranclp-dpll-bj-atms-in-trail*:
assumes
dpll-bj** *S T* **and**
inv S **and**
atm-of ' (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
shows *atm-of* ' (*lits-of-l* (*trail T*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *T*)
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
using *dpll-bj-atms-in-trail dpll-bj-atms-of-ms-clauses-inv rtranclp-dpll-bj-inv* **by** *auto*

lemma *rtranclp-dpll-bj-sat-iff*:
assumes *dpll-bj*** *S T* **and** *inv S*
shows *I* ⊨*sm clauses*$_{NOT}$ *S* ⟷ *I* ⊨*sm clauses*$_{NOT}$ *T*
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *dpll-bj-sat-iff simp*: *rtranclp-dpll-bj-inv*)

lemma *rtranclp-dpll-bj-atms-in-trail-in-set*:
assumes
dpll-bj** *S T* **and**
inv S
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ' (*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ' (*lits-of-l* (*trail T*)) ⊆ *A*
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto dest*: *rtranclp-dpll-bj-inv*
simp: *dpll-bj-atms-in-trail-in-set rtranclp-dpll-bj-atms-of-ms-clauses-inv rtranclp-dpll-bj-inv*)

lemma *rtranclp-dpll-bj-all-decomposition-implies-inv*:
assumes
dpll-bj** *S T* **and**
inv S
all-decomposition-implies-m (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*auto intro*: *dpll-bj-all-decomposition-implies-inv simp*: *rtranclp-dpll-bj-inv*)

lemma *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*:
{(*T*, *S*). *dpll-bj*$^{++}$ *S T*
∧ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*) ∧ *inv S*}
⊆ {(*T*, *S*). *dpll-bj S T* ∧ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A*
∧ *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧ *no-dup* (*trail S*) ∧ *inv S*}$^+$
(**is** *?A* ⊆ *?B*$^+$)
proof *standard*
fix *x*
assume *x-A*: *x* ∈ *?A*
obtain *S T*::*'st* **where**
x[*simp*]: *x* = (*T*, *S*) **by** (*cases x*) *auto*
have

dpll-bj^{++} *S T* **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
no-dup (*trail S*) **and**
inv S
using *x-A* **by** *auto*
then show *x* ∈ *?B*$^+$ **unfolding** *x*
proof (*induction rule*: *tranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step T U*) **note** *step* = *this*(*1*) **and** *ST* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*−*7*)]
and *N-A* = *this*(*4*) **and** *M-A* = *this*(*5*) **and** *nd* = *this*(*6*) **and** *inv* = *this*(*7*)

have [*simp*]: *atms-of-mm* (*clauses*$_{NOT}$ *S*) = *atms-of-mm* (*clauses*$_{NOT}$ *T*)
using *step rtranclp-dpll-bj-atms-of-ms-clauses-inv tranclp-into-rtranclp inv* **by** *fastforce*
have *no-dup* (*trail T*)
using *local.step nd rtranclp-dpll-bj-no-dup tranclp-into-rtranclp inv* **by** *fastforce*
moreover have *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *atms-of-ms A*
by (*metis inv M-A N-A local.step rtranclp-dpll-bj-atms-in-trail-in-set*
tranclp-into-rtranclp)
moreover have *inv T*
using *inv local.step rtranclp-dpll-bj-inv tranclp-into-rtranclp* **by** *fastforce*
ultimately have (*U*, *T*) ∈ *?B* **using** *ST N-A M-A inv* **by** *auto*
then show *?case* **using** *IH* **by** (*rule trancl-into-trancl2*)
qed
qed

lemma *wf-tranclp-dpll-bj*:
assumes *fin*: *finite A*
shows *wf* {(*T*, *S*). *dpll-bj*$^{++}$ *S T*
∧ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*) ∧ *inv S*}
using *wf-trancl*[*OF wf-dpll-bj*[*OF fin*]] *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*
by (*rule wf-subset*)

lemma *dpll-bj-sat-ext-iff*:
dpll-bj S T ⟹ *inv S* ⟹ *I*⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I*⊨*sextm clauses*$_{NOT}$ *T*
by (*simp add*: *dpll-bj-clauses*)

lemma *rtranclp-dpll-bj-sat-ext-iff*:
dpll-bj** *S T* ⟹ *inv S* ⟹ *I*⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I*⊨*sextm clauses*$_{NOT}$ *T*
by (*induction rule*: *rtranclp-induct*) (*simp-all add*: *rtranclp-dpll-bj-inv dpll-bj-sat-ext-iff*)

theorem *full-dpll-backjump-final-state*:
fixes *A* :: *'v clause set* **and** *S T* :: *'st*
assumes
full: *full dpll-bj S T* **and**
atms-S: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
∨ (*trail T* ⊨*asm clauses*$_{NOT}$ *S* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))

proof −
  have *st*: *dpll-bj*** *S T* **and** *no-step dpll-bj T*
    using *full* **unfolding** *full-def* **by** *fast+*
  moreover have *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
    using *atms-S inv rtranclp-dpll-bj-atms-of-ms-clauses-inv st* **by** *blast*
  moreover have *atm-of* ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
    using *atms-S atms-trail inv rtranclp-dpll-bj-atms-in-trail-in-set st* **by** *auto*
  moreover have *no-dup* (*trail T*)
    using *n-d inv rtranclp-dpll-bj-no-dup st* **by** *blast*
  moreover have *inv*: *inv T*
    using *inv rtranclp-dpll-bj-inv st* **by** *blast*
  moreover
    have *decomp*: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
      using ‹*inv S*› *decomp rtranclp-dpll-bj-all-decomposition-implies-inv st* **by** *blast*
  ultimately have *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*))
    ∨ (*trail T* ⊨*asm clauses*$_{NOT}$ *T* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*)))
    using ‹*finite A*› *dpll-backjump-final-state* **by** *force*
  then show *?thesis*
    by (*meson* ‹*inv S*› *rtranclp-dpll-bj-sat-iff satisfiable-carac st true-annots-true-cls*)
qed

corollary *full-dpll-backjump-final-state-from-init-state*:
  fixes *A* :: *′v clause set* **and** *S T* :: *′st*
  assumes
    full: *full dpll-bj S T* **and**
    trail S = [] **and**
    clauses$_{NOT}$ *S* = *N* **and**
    inv S
  shows *unsatisfiable* (*set-mset N*) ∨ (*trail T* ⊨*asm N* ∧ *satisfiable* (*set-mset N*))
  using *assms full-dpll-backjump-final-state*[*of S T set-mset N*] **by** *auto*

lemma *tranclp-dpll-bj-trail-mes-decreasing-prop*:
  assumes *dpll*: *dpll-bj*$^{++}$ *S T* **and** *inv*: *inv S* **and**
  N-A: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
  M-A: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
  n-d: *no-dup* (*trail S*) **and**
  fin-A: *finite A*
  shows (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
        − μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*)
      < (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
        − μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)
  using *dpll*
proof (*induction*)
  case *base*
  then show *?case*
    using *N-A M-A n-d dpll-bj-trail-mes-decreasing-prop fin-A inv* **by** *blast*
next
  case (*step T U*) **note** *st* = *this*(*1*) **and** *dpll* = *this*(*2*) **and** *IH* = *this*(*3*)
  have *atms-of-mm* (*clauses*$_{NOT}$ *S*) = *atms-of-mm* (*clauses*$_{NOT}$ *T*)
    using *rtranclp-dpll-bj-atms-of-ms-clauses-inv* **by** (*metis dpll-bj-clauses dpll-bj-inv inv st tranclpD*)
  then have *N-A′*: *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
    using *N-A* **by** *auto*
  moreover have *M-A′*: *atm-of* ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
    by (*meson M-A N-A inv rtranclp-dpll-bj-atms-in-trail-in-set st dpll tranclp.r-into-trancl tranclp-into-rtranclp tranclp-trans*)

moreover have *nd*: *no-dup* (*trail* T)
by (*metis inv n-d rtranclp-dpll-bj-no-dup st tranclp-into-rtranclp*)
moreover have *inv* T
by (*meson dpll dpll-bj-inv inv rtranclp-dpll-bj-inv st tranclp-into-rtranclp*)
ultimately show *?case*
using *IH dpll-bj-trail-mes-decreasing-prop*[*of* T U A] *dpll fin-A* **by** *linarith*
qed

end — End of *dpll-with-backjumping*

5.2.4 CDCL

In this section we will now define the conflict driven clause learning above DPLL: we first introduce the rules learn and forget, and the add these rules to the DPLL calculus.

Learn and Forget

Learning adds a new clause where all the literals are already included in the clauses.

locale *learn-ops* =
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
$clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ +
fixes
learn-cond :: $'v$ *clause* $\Rightarrow 'st \Rightarrow bool$
begin
inductive *learn* :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$learn_{NOT}$*-rule*: $clauses_{NOT}$ S $\models pm$ C $\Longrightarrow$
atms-of $C \subseteq$ *atms-of-mm* ($clauses_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S)) $\Longrightarrow$
learn-cond C S $\Longrightarrow$
$T \sim add\text{-}cls_{NOT}$ C S $\Longrightarrow$
learn S T
inductive-cases $learn_{NOT}E$: *learn* S T

lemma *learn-*μ_C*-stable*:
assumes *learn* S T **and** *no-dup* (*trail* S)
shows μ_C A B (*trail-weight* S) = μ_C A B (*trail-weight* T)
using *assms* **by** (*auto elim*: $learn_{NOT}E$)
end

Forget removes an information that can be deduced from the context (e.g. redundant clauses, tautologies)

locale *forget-ops* =
dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
trail :: $'st \Rightarrow ('v, unit)$ *ann-lits* **and**
$clauses_{NOT}$ :: $'st \Rightarrow 'v$ *clauses* **and**
prepend-trail :: $('v, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**

remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* +
fixes
forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin
inductive *forget*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget$_{NOT}$:
removeAll-mset C(*clauses*$_{NOT}$ *S*) ⊨*pm C* ⟹
forget-cond C S ⟹
C ∈# *clauses*$_{NOT}$ *S* ⟹
T ∼ *remove-cls*$_{NOT}$ *C S* ⟹
forget$_{NOT}$ *S T*
inductive-cases *forget*$_{NOT}$*E*: *forget*$_{NOT}$ *S T*

lemma *forget-*μ_C*-stable*:
assumes *forget*$_{NOT}$ *S T*
shows μ_C *A B* (*trail-weight S*) = μ_C *A B* (*trail-weight T*)
using *assms* **by** (*auto elim*!: *forget*$_{NOT}$*E*)
end

locale *learn-and-forget*$_{NOT}$ =
learn-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *learn-cond* +
forget-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *forget-cond*
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
learn-cond forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin
inductive *learn-and-forget*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool*
where
lf-learn: *learn S T* ⟹ *learn-and-forget*$_{NOT}$ *S T* |
lf-forget: *forget*$_{NOT}$ *S T* ⟹ *learn-and-forget*$_{NOT}$ *S T*
end

Definition of CDCL

locale *conflict-driven-clause-learning-ops* =
dpll-with-backjumping-ops trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds +
learn-and-forget$_{NOT}$ *trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ *learn-cond*
forget-cond
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
learn-cond forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin

inductive $cdcl_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ **for** S :: $'st$ **where**
c-dpll-bj: *dpll-bj* S $S' \Longrightarrow cdcl_{NOT}$ S S' |
c-learn: *learn* S $S' \Longrightarrow cdcl_{NOT}$ S S' |
c-forget$_{NOT}$: *forget*$_{NOT}$ S $S' \Longrightarrow cdcl_{NOT}$ S S'

lemma $cdcl_{NOT}$*-all-induct*[*consumes 1*, *case-names dpll-bj learn forget*$_{NOT}$]:
fixes S T :: $'st$
assumes $cdcl_{NOT}$ S T **and**
dpll: $\bigwedge T$. *dpll-bj* S $T \Longrightarrow P$ S T **and**
learning:
$\bigwedge C$ T. *clauses*$_{NOT}$ $S \models pm$ $C \Longrightarrow$
atms-of $C \subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S)) $\Longrightarrow$
$T \sim$ *add-cls*$_{NOT}$ C $S \Longrightarrow$
P S T **and**
forgetting: $\bigwedge C$ T. *removeAll-mset* C (*clauses*$_{NOT}$ S) $\models pm$ $C \Longrightarrow$
$C \in\#$ *clauses*$_{NOT}$ $S \Longrightarrow$
$T \sim$ *remove-cls*$_{NOT}$ C $S \Longrightarrow$
P S T
shows P S T
using *assms*(*1*) **by** (*induction rule*: $cdcl_{NOT}$*.induct*)
(*auto intro*: *assms*(*2*, *3*, *4*) *elim*!: *learn*$_{NOT}$*E* *forget*$_{NOT}$*E*)+

lemma $cdcl_{NOT}$*-no-dup*:
assumes
$cdcl_{NOT}$ S T **and**
inv S **and**
no-dup (*trail* S)
shows *no-dup* (*trail* T)
using *assms* **by** (*induction rule*: $cdcl_{NOT}$*-all-induct*) (*auto intro*: *dpll-bj-no-dup*)

Consistency of the trail **lemma** $cdcl_{NOT}$*-consistent*:
assumes
$cdcl_{NOT}$ S T **and**
inv S **and**
no-dup (*trail* S)
shows *consistent-interp* (*lits-of-l* (*trail* T))
using $cdcl_{NOT}$*-no-dup*[*OF assms*] *distinct-consistent-interp* **by** *fast*

The subtle problem here is that tautologies can be removed, meaning that some variable can disappear of the problem. It is also means that some variable of the trail might not be present in the clauses anymore.

lemma $cdcl_{NOT}$*-atms-of-ms-clauses-decreasing*:
assumes $cdcl_{NOT}$ S T**and** *inv* S **and** *no-dup* (*trail* S)
shows *atms-of-mm* (*clauses*$_{NOT}$ T) $\subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S))
using *assms* **by** (*induction rule*: $cdcl_{NOT}$*-all-induct*)
(*auto dest*!: *dpll-bj-atms-of-ms-clauses-inv set-mp simp add*: *atms-of-ms-def Union-eq*)

lemma $cdcl_{NOT}$*-atms-in-trail*:
assumes $cdcl_{NOT}$ S T**and** *inv* S **and** *no-dup* (*trail* S)
and *atm-of* ‘ (*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S)
shows *atm-of* ‘ (*lits-of-l* (*trail* T)) $\subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S)
using *assms* **by** (*induction rule*: $cdcl_{NOT}$*-all-induct*) (*auto simp add*: *dpll-bj-atms-in-trail*)

lemma $cdcl_{NOT}$*-atms-in-trail-in-set*:

assumes
$cdcl_{NOT}$ *S T* **and** *inv S* **and** *no-dup* (*trail S*) **and**
atms-of-mm ($clauses_{NOT}$ *S*) ⊆ *A* **and**
atm-of ' (*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ' (*lits-of-l* (*trail T*)) ⊆ *A*
using *assms*
by (*induction rule*: $cdcl_{NOT}$*-all-induct*)
(*simp-all add*: *dpll-bj-atms-in-trail-in-set dpll-bj-atms-of-ms-clauses-inv*)

lemma $cdcl_{NOT}$*-all-decomposition-implies*:
assumes $cdcl_{NOT}$ *S T* **and** *inv S* **and** *n-d*[*simp*]: *no-dup* (*trail S*) **and**
all-decomposition-implies-m ($clauses_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows
all-decomposition-implies-m ($clauses_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
using *assms*(*1*,*2*,*4*)
proof (*induction rule*: $cdcl_{NOT}$*-all-induct*)
case *dpll-bj*
then show *?case*
using *dpll-bj-all-decomposition-implies-inv n-d* **by** *blast*
next
case *learn*
then show *?case* **by** (*auto simp add*: *all-decomposition-implies-def*)
next
case ($forget_{NOT}$ *C T*) **note** *cls-C* = *this*(*1*) **and** *C* = *this*(*2*) **and** *T* = *this*(*3*) **and** *iniv* = *this*(*4*)
and
decomp = *this*(*5*)
show *?case*
unfolding *all-decomposition-implies-def Ball-def*
proof (*intro allI*, *clarify*)
fix *a b*
assume (*a*, *b*) ∈ *set* (*get-all-ann-decomposition* (*trail T*))
then have *unmark-l a* ∪ *set-mset* ($clauses_{NOT}$ *S*) ⊨*ps unmark-l b*
using *decomp T* **by** (*auto simp add*: *all-decomposition-implies-def*)
moreover
have *a1*:*C* ∈ *set-mset* ($clauses_{NOT}$ *S*)
using *C* **by** *blast*
have $clauses_{NOT}$ *T* = $clauses_{NOT}$ ($remove\text{-}cls_{NOT}$ *C S*)
using *T* $state\text{-}eq_{NOT}$*-clauses* **by** *blast*
then have *set-mset* ($clauses_{NOT}$ *T*) ⊨*ps set-mset* ($clauses_{NOT}$ *S*)
using *a1* **by** (*metis* (*no-types*) $clauses\text{-}remove\text{-}cls_{NOT}$ *cls-C insert-Diff order-refl*
set-mset-minus-replicate-mset(*1*) *true-clss-clss-def true-clss-clss-insert*)
ultimately show *unmark-l a* ∪ *set-mset* ($clauses_{NOT}$ *T*)
⊨*ps unmark-l b*
using *true-clss-clss-generalise-true-clss-clss* **by** *blast*
qed
qed

Extension of models **lemma** $cdcl_{NOT}$*-bj-sat-ext-iff*:
assumes $cdcl_{NOT}$ *S T***and** *inv S* **and** *n-d*: *no-dup* (*trail S*)
shows *I*⊨*sextm* $clauses_{NOT}$ *S* ⟷ *I*⊨*sextm* $clauses_{NOT}$ *T*
using *assms*
proof (*induction rule*:$cdcl_{NOT}$*-all-induct*)
case *dpll-bj*
then show *?case* **by** (*simp add*: *dpll-bj-clauses*)
next
case (*learn C T*) **note** *T* = *this*(*3*)

{ **fix** *J*
  assume
    I $\models$*sextm clauses*$_{NOT}$ *S* **and**
    I $\subseteq$ *J* **and**
    tot: *total-over-m J* (*set-mset* ({#*C*#} + *clauses*$_{NOT}$ *S*)) **and**
    cons: *consistent-interp J*
  then have *J* $\models$*sm clauses*$_{NOT}$ *S* **unfolding** *true-clss-ext-def* **by** *auto*

  moreover
    with ‹*clauses*$_{NOT}$ *S*$\models$*pm C*› **have** *J* $\models$ *C*
      using *tot cons* **unfolding** *true-clss-cls-def* **by** *auto*
  ultimately have *J* $\models$*sm* {#*C*#} + *clauses*$_{NOT}$ *S* **by** *auto*
}
then have *H*: *I* $\models$*sextm* (*clauses*$_{NOT}$ *S*) $\Longrightarrow$ *I* $\models$*sext insert C* (*set-mset* (*clauses*$_{NOT}$ *S*))
  unfolding *true-clss-ext-def* **by** *auto*
show *?case*
  apply *standard*
    using *T n-d* **apply** (*auto simp add*: *H*)[]
  using *T n-d* **apply** *simp*
  by (*metis Diff-insert-absorb insert-subset subsetI subset-antisym*
    true-clss-ext-decrease-right-remove-r)
next
case (*forget*$_{NOT}$ *C T*) **note** *cls-C* = *this*(*1*) **and** *T* = *this*(*3*)
{ **fix** *J*
  assume
    I $\models$*sext set-mset* (*clauses*$_{NOT}$ *S*) − {*C*} **and**
    I $\subseteq$ *J* **and**
    tot: *total-over-m J* (*set-mset* (*clauses*$_{NOT}$ *S*)) **and**
    cons: *consistent-interp J*
  then have *J* $\models$*s set-mset* (*clauses*$_{NOT}$ *S*) − {*C*}
    unfolding *true-clss-ext-def* **by** (*meson Diff-subset total-over-m-subset*)

  moreover
    with *cls-C* **have** *J* $\models$ *C*
      using *tot cons* **unfolding** *true-clss-cls-def*
      by (*metis Un-commute forget*$_{NOT}$*.hyps*(*2*) *insert-Diff insert-is-Un order-refl*
        set-mset-minus-replicate-mset(*1*))
  ultimately have *J* $\models$*sm* (*clauses*$_{NOT}$ *S*) **by** (*metis insert-Diff-single true-clss-insert*)
}
then have *H*: *I* $\models$*sext set-mset* (*clauses*$_{NOT}$ *S*) − {*C*} $\Longrightarrow$ *I* $\models$*sextm* (*clauses*$_{NOT}$ *S*)
  unfolding *true-clss-ext-def* **by** *blast*
show *?case* **using** *T* **by** (*auto simp*: *true-clss-ext-decrease-right-remove-r H*)
qed

end — end of *conflict-driven-clause-learning-ops*

CDCL with invariant

locale *conflict-driven-clause-learning* =
  conflict-driven-clause-learning-ops +
  assumes *cdcl*$_{NOT}$*-inv*: $\bigwedge$*S T*. *cdcl*$_{NOT}$ *S T* $\Longrightarrow$ *inv S* $\Longrightarrow$ *inv T*
begin
sublocale *dpll-with-backjumping*
  apply *unfold-locales*
  using *cdcl*$_{NOT}$*.simps cdcl*$_{NOT}$*-inv* **by** *auto*

lemma *rtranclp-cdcl$_{NOT}$-inv*:
cdcl$_{NOT}$** *S T* $\Longrightarrow$ *inv S* $\Longrightarrow$ *inv T*
by (*induction rule*: *rtranclp-induct*) (*auto simp add*: *cdcl$_{NOT}$-inv*)

lemma *rtranclp-cdcl$_{NOT}$-no-dup*:
assumes *cdcl$_{NOT}$*** *S T* **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto intro*: *cdcl$_{NOT}$-no-dup rtranclp-cdcl$_{NOT}$-inv*)

lemma *rtranclp-cdcl$_{NOT}$-trail-clauses-bound*:
assumes
cdcl: *cdcl$_{NOT}$*** *S T* **and**
inv: *inv S* **and**
n-d: *no-dup* (*trail S*) **and**
atms-clauses-S: *atms-of-mm* (*clauses$_{NOT}$ S*) $\subseteq$ *A* **and**
atms-trail-S: *atm-of* ‘(*lits-of-l* (*trail S*)) $\subseteq$ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) $\subseteq$ *A* $\wedge$ *atms-of-mm* (*clauses$_{NOT}$ T*) $\subseteq$ *A*
using *cdcl*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **using** *atms-clauses-S atms-trail-S* **by** *simp*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *cdcl$_{NOT}$* = *this*(*2*) **and** *IH* = *this*(*3*)
have *inv T* **using** *inv st rtranclp-cdcl$_{NOT}$-inv* **by** *blast*
have *no-dup* (*trail T*)
using *rtranclp-cdcl$_{NOT}$-no-dup*[*of S T*] *st cdcl$_{NOT}$ inv n-d* **by** *blast*
then have *atms-of-mm* (*clauses$_{NOT}$ U*) $\subseteq$ *A*
using *cdcl$_{NOT}$-atms-of-ms-clauses-decreasing*[*OF cdcl$_{NOT}$*] *IH n-d* ‹*inv T*› **by** *fast*
moreover
have *atm-of* ‘(*lits-of-l* (*trail U*)) $\subseteq$ *A*
using *cdcl$_{NOT}$-atms-in-trail-in-set*[*OF cdcl$_{NOT}$*, *of A*] ‹*no-dup* (*trail T*)›
by (*meson atms-trail-S atms-clauses-S IH* ‹*inv T*› *cdcl$_{NOT}$*)
ultimately show *?case* **by** *fast*
qed

lemma *rtranclp-cdcl$_{NOT}$-all-decomposition-implies*:
assumes *cdcl$_{NOT}$*** *S T* **and** *inv S* **and** *no-dup* (*trail S*) **and**
all-decomposition-implies-m (*clauses$_{NOT}$ S*) (*get-all-ann-decomposition* (*trail S*))
shows
all-decomposition-implies-m (*clauses$_{NOT}$ T*) (*get-all-ann-decomposition* (*trail T*))
using *assms* **by** (*induction*)
(*auto intro*: *rtranclp-cdcl$_{NOT}$-inv cdcl$_{NOT}$-all-decomposition-implies rtranclp-cdcl$_{NOT}$-no-dup*)

lemma *rtranclp-cdcl$_{NOT}$-bj-sat-ext-iff*:
assumes *cdcl$_{NOT}$*** *S T***and** *inv S* **and** *no-dup* (*trail S*)
shows *I*$\models$*sextm clauses$_{NOT}$ S* $\longleftrightarrow$ *I*$\models$*sextm clauses$_{NOT}$ T*
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
using *cdcl$_{NOT}$-bj-sat-ext-iff* **by** (*auto intro*: *rtranclp-cdcl$_{NOT}$-inv rtranclp-cdcl$_{NOT}$-no-dup*)

definition *cdcl$_{NOT}$-NOT-all-inv* **where**
cdcl$_{NOT}$-NOT-all-inv A S $\longleftrightarrow$ (*finite A* $\wedge$ *inv S* $\wedge$ *atms-of-mm* (*clauses$_{NOT}$ S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* $\wedge$ *no-dup* (*trail S*))

lemma *cdcl$_{NOT}$-NOT-all-inv*:
assumes *cdcl$_{NOT}$*** *S T* **and** *cdcl$_{NOT}$-NOT-all-inv A S*

shows $cdcl_{NOT}$*-NOT-all-inv A T*
using *assms* **unfolding** $cdcl_{NOT}$*-NOT-all-inv-def*
by (*simp add*: *rtranclp-*$cdcl_{NOT}$*-inv rtranclp-*$cdcl_{NOT}$*-no-dup rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*)

abbreviation *learn-or-forget* **where**
learn-or-forget S T $\equiv$ *learn S T* $\vee$ $forget_{NOT}$ *S T*

lemma *rtranclp-learn-or-forget-*$cdcl_{NOT}$:
learn-or-forget** *S T* $\Longrightarrow$ $cdcl_{NOT}{}^{**}$ *S T*
using *rtranclp-mono*[*of learn-or-forget* $cdcl_{NOT}$] **by** (*blast intro*: $cdcl_{NOT}$*.c-learn* $cdcl_{NOT}$*.c-*$forget_{NOT}$)

lemma *learn-or-forget-dpll-*μ_C:
assumes
l-f: *learn-or-forget*** *S T* **and**
dpll: *dpll-bj T U* **and**
inv: $cdcl_{NOT}$*-NOT-all-inv A S*
shows (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight U*)
< (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)
(**is** *?μ U* < *?μ S*)
proof −
have *?μ S* = *?μ T*
using *l-f*
proof (*induction*)
case *base*
then show *?case* **by** *simp*
next
case (*step T U*)
moreover then have *no-dup* (*trail T*)
using *rtranclp-*$cdcl_{NOT}$*-no-dup*[*of S T*] $cdcl_{NOT}$*-NOT-all-inv-def inv*
rtranclp-learn-or-forget-$cdcl_{NOT}$ **by** *auto*
ultimately show *?case*
using *forget-*μ_C*-stable learn-*μ_C*-stable inv* **unfolding** $cdcl_{NOT}$*-NOT-all-inv-def* **by** *presburger*
qed
moreover have $cdcl_{NOT}$*-NOT-all-inv A T*
using *rtranclp-learn-or-forget-*$cdcl_{NOT}$ $cdcl_{NOT}$*-NOT-all-inv l-f inv* **by** *blast*
ultimately show *?thesis*
using *dpll-bj-trail-mes-decreasing-prop*[*of T U A*, *OF dpll*] *finite*
unfolding $cdcl_{NOT}$*-NOT-all-inv-def* **by** *presburger*
qed

lemma *infinite-*$cdcl_{NOT}$*-exists-learn-and-forget-infinite-chain*:
assumes
$\bigwedge$*i*. $cdcl_{NOT}$ (*f i*) (*f*(*Suc i*)) **and**
inv: $cdcl_{NOT}$*-NOT-all-inv A* (*f 0*)
shows ∃*j*. ∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*))
using *assms*
proof (*induction* (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight* (*f 0*))
arbitrary: *f*
rule: *nat-less-induct-case*)
case (*Suc n*) **note** *IH* = *this*(*1*) **and** *μ* = *this*(*2*) **and** $cdcl_{NOT}$ = *this*(*3*) **and** *inv* = *this*(*4*)
consider
(*dpll-end*) ∃*j*. ∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*))

| (*dpll-more*) ¬(∃ *j*. ∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*)))
by *blast*
then show *?case*
proof *cases*
case *dpll-end*
then show *?thesis* **by** *auto*
next
case *dpll-more*
then have *j*: ∃ *i*. ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬*forget*$_{NOT}$ (*f i*) (*f* (*Suc i*))
by *blast*
obtain *i* **where**
¬*learn* (*f i*) (*f* (*Suc i*)) ∧ ¬*forget*$_{NOT}$ (*f i*) (*f* (*Suc i*)) **and**
∀ *k*<*i*. *learn-or-forget* (*f k*) (*f* (*Suc k*))
proof −
obtain i_0 **where** ¬ *learn* (*f* i_0) (*f* (*Suc* i_0)) ∧ ¬*forget*$_{NOT}$ (*f* i_0) (*f* (*Suc* i_0))
using *j* **by** *auto*
then have {*i*. *i*≤i_0 ∧ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬*forget*$_{NOT}$ (*f i*) (*f* (*Suc i*))} ≠ {}
by *auto*
let *?I* = {*i*. *i*≤i_0 ∧ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬*forget*$_{NOT}$ (*f i*) (*f* (*Suc i*))}
let *?i* = *Min ?I*
have *finite ?I*
by *auto*
have ¬ *learn* (*f ?i*) (*f* (*Suc ?i*)) ∧ ¬*forget*$_{NOT}$ (*f ?i*) (*f* (*Suc ?i*))
using *Min-in*[*OF* ‹*finite ?I*› ‹*?I* ≠ {}›] **by** *auto*
moreover have ∀ *k*<*?i*. *learn-or-forget* (*f k*) (*f* (*Suc k*))
using *Min.coboundedI*[*of* {*i*. *i* ≤ i_0 ∧ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬ *forget*$_{NOT}$ (*f i*) (*f* (*Suc i*))}, *simplified*]
by (*meson* ‹¬ *learn* (*f* i_0) (*f* (*Suc* i_0)) ∧ ¬ *forget*$_{NOT}$ (*f* i_0) (*f* (*Suc* i_0))› *less-imp-le dual-order.trans not-le*)
ultimately show *?thesis* **using** *that* **by** *blast*
qed
def *g* ≡ λ*n*. *f* (*n* + *Suc i*)
have *dpll-bj* (*f i*) (*g 0*)
using ‹¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬ *forget*$_{NOT}$ (*f i*) (*f* (*Suc i*))› *cdcl*$_{NOT}$ *cdcl*$_{NOT}$*.cases*
g-def **by** *auto*
{
fix *j*
assume *j* ≤ *i*
then have *learn-or-forget*** (*f 0*) (*f j*)
apply (*induction j*)
apply *simp*
by (*metis* (*no-types*, *lifting*) *Suc-leD Suc-le-lessD rtranclp.simps*
‹∀ *k*<*i*. *learn* (*f k*) (*f* (*Suc k*)) ∨ *forget*$_{NOT}$ (*f k*) (*f* (*Suc k*))›)
}
then have *learn-or-forget*** (*f 0*) (*f i*) **by** *blast*
then have (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight* (*g 0*))
< (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight* (*f 0*))
using *learn-or-forget-dpll-*μ_C[*of f 0 f i g 0 A*] *inv* ‹*dpll-bj* (*f i*) (*g 0*)›
unfolding *cdcl*$_{NOT}$*-NOT-all-inv-def* **by** *linarith*

moreover have *cdcl*$_{NOT}$*-i*: *cdcl*$_{NOT}$** (*f 0*) (*g 0*)
using *rtranclp-learn-or-forget-cdcl*$_{NOT}$[*of f 0 f i*] ‹*learn-or-forget*** (*f 0*) (*f i*)›
cdcl$_{NOT}$[*of i*] **unfolding** *g-def* **by** *auto*
moreover have ⋀*i*. *cdcl*$_{NOT}$ (*g i*) (*g* (*Suc i*))

using $cdcl_{NOT}$ *g-def* **by** *auto*
moreover have $cdcl_{NOT}$*-NOT-all-inv A* (*g 0*)
using *inv* $cdcl_{NOT}$*-i rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound g-def* $cdcl_{NOT}$*-NOT-all-inv* **by** *auto*
ultimately obtain *j* **where** *j*: $\bigwedge i.\ i \geq j \Longrightarrow$ *learn-or-forget* (*g i*) (*g* (*Suc i*))
using *IH* **unfolding** μ[*symmetric*] **by** *presburger*
show *?thesis*
proof
{
fix *k*
assume $k \geq j + Suc\ i$
then have *learn-or-forget* (*f k*) (*f* (*Suc k*))
using *j*[*of k−Suc i*] **unfolding** *g-def* **by** *auto*
}
then show ∀ *k*≥*j*+*Suc i*. *learn-or-forget* (*f k*) (*f* (*Suc k*))
by *auto*
qed
qed
next
case *0* **note** *H* = *this*(*1*) **and** $cdcl_{NOT}$ = *this*(*2*) **and** *inv* = *this*(*3*)
show *?case*
proof (*rule ccontr*)
assume ¬ *?case*
then have *j*: ∃ *i*. ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬$forget_{NOT}$ (*f i*) (*f* (*Suc i*))
by *blast*
obtain *i* **where**
¬*learn* (*f i*) (*f* (*Suc i*)) ∧ ¬$forget_{NOT}$ (*f i*) (*f* (*Suc i*)) **and**
∀ *k*<*i*. *learn-or-forget* (*f k*) (*f* (*Suc k*))
proof −
obtain i_0 **where** ¬ *learn* (*f* i_0) (*f* (*Suc* i_0)) ∧ ¬$forget_{NOT}$ (*f* i_0) (*f* (*Suc* i_0))
using *j* **by** *auto*
then have {*i*. *i*≤i_0 ∧ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬$forget_{NOT}$ (*f i*) (*f* (*Suc i*))} ≠ {}
by *auto*
let *?I* = {*i*. *i*≤i_0 ∧ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬$forget_{NOT}$ (*f i*) (*f* (*Suc i*))}
let *?i* = *Min ?I*
have *finite ?I*
by *auto*
have ¬ *learn* (*f ?i*) (*f* (*Suc ?i*)) ∧ ¬$forget_{NOT}$ (*f ?i*) (*f* (*Suc ?i*))
using *Min-in*[*OF* ‹*finite ?I*› ‹*?I* ≠ {}›] **by** *auto*
moreover have ∀ *k*<*?i*. *learn-or-forget* (*f k*) (*f* (*Suc k*))
using *Min.coboundedI*[*of* {*i*. *i* ≤ i_0 ∧ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬ $forget_{NOT}$ (*f i*) (*f* (*Suc i*))}, *simplified*]
by (*meson* ‹¬ *learn* (*f* i_0) (*f* (*Suc* i_0)) ∧ ¬ $forget_{NOT}$ (*f* i_0) (*f* (*Suc* i_0))› *less-imp-le dual-order.trans not-le*)
ultimately show *?thesis* **using** *that* **by** *blast*
qed
have *dpll-bj* (*f i*) (*f* (*Suc i*))
using ‹¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬ $forget_{NOT}$ (*f i*) (*f* (*Suc i*))› $cdcl_{NOT}$ $cdcl_{NOT}$*.cases*
by *blast*
{
fix *j*
assume $j \leq i$
then have *learn-or-forget*** (*f 0*) (*f j*)
apply (*induction j*)
apply *simp*
by (*metis* (*no-types*, *lifting*) *Suc-leD Suc-le-lessD rtranclp.simps*
‹∀ *k*<*i*. *learn* (*f k*) (*f* (*Suc k*)) ∨ $forget_{NOT}$ (*f k*) (*f* (*Suc k*))›)

}
then have *learn-or-forget*** (*f 0*) (*f i*) **by** *blast*

then show *False*
using *learn-or-forget-dpll-*μ_C[*of f 0 f i f* (*Suc i*) *A*] *inv 0*
‹*dpll-bj* (*f i*) (*f* (*Suc i*))› **unfolding** *cdcl*$_{NOT}$*-NOT-all-inv-def* **by** *linarith*
qed
qed

lemma *wf-cdcl*$_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge f\, j.\ \neg\ (\forall\, i{\ge}j.$ *learn-or-forget* (*f i*) (*f* (*Suc i*)))
shows *wf* {(*T*, *S*). *cdcl*$_{NOT}$ *S T* ∧ *cdcl*$_{NOT}$*-NOT-all-inv A S*}
(**is** *wf* {(*T*, *S*). *cdcl*$_{NOT}$ *S T* ∧ *?inv S*})
unfolding *wf-iff-no-infinite-down-chain*
proof (*rule ccontr*)
assume ¬ ¬ (∃*f*. ∀*i*. (*f* (*Suc i*), *f i*) ∈ {(*T*, *S*). *cdcl*$_{NOT}$ *S T* ∧ *?inv S*})
then obtain *f* **where**
∀*i*. *cdcl*$_{NOT}$ (*f i*) (*f* (*Suc i*)) ∧ *?inv* (*f i*)
by *fast*
then have ∃*j*. ∀*i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*))
using *infinite-cdcl*$_{NOT}$*-exists-learn-and-forget-infinite-chain*[*of f*] **by** *meson*
then show *False* **using** *no-infinite-lf* **by** *blast*
qed

lemma *inv-and-tranclp-cdcl-*$_{NOT}$*-tranclp-cdcl*$_{NOT}$*-and-inv*:
cdcl$_{NOT}$$^{++}$ *S T* ∧ *cdcl*$_{NOT}$*-NOT-all-inv A S* ⟷ (λ*S T*. *cdcl*$_{NOT}$ *S T* ∧ *cdcl*$_{NOT}$*-NOT-all-inv A S*)$^{++}$ *S T*
(**is** *?A* ∧ *?I* ⟷ *?B*)
proof
assume *?A* ∧ *?I*
then have *?A* **and** *?I* **by** *blast+*
then show *?B*
apply *induction*
apply (*simp add*: *tranclp.r-into-trancl*)
by (*subst tranclp.simps*) (*auto intro*: *cdcl*$_{NOT}$*-NOT-all-inv tranclp-into-rtranclp*)
next
assume *?B*
then have *?A* **by** *induction auto*
moreover have *?I* **using** ‹*?B*› *tranclpD* **by** *fastforce*
ultimately show *?A* ∧ *?I* **by** *blast*
qed

lemma *wf-tranclp-cdcl*$_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge f\, j.\ \neg\ (\forall\, i{\ge}j.$ *learn-or-forget* (*f i*) (*f* (*Suc i*)))
shows *wf* {(*T*, *S*). *cdcl*$_{NOT}$$^{++}$ *S T* ∧ *cdcl*$_{NOT}$*-NOT-all-inv A S*}
using *wf-trancl*[*OF wf-cdcl*$_{NOT}$*-no-learn-and-forget-infinite-chain*[*OF no-infinite-lf*]]
apply (*rule wf-subset*)
by (*auto simp*: *trancl-set-tranclp inv-and-tranclp-cdcl-*$_{NOT}$*-tranclp-cdcl*$_{NOT}$*-and-inv*)

lemma *cdcl*$_{NOT}$*-final-state*:
assumes
n-s: *no-step cdcl*$_{NOT}$ *S* **and**
inv: *cdcl*$_{NOT}$*-NOT-all-inv A S* **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))

shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
∨ (*trail S* ⊨*asm clauses*$_{NOT}$ *S* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))
proof −
have *n-s′*: *no-step dpll-bj S*
using *n-s* **by** (*auto simp*: *cdcl*$_{NOT}$*.simps*)
show *?thesis*
apply (*rule dpll-backjump-final-state*[*of S A*])
using *inv decomp n-s′* **unfolding** *cdcl*$_{NOT}$*-NOT-all-inv-def* **by** *auto*
qed

lemma *full-cdcl*$_{NOT}$*-final-state*:
assumes
full: *full cdcl*$_{NOT}$ *S T* **and**
inv: *cdcl*$_{NOT}$*-NOT-all-inv A S* **and**
n-d: *no-dup* (*trail S*) **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*))
∨ (*trail T* ⊨*asm clauses*$_{NOT}$ *T* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*)))
proof −
have *st*: *cdcl*$_{NOT}$** *S T* **and** *n-s*: *no-step cdcl*$_{NOT}$ *T*
using *full* **unfolding** *full-def* **by** *blast+*
have *n-s′*: *cdcl*$_{NOT}$*-NOT-all-inv A T*
using *cdcl*$_{NOT}$*-NOT-all-inv inv st* **by** *blast*
moreover have *all-decomposition-implies-m* (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
using *cdcl*$_{NOT}$*-NOT-all-inv-def decomp inv rtranclp-cdcl*$_{NOT}$*-all-decomposition-implies st* **by** *auto*
ultimately show *?thesis*
using *cdcl*$_{NOT}$*-final-state n-s* **by** *blast*
qed

end — end of *conflict-driven-clause-learning*

Termination

To prove termination we need to restrict learn and forget. Otherwise we could forget and relearn the exact same clause over and over. A first idea is to forbid removing clauses that can be used to backjump. This does not change the rules of the calculus. A second idea is to "merge" backjump and learn: that way, though closer to implementation, needs a change of the rules, since the backjump-rule learns the clause used to backjump.

Restricting learn and forget

locale *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =
dpll-state trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$ +
conflict-driven-clause-learning trail clauses$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds
$\lambda C\ S.$ *distinct-mset C* ∧ ¬*tautology C* ∧ *learn-restrictions C S* ∧
(∃ *F K d F′ C′ L*. *trail S* = *F′* @ *Decided K* # *F* ∧ *C* = *C′* + {#*L*#} ∧ *F* ⊨*as CNot C′*
∧ *C′* + {#*L*#} ∉# *clauses*$_{NOT}$ *S*)
$\lambda C\ S.$ ¬(∃ *F′ F K d L*. *trail S* = *F′* @ *Decided K* # *F* ∧ *F* ⊨*as CNot* (*remove1-mset L C*))
∧ *forget-restrictions C S*
for
trail :: *′st* ⇒ (*′v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *′st* ⇒ *′v clauses* **and**
prepend-trail :: (*′v*, *unit*) *ann-lit* ⇒ *′st* ⇒ *′st* **and**
tl-trail :: *′st* ⇒*′st* **and**

add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
learn-restrictions forget-restrictions :: *'v clause* ⇒ *'st* ⇒ *bool*
begin

lemma *cdcl*$_{NOT}$*-learn-all-induct*[*consumes 1*, *case-names dpll-bj learn forget*$_{NOT}$]:
fixes *S T* :: *'st*
assumes *cdcl*$_{NOT}$ *S T* **and**
dpll: ⋀*T*. *dpll-bj S T* ⟹ *P S T* **and**
learning:
⋀*C F K F' C' L T*. *clauses*$_{NOT}$ *S* ⊨*pm C* ⟹
atms-of C ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*)) ⟹
distinct-mset C ⟹
¬ *tautology C* ⟹
learn-restrictions C S ⟹
trail S = *F'* @ *Decided K* # *F* ⟹
C = *C'* + {#*L*#} ⟹
F ⊨*as CNot C'* ⟹
C' + {#*L*#} ∉# *clauses*$_{NOT}$ *S* ⟹
T ∼ *add-cls*$_{NOT}$ *C S* ⟹
P S T **and**
forgetting: ⋀*C T*. *removeAll-mset C* (*clauses*$_{NOT}$ *S*) ⊨*pm C* ⟹
C ∈# *clauses*$_{NOT}$ *S* ⟹
¬(∃ *F' F K L*. *trail S* = *F'* @ *Decided K* # *F* ∧ *F* ⊨*as CNot* (*C* − {#*L*#})) ⟹
T ∼ *remove-cls*$_{NOT}$ *C S* ⟹
forget-restrictions C S ⟹
P S T
shows *P S T*
using *assms*(*1*)
apply (*induction rule*: *cdcl*$_{NOT}$*.induct*)
apply (*auto dest*: *assms*(*2*) *simp add*: *learn-ops-axioms*)[]
apply (*auto elim*!: *learn-ops.learn.cases*[*OF learn-ops-axioms*] *dest*: *assms*(*3*))[]
apply (*auto elim*!: *forget-ops.forget*$_{NOT}$*.cases*[*OF forget-ops-axioms*] *dest*!: *assms*(*4*))
done

lemma *rtranclp-cdcl*$_{NOT}$*-inv*:
cdcl$_{NOT}$** *S T* ⟹ *inv S* ⟹ *inv T*
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
using *cdcl*$_{NOT}$*-inv* **unfolding** *conflict-driven-clause-learning-def*
conflict-driven-clause-learning-axioms-def **by** *blast*

lemma *learn-always-simple-clauses*:
assumes
learn: *learn S T* **and**
n-d: *no-dup* (*trail S*)
shows *set-mset* (*clauses*$_{NOT}$ *T* − *clauses*$_{NOT}$ *S*)
⊆ *simple-clss* (*atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ *lits-of-l* (*trail S*))
proof
fix *C* **assume** *C*: *C* ∈ *set-mset* (*clauses*$_{NOT}$ *T* − *clauses*$_{NOT}$ *S*)
have *distinct-mset C* ¬*tautology C* **using** *learn C n-d* **by** (*elim learn*$_{NOT}$*E*; *auto*)+
then have *C* ∈ *simple-clss* (*atms-of C*)
using *distinct-mset-not-tautology-implies-in-simple-clss* **by** *blast*

moreover have *atms-of* $C \subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ' *lits-of-l* (*trail* S)
using *learn* C *n-d* **by** (*elim learn*$_{NOT}E$) (*auto simp*: *atms-of-ms-def atms-of-def image-Un true-annots-CNot-all-atms-defined*)
moreover have *finite* (*atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ' *lits-of-l* (*trail* S))
by *auto*
ultimately show $C \in$ *simple-clss* (*atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ' *lits-of-l* (*trail* S))
using *simple-clss-mono* **by** (*metis* (*no-types*) *insert-subset mk-disjoint-insert*)
qed

definition *conflicting-bj-clss* $S \equiv$
$\{C+\{\#L\#\}$ $|C$ $L.$ $C+\{\#L\#\} \in\#$ *clauses*$_{NOT}$ $S \wedge$ *distinct-mset* $(C+\{\#L\#\})$
$\wedge \neg$*tautology* $(C+\{\#L\#\})$
$\wedge$ ($\exists F'$ K F. *trail* $S = F'$ @ *Decided* K # $F \wedge F \models$*as CNot* C)}

lemma *conflicting-bj-clss-remove-cls*$_{NOT}$[*simp*]:
conflicting-bj-clss (*remove-cls*$_{NOT}$ C S) = *conflicting-bj-clss* $S - \{C\}$
unfolding *conflicting-bj-clss-def* **by** *fastforce*

lemma *conflicting-bj-clss-remove-cls*$_{NOT}$'[*simp*]:
$T \sim$ *remove-cls*$_{NOT}$ C $S \Longrightarrow$ *conflicting-bj-clss* T = *conflicting-bj-clss* $S - \{C\}$
unfolding *conflicting-bj-clss-def* **by** *fastforce*

lemma *conflicting-bj-clss-add-cls*$_{NOT}$*-state-eq*:
assumes
T: $T \sim$ *add-cls*$_{NOT}$ C' S **and**
n-d: *no-dup* (*trail* S)
shows *conflicting-bj-clss* T
= *conflicting-bj-clss* S
$\cup$ (*if* $\exists C$ $L.$ $C' = C$ $+\{\#L\#\} \wedge$ *distinct-mset* $(C+\{\#L\#\}) \wedge \neg$*tautology* $(C+\{\#L\#\})$
$\wedge$ ($\exists F'$ K d F. *trail* $S = F'$ @ *Decided* K # $F \wedge F \models$*as CNot* C)
then $\{C'\}$ *else* $\{\}$)
proof −
def $P \equiv \lambda C$ L T. *distinct-mset* $(C + \{\#L\#\}) \wedge \neg$ *tautology* $(C + \{\#L\#\}) \wedge$
($\exists F'$ K F. *trail* $T = F'$ @ *Decided* K # $F \wedge F \models$*as CNot* C)
have *conf*: $\bigwedge T$. *conflicting-bj-clss* $T = \{C + \{\#L\#\}$ $|C$ $L.$ $C + \{\#L\#\} \in\#$ *clauses*$_{NOT}$ $T \wedge P$ C L $T\}$
unfolding *conflicting-bj-clss-def P-def* **by** *auto*
have *P-S-T*: $\bigwedge C$ $L.$ P C L $T = P$ C L S
using T *n-d* **unfolding** *P-def* **by** *auto*
have P: *conflicting-bj-clss* $T = \{C + \{\#L\#\}$ $|C$ $L.$ $C + \{\#L\#\} \in\#$ *clauses*$_{NOT}$ $S \wedge P$ C L $T\} \cup$
$\{C + \{\#L\#\}$ $|C$ $L.$ $C + \{\#L\#\} \in\# \{\#C'\#\} \wedge P$ C L $T\}$
using T *n-d* **unfolding** *conf* **by** *auto*
moreover have $\{C + \{\#L\#\}$ $|C$ $L.$ $C + \{\#L\#\} \in\#$ *clauses*$_{NOT}$ $S \wedge P$ C L $T\}$ = *conflicting-bj-clss* S
using T *n-d* **unfolding** *P-def conflicting-bj-clss-def* **by** *auto*
moreover have $\{C + \{\#L\#\}$ $|C$ $L.$ $C + \{\#L\#\} \in\# \{\#C'\#\} \wedge P$ C L $T\}$ =
(*if* $\exists C$ $L.$ $C' = C$ $+\{\#L\#\} \wedge P$ C L S *then* $\{C'\}$ *else* $\{\}$)
using *n-d* T **by** (*force simp*: *P-S-T*)
ultimately show *?thesis* **unfolding** *P-def* **by** *presburger*
qed

lemma *conflicting-bj-clss-add-cls*$_{NOT}$:
no-dup (*trail* S) $\Longrightarrow$
conflicting-bj-clss (*add-cls*$_{NOT}$ C' S)
= *conflicting-bj-clss* S
$\cup$ (*if* $\exists C$ $L.$ $C' = C$ $+\{\#L\#\} \wedge$ *distinct-mset* $(C+\{\#L\#\}) \wedge \neg$*tautology* $(C+\{\#L\#\})$

$\wedge$ ($\exists F'$ K d F. *trail* S = F' @ *Decided* K # F $\wedge$ F $\models$*as* *CNot* C)
then $\{C'\}$ *else* $\{\}$)
using *conflicting-bj-clss-add-cls$_{NOT}$-state-eq* **by** *auto*

lemma *conflicting-bj-clss-incl-clauses*:
conflicting-bj-clss $S \subseteq$ *set-mset* (*clauses$_{NOT}$* S)
unfolding *conflicting-bj-clss-def* **by** *auto*

lemma *finite-conflicting-bj-clss*[*simp*]:
finite (*conflicting-bj-clss* S)
using *conflicting-bj-clss-incl-clauses*[*of* S] *rev-finite-subset* **by** *blast*

lemma *learn-conflicting-increasing*:
no-dup (*trail* S) $\Longrightarrow$ *learn* S T $\Longrightarrow$ *conflicting-bj-clss* $S \subseteq$ *conflicting-bj-clss* T
apply (*elim learn$_{NOT}$E*)
by (*subst conflicting-bj-clss-add-cls$_{NOT}$-state-eq*[*of* T]) *auto*

abbreviation *conflicting-bj-clss-yet* b S $\equiv$
3 ^ b − *card* (*conflicting-bj-clss* S)

abbreviation μ_L :: *nat* $\Rightarrow$ *'st* $\Rightarrow$ *nat* $\times$ *nat* **where**
μ_L b S $\equiv$ (*conflicting-bj-clss-yet* b S, *card* (*set-mset* (*clauses$_{NOT}$* S)))

lemma *remove1-mset-single-add-if*:
remove1-mset L (C + {#L'#}) = (*if* $L = L'$ *then* C *else remove1-mset* L C + {#L'#})
by (*auto simp*: *multiset-eq-iff*)

lemma *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*:
assumes *forget$_{NOT}$* S T
shows *conflicting-bj-clss* S = *conflicting-bj-clss* T
using *assms* **apply** (*elim forget$_{NOT}$E*)
apply *rule*
apply (*subst conflicting-bj-clss-remove-cls$_{NOT}$'*[*of* T], *simp*)
apply (*fastforce simp*: *conflicting-bj-clss-def remove1-mset-single-add-if split*: *if-splits*)
apply *fastforce*
done

lemma *forget-μ_L-decrease*:
assumes *forget$_{NOT}$*: *forget$_{NOT}$* S T
shows (μ_L b T, μ_L b S) $\in$ *less-than* <*lex*> *less-than*
proof −
have *card* (*set-mset* (*clauses$_{NOT}$* S)) > *0*
using *forget$_{NOT}$* **by** (*elim forget$_{NOT}$E*) (*auto simp*: *size-mset-removeAll-mset-le-iff card-gt-0-iff*)
then have *card* (*set-mset* (*clauses$_{NOT}$* T)) < *card* (*set-mset* (*clauses$_{NOT}$* S))
using *forget$_{NOT}$* **by** (*elim forget$_{NOT}$E*) (*auto simp*: *size-mset-removeAll-mset-le-iff*)
then show *?thesis*
unfolding *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*[*OF forget$_{NOT}$*]
by *auto*
qed

lemma *set-condition-or-split*:
{a. (a = b $\vee$ Q a) $\wedge$ S a} = (*if* S b *then* {b} *else* {}) $\cup$ {a. Q a $\wedge$ S a}
by *auto*

lemma *set-insert-neq*:
$A \neq$ *insert* a A $\longleftrightarrow$ $a \notin A$

by *auto*

lemma *learn-μ_L-decrease*:
assumes *learnST*: *learn S T* **and** *n-d*: *no-dup* (*trail S*) **and**
A: *atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ' *lits-of-l* (*trail S*) ⊆ *A* **and**
fin-A: *finite A*
shows (μ_L (*card A*) *T*, μ_L (*card A*) *S*) ∈ *less-than* <∗*lex*∗> *less-than*
proof −
have [*simp*]: (*atms-of-mm* (*clauses$_{NOT}$ T*) ∪ *atm-of* ' *lits-of-l* (*trail T*))
= (*atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ' *lits-of-l* (*trail S*))
using *learnST n-d* **by** (*elim learn$_{NOT}$E*) *auto*

then have *card* (*atms-of-mm* (*clauses$_{NOT}$ T*) ∪ *atm-of* ' *lits-of-l* (*trail T*))
= *card* (*atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ' *lits-of-l* (*trail S*))
by (*auto intro*!: *card-mono*)
then have *3*: (*3*::*nat*) ^ *card* (*atms-of-mm* (*clauses$_{NOT}$ T*) ∪ *atm-of* ' *lits-of-l* (*trail T*))
= *3* ^ *card* (*atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ' *lits-of-l* (*trail S*))
by (*auto intro*: *power-mono*)
moreover have *conflicting-bj-clss S* ⊆ *conflicting-bj-clss T*
using *learnST n-d* **by** (*simp add*: *learn-conflicting-increasing*)
moreover have *conflicting-bj-clss S* ≠ *conflicting-bj-clss T*
using *learnST*
proof (*elim learn$_{NOT}$E*, *goal-cases*)
case (*1 C*) **note** *clss-S* = *this*(*1*) **and** *atms-C* = *this*(*2*) **and** *inv* = *this*(*3*) **and** *T* = *this*(*4*)
then obtain *F K F′ C′ L* **where**
tr-S: *trail S* = *F′* @ *Decided K* # *F* **and**
C: *C* = *C′* + {#*L*#} **and**
F: *F* ⊨*as CNot C′* **and**
C-S:*C′* + {#*L*#} ∉# *clauses$_{NOT}$ S*
by *blast*
moreover have *distinct-mset C* ¬ *tautology C* **using** *inv* **by** *blast*+
ultimately have *C′* + {#*L*#} ∈ *conflicting-bj-clss T*
using *T n-d* **unfolding** *conflicting-bj-clss-def* **by** *fastforce*
moreover have *C′* + {#*L*#} ∉ *conflicting-bj-clss S*
using *C-S* **unfolding** *conflicting-bj-clss-def* **by** *auto*
ultimately show *?case* **by** *blast*
qed
moreover have *fin-T*: *finite* (*conflicting-bj-clss T*)
using *learnST* **by** *induction* (*auto simp add*: *conflicting-bj-clss-add-cls$_{NOT}$*)
ultimately have *card* (*conflicting-bj-clss T*) ≥ *card* (*conflicting-bj-clss S*)
using *card-mono* **by** *blast*

moreover
have *fin′*: *finite* (*atms-of-mm* (*clauses$_{NOT}$ T*) ∪ *atm-of* ' *lits-of-l* (*trail T*))
by *auto*
have *1*:*atms-of-ms* (*conflicting-bj-clss T*) ⊆ *atms-of-mm* (*clauses$_{NOT}$ T*)
unfolding *conflicting-bj-clss-def atms-of-ms-def* **by** *auto*
have *2*: ⋀*x*. *x*∈ *conflicting-bj-clss T* ⟹ ¬ *tautology x* ∧ *distinct-mset x*
unfolding *conflicting-bj-clss-def* **by** *auto*
have *T*: *conflicting-bj-clss T*
⊆ *simple-clss* (*atms-of-mm* (*clauses$_{NOT}$ T*) ∪ *atm-of* ' *lits-of-l* (*trail T*))
by *standard* (*meson 1 2 fin′* ‹*finite* (*conflicting-bj-clss T*)› *simple-clss-mono*
distinct-mset-set-def simplified-in-simple-clss subsetCE sup.coboundedI1)
moreover
then have #: *3* ^ *card* (*atms-of-mm* (*clauses$_{NOT}$ T*) ∪ *atm-of* ' *lits-of-l* (*trail T*))
≥ *card* (*conflicting-bj-clss T*)

by (*meson Nat.le-trans simple-clss-card simple-clss-finite card-mono fin'*)
have *atms-of-mm* ($clauses_{NOT}$ *T*) ∪ *atm-of* ' *lits-of-l* (*trail T*) ⊆ *A*
using $learn_{NOT}E$[*OF learnST*] *A* **by** *simp*
then have *3* ^ (*card A*) ≥ *card* (*conflicting-bj-clss T*)
using # *fin-A* **by** (*meson simple-clss-card simple-clss-finite*
simple-clss-mono calculation(*2*) *card-mono dual-order.trans*)
ultimately show *?thesis*
using *psubset-card-mono*[*OF fin-T*]
unfolding *less-than-iff lex-prod-def* **by** *clarify*
(*meson* ⟨*conflicting-bj-clss S* ≠ *conflicting-bj-clss T*⟩
⟨*conflicting-bj-clss S* ⊆ *conflicting-bj-clss T*⟩
diff-less-mono2 le-less-trans not-le psubsetI)
qed

We have to assume the following:

- *inv S*: the invariant holds in the inital state.
- *A* is a (finite *finite A*) superset of the literals in the trail *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* and in the clauses *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A*. This can the the set of all the literals in the starting set of clauses.
- *no-dup* (*trail S*): no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} **where**
μ_{CDCL} *A T* ≡ ((*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight T*),
conflicting-bj-clss-yet (*card* (*atms-of-ms A*)) *T*, *card* (*set-mset* ($clauses_{NOT}$ *T*)))
lemma $cdcl_{NOT}$*-decreasing-measure*:
assumes
$cdcl_{NOT}$ *S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-lits: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows (μ_{CDCL} *A T*, μ_{CDCL} *A S*)
∈ *less-than* <∗*lex*∗> (*less-than* <∗*lex*∗> *less-than*)
using *assms*(*1*)
proof *induction*
case (*c-dpll-bj T*)
from *dpll-bj-trail-mes-decreasing-prop*[*OF this*(*1*) *inv atm-clss atm-lits n-d fin-A*]
show *?case* **unfolding** μ_{CDCL}*-def*
by (*meson in-lex-prod less-than-iff*)
next
case (*c-learn T*) **note** *learn* = *this*(*1*)
then have *S*: *trail S* = *trail T*
using *inv atm-clss atm-lits n-d fin-A*
by (*elim* $learn_{NOT}E$) *auto*
show *?case*
using *learn-*μ_L*-decrease*[*OF learn n-d*, *of atms-of-ms A*] *atm-clss atm-lits fin-A n-d*
unfolding *S* μ_{CDCL}*-def* **by** *auto*
next
case (*c-*$forget_{NOT}$ *T*) **note** $forget_{NOT}$ = *this*(*1*)
have *trail S* = *trail T* **using** $forget_{NOT}$ **by** *induction auto*

then show *?case*
using *forget-μ_L-decrease*[*OF forget$_{NOT}$*] **unfolding** *μ_{CDCL}-def* **by** *auto*
qed

lemma *wf-cdcl$_{NOT}$-restricted-learning*:
assumes *finite A*
shows *wf* {(*T*, *S*).
(*atms-of-mm* (*clauses$_{NOT}$ S*) $\subseteq$ *atms-of-ms A* $\wedge$ *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *no-dup* (*trail S*)
$\wedge$ *inv S*)
$\wedge$ *cdcl$_{NOT}$ S T* }
by (*rule wf-wf-if-measure'*[*of less-than* <*lex*> (*less-than* <*lex*> *less-than*)])
(*auto intro*: *cdcl$_{NOT}$-decreasing-measure*[*OF - - - - - assms*])

definition $\mu_C{}'$ :: *'v clause set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_C{}'$ *A T* $\equiv$ μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight T*)

definition $\mu_{CDCL}{}'$ :: *'v clause set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_{CDCL}{}'$ *A T* $\equiv$
((*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A T*) $*$ (*1+ 3^card* (*atms-of-ms A*)) $*$
2
$+$ *conflicting-bj-clss-yet* (*card* (*atms-of-ms A*)) *T* $*$ *2*
$+$ *card* (*set-mset* (*clauses$_{NOT}$ T*))

lemma *cdcl$_{NOT}$-decreasing-measure'*:
assumes
cdcl$_{NOT}$ S T **and**
inv: *inv S* **and**
atms-clss: *atms-of-mm* (*clauses$_{NOT}$ S*) $\subseteq$ *atms-of-ms A* **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows $\mu_{CDCL}{}'$ *A T* $<$ $\mu_{CDCL}{}'$ *A S*
using *assms*(*1*)
proof (*induction rule*: *cdcl$_{NOT}$-learn-all-induct*)
case (*dpll-bj T*)
then have (*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A T*
$<$ (*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A S*
using *dpll-bj-trail-mes-decreasing-prop fin-A inv n-d atms-clss atms-trail*
unfolding *$\mu_C{}'$-def* **by** *blast*
then have *XX*: ((*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A T*) + *1*
$\leq$ (*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A S*
by *auto*
from *mult-le-mono1*[*OF this*, *of 1 + 3 ^ card* (*atms-of-ms A*)]
have ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A T*) $*$
(*1* + *3* ^ *card* (*atms-of-ms A*)) + (*1* + *3* ^ *card* (*atms-of-ms A*))
$\leq$ ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A S*)
$*$ (*1* + *3* ^ *card* (*atms-of-ms A*))
unfolding *Nat.add-mult-distrib*
by *presburger*
moreover
have *cl-T-S*: *clauses$_{NOT}$ T* = *clauses$_{NOT}$ S*
using *dpll-bj.hyps inv dpll-bj-clauses* **by** *auto*
have *conflicting-bj-clss-yet* (*card* (*atms-of-ms A*)) *S* $<$ *1+ 3* ^ *card* (*atms-of-ms A*)
by *simp*
ultimately have ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A T*)

$* (1 + 3 \,\hat{}\, card\ (atms\text{-}of\text{-}ms\ A)) + conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ T$
$< ((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ S) * (1 + 3 \,\hat{}\, card\ (atms\text{-}of\text{-}ms\ A))$
by *linarith*
then have $((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ T)$
$* (1 + 3 \,\hat{}\, card\ (atms\text{-}of\text{-}ms\ A))$
$+ conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ T$
$< ((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ S)$
$* (1 + 3 \,\hat{}\, card\ (atms\text{-}of\text{-}ms\ A))$
$+ conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ S$
by *linarith*
then have $((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ T)$
$* (1 + 3 \,\hat{}\, card\ (atms\text{-}of\text{-}ms\ A)) * 2$
$+ conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ T * 2$
$< ((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ S)$
$* (1 + 3 \,\hat{}\, card\ (atms\text{-}of\text{-}ms\ A)) * 2$
$+ conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ S * 2$
by *linarith*
then show *?case* **unfolding** $\mu_{CDCL}{}'$*-def cl-T-S* **by** *presburger*
next
case $(learn\ C\ F'\ K\ F\ C'\ L\ T)$ **note** $clss\text{-}S\text{-}C = this(1)$ **and** $atms\text{-}C = this(2)$ **and** $dist = this(3)$
and $tauto = this(4)$ **and** $learn\text{-}restr = this(5)$ **and** $tr\text{-}S = this(6)$ **and** $C' = this(7)$ **and**
$F\text{-}C = this(8)$ **and** $C\text{-}new = this(9)$ **and** $T = this(10)$
have $insert\ C\ (conflicting\text{-}bj\text{-}clss\ S) \subseteq simple\text{-}clss\ (atms\text{-}of\text{-}ms\ A)$
proof −
have $C \in simple\text{-}clss\ (atms\text{-}of\text{-}ms\ A)$
using C'
by (*metis* (*no-types*, *hide-lams*) *Un-subset-iff simple-clss-mono*
contra-subsetD dist distinct-mset-not-tautology-implies-in-simple-clss
dual-order.trans atms-C atms-clss atms-trail tauto)
moreover have $conflicting\text{-}bj\text{-}clss\ S \subseteq simple\text{-}clss\ (atms\text{-}of\text{-}ms\ A)$
proof
fix $x :: 'v\ clause$
assume $x \in conflicting\text{-}bj\text{-}clss\ S$
then have $x \in\#\ clauses_{NOT}\ S \wedge distinct\text{-}mset\ x \wedge \neg\ tautology\ x$
unfolding *conflicting-bj-clss-def* **by** *blast*
then show $x \in simple\text{-}clss\ (atms\text{-}of\text{-}ms\ A)$
by (*meson atms-clss atms-of-atms-of-ms-mono atms-of-ms-finite simple-clss-mono*
distinct-mset-not-tautology-implies-in-simple-clss fin-A finite-subset
set-rev-mp)
qed
ultimately show *?thesis*
by *auto*
qed
then have $card\ (insert\ C\ (conflicting\text{-}bj\text{-}clss\ S)) \leq 3\ \hat{}\ (card\ (atms\text{-}of\text{-}ms\ A))$
by (*meson Nat.le-trans atms-of-ms-finite simple-clss-card simple-clss-finite*
card-mono fin-A)
moreover have [*simp*]: $card\ (insert\ C\ (conflicting\text{-}bj\text{-}clss\ S))$
$= Suc\ (card\ ((conflicting\text{-}bj\text{-}clss\ S)))$
by (*metis* (*no-types*) C' *C-new card-insert-if conflicting-bj-clss-incl-clauses contra-subsetD*
finite-conflicting-bj-clss)
moreover have [*simp*]: $conflicting\text{-}bj\text{-}clss\ (add\text{-}cls_{NOT}\ C\ S) = conflicting\text{-}bj\text{-}clss\ S \cup \{C\}$
using *dist tauto F-C* **by** (*subst conflicting-bj-clss-add-cls*$_{NOT}$[*OF n-d*]) (*force simp*: C' *tr-S n-d*)
ultimately have [*simp*]: $conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ S$
$= Suc\ (conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ (add\text{-}cls_{NOT}\ C\ S))$
by *simp*

have *1*: $clauses_{NOT}$ T = $clauses_{NOT}$ (*add-cls*$_{NOT}$ C S) **using** T **by** *auto*
have *2*: *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) T
= *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) (*add-cls*$_{NOT}$ C S)
using T **unfolding** *conflicting-bj-clss-def* **by** *auto*
have *3*: $\mu_C{}'$ A T = $\mu_C{}'$ A (*add-cls*$_{NOT}$ C S)
using T **unfolding** $\mu_C{}'$*-def* **by** *auto*
have ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − $\mu_C{}'$ A (*add-cls*$_{NOT}$ C S))
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) ∗ *2*
= ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − $\mu_C{}'$ A S)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) ∗ *2*
using *n-d* **unfolding** $\mu_C{}'$*-def* **by** *auto*
moreover
have *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) (*add-cls*$_{NOT}$ C S)
∗ *2*
+ *card* (*set-mset* ($clauses_{NOT}$ (*add-cls*$_{NOT}$ C S)))
< *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) S ∗ *2*
+ *card* (*set-mset* ($clauses_{NOT}$ S))
by (*simp add*: C' *C-new n-d*)
ultimately show *?case* **unfolding** $\mu_{CDCL}{}'$*-def* *1 2 3* **by** *presburger*
next
case (*forget*$_{NOT}$ C T) **note** T = *this*(*4*)
have [*simp*]: $\mu_C{}'$ A (*remove-cls*$_{NOT}$ C S) = $\mu_C{}'$ A S
unfolding $\mu_C{}'$*-def* **by** *auto*
have *forget*$_{NOT}$ S T
apply (*rule forget*$_{NOT}$*.intros*) **using** *forget*$_{NOT}$ **by** *auto*
then have *conflicting-bj-clss* T = *conflicting-bj-clss* S
using *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched* **by** *blast*
moreover have *card* (*set-mset* ($clauses_{NOT}$ T)) < *card* (*set-mset* ($clauses_{NOT}$ S))
by (*metis* T *card-Diff1-less clauses-remove-cls*$_{NOT}$ *finite-set-mset forget*$_{NOT}$*.hyps*(*2*)
order-refl set-mset-minus-replicate-mset(*1*) *state-eq*$_{NOT}$*-clauses*)
ultimately show *?case* **unfolding** $\mu_{CDCL}{}'$*-def*
using T ‹$\mu_C{}'$ A (*remove-cls*$_{NOT}$ C S) = $\mu_C{}'$ A S› **by** (*metis* (*no-types*) *add-le-cancel-left*
$\mu_C{}'$*-def not-le state-eq*$_{NOT}$*-trail*)
qed

lemma *cdcl*$_{NOT}$*-clauses-bound*:
assumes
cdcl$_{NOT}$ S T **and**
inv S **and**
atms-of-mm ($clauses_{NOT}$ S) ⊆ A **and**
atm-of ‘(*lits-of-l* (*trail* S)) ⊆ A **and**
n-d: *no-dup* (*trail* S) **and**
fin-A[*simp*]: *finite* A
shows *set-mset* ($clauses_{NOT}$ T) ⊆ *set-mset* ($clauses_{NOT}$ S) ∪ *simple-clss* A
using *assms*
proof (*induction rule*: *cdcl*$_{NOT}$*-learn-all-induct*)
case *dpll-bj*
then show *?case* **using** *dpll-bj-clauses* **by** *simp*
next
case *forget*$_{NOT}$
then show *?case* **using** *clauses-remove-cls*$_{NOT}$ **unfolding** *state-eq*$_{NOT}$*-def* **by** *auto*
next
case (*learn* C F K d F' C' L) **note** *atms-C* = *this*(*2*) **and** *dist* = *this*(*3*) **and** *tauto* = *this*(*4*) **and**
T = *this*(*10*) **and** *atms-clss-S* = *this*(*12*) **and** *atms-trail-S* = *this*(*13*)
have *atms-of* C ⊆ A
using *atms-C atms-clss-S atms-trail-S* **by** *fast*

then have *simple-clss* (*atms-of* C) $\subseteq$ *simple-clss* A
  by (*simp add*: *simple-clss-mono*)
then have $C \in$ *simple-clss* A
  using *finite dist tauto* **by** (*auto dest*: *distinct-mset-not-tautology-implies-in-simple-clss*)
then show *?case* **using** T *n-d* **by** *auto*
qed

lemma *rtranclp-cdcl*$_{NOT}$*-clauses-bound*:
assumes
  cdcl$_{NOT}$** S T **and**
  inv S **and**
  atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq A$ **and**
  atm-of ‘(*lits-of-l* (*trail* S)) $\subseteq A$ **and**
  n-d: *no-dup* (*trail* S) **and**
  finite: *finite* A
shows *set-mset* (*clauses*$_{NOT}$ T) $\subseteq$ *set-mset* (*clauses*$_{NOT}$ S) $\cup$ *simple-clss* A
using *assms*($1-5$)
proof *induction*
case *base*
then show *?case* **by** *simp*
next
case (*step* T U) **note** *st* = *this*(1) **and** *cdcl*$_{NOT}$ = *this*(2) **and** *IH* = *this*(3)[*OF this*($4-7$)] **and**
  inv = *this*(4) **and** *atms-clss-S* = *this*(5) **and** *atms-trail-S* = *this*(6) **and** *finite-cls-S* = *this*(7)
have *inv* T
  using *rtranclp-cdcl*$_{NOT}$*-inv st inv* **by** *blast*
moreover have *atms-of-mm* (*clauses*$_{NOT}$ T) $\subseteq A$ **and** *atm-of* ‘ *lits-of-l* (*trail* T) $\subseteq A$
  using *rtranclp-cdcl*$_{NOT}$*-trail-clauses-bound*[*OF st*] *inv atms-clss-S atms-trail-S n-d* **by** *auto*
moreover have *no-dup* (*trail* T)
 using *rtranclp-cdcl*$_{NOT}$*-no-dup*[*OF st* ‹*inv* S› *n-d*] **by** *simp*
ultimately have *set-mset* (*clauses*$_{NOT}$ U) $\subseteq$ *set-mset* (*clauses*$_{NOT}$ T) $\cup$ *simple-clss* A
  using *cdcl*$_{NOT}$ *finite n-d* **by** (*auto simp*: *cdcl*$_{NOT}$*-clauses-bound*)
then show *?case* **using** *IH* **by** *auto*
qed

lemma *rtranclp-cdcl*$_{NOT}$*-card-clauses-bound*:
assumes
  cdcl$_{NOT}$** S T **and**
  inv S **and**
  atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq A$ **and**
  atm-of ‘(*lits-of-l* (*trail* S)) $\subseteq A$ **and**
  n-d: *no-dup* (*trail* S) **and**
  finite: *finite* A
shows *card* (*set-mset* (*clauses*$_{NOT}$ T)) $\leq$ *card* (*set-mset* (*clauses*$_{NOT}$ S)) + 3 ^ (*card* A)
using *rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] *finite* **by** (*meson Nat.le-trans*
  simple-clss-card simple-clss-finite card-Un-le card-mono finite-UnI
  finite-set-mset nat-add-left-cancel-le)

lemma *rtranclp-cdcl*$_{NOT}$*-card-clauses-bound'*:
assumes
  cdcl$_{NOT}$** S T **and**
  inv S **and**
  atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq A$ **and**
  atm-of ‘(*lits-of-l* (*trail* S)) $\subseteq A$ **and**
  n-d: *no-dup* (*trail* S) **and**
  finite: *finite* A
shows *card* $\{C|C.\ C \in\#$ *clauses*$_{NOT}$ $T \wedge$ (*tautology* $C \vee \neg$*distinct-mset* C)$\}$

$\leq$ *card* $\{C|C.\ C\in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ (card\ A)$
(**is** *card ?T* $\leq$ *card ?S* + -)
using *rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] *finite*
proof –
have *?T* $\subseteq$ *?S* $\cup$ *simple-clss A*
using *rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] **by** *force*
then have *card ?T* $\leq$ *card* (*?S* $\cup$ *simple-clss A*)
using *finite* **by** (*simp add*: *assms*(*5*) *simple-clss-finite card-mono*)
then show *?thesis*
by (*meson le-trans simple-clss-card card-Un-le local.finite nat-add-left-cancel-le*)
qed

lemma *rtranclp-cdcl*$_{NOT}$*-card-simple-clauses-bound*:
assumes
cdcl$_{NOT}{}^{**}$ *S T* **and**
inv S **and**
NA: *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *A* **and**
MA: *atm-of* ‘ (*lits-of-l* (*trail S*)) $\subseteq$ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* (*set-mset* (*clauses*$_{NOT}$ *T*))
$\leq$ *card* $\{C.\ C \in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ (card\ A)$
(**is** *card ?T* $\leq$ *card ?S* + -)
using *rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] *finite*
proof –
have $\bigwedge x.\ x \in\#\ clauses_{NOT}\ T \Longrightarrow \neg\ tautology\ x \Longrightarrow distinct\text{-}mset\ x \Longrightarrow x \in simple\text{-}clss\ A$
using *rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] **by** (*metis* (*no-types*, *hide-lams*) *Un-iff NA*
atms-of-atms-of-ms-mono simple-clss-mono contra-subsetD subset-trans
distinct-mset-not-tautology-implies-in-simple-clss)
then have *set-mset* (*clauses*$_{NOT}$ *T*) $\subseteq$ *?S* $\cup$ *simple-clss A*
using *rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] **by** *auto*
then have *card*(*set-mset* (*clauses*$_{NOT}$ *T*)) $\leq$ *card* (*?S* $\cup$ *simple-clss A*)
using *finite* **by** (*simp add*: *assms*(*5*) *simple-clss-finite card-mono*)
then show *?thesis*
by (*meson le-trans simple-clss-card card-Un-le local.finite nat-add-left-cancel-le*)
qed

definition $\mu_{CDCL}{}'$*-bound* :: *'v clause set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_{CDCL}{}'$*-bound A S* =
$((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A))) * (1 + 3\ \hat{}\ card\ (atms\text{-}of\text{-}ms\ A)) * 2$
$+ 2*3\ \hat{}\ (card\ (atms\text{-}of\text{-}ms\ A))$
$+ card\ \{C.\ C \in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ (card\ (atms\text{-}of\text{-}ms\ A))$

lemma $\mu_{CDCL}{}'$*-bound-reduce-trail-to*$_{NOT}$[*simp*]:
$\mu_{CDCL}{}'$*-bound A* (*reduce-trail-to*$_{NOT}$ *M S*) = $\mu_{CDCL}{}'$*-bound A S*
unfolding $\mu_{CDCL}{}'$*-bound-def* **by** *auto*

lemma *rtranclp-cdcl*$_{NOT}$*-*$\mu_{CDCL}{}'$*-bound-reduce-trail-to*$_{NOT}$:
assumes
cdcl$_{NOT}{}^{**}$ *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite* (*atms-of-ms A*) **and**
U: *U* $\sim$ *reduce-trail-to*$_{NOT}$ *M T*

shows μ_{CDCL}' *A U* $\leq$ μ_{CDCL}'*-bound A S*
proof −
have ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) − μ_C' *A U*)
$\leq$ (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
by *auto*
then have ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) − μ_C' *A U*)
∗ (*1* + *3* ^ *card* (*atms-of-ms A*)) ∗ *2*
$\leq$ (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) ∗ (*1* + *3* ^ *card* (*atms-of-ms A*)) ∗ *2*
using *mult-le-mono1* **by** *blast*
moreover
have *conflicting-bj-clss-yet* (*card* (*atms-of-ms A*)) *T* ∗ *2* $\leq$ *2* ∗ *3* ^ *card* (*atms-of-ms A*)
by *linarith*
moreover have *card* (*set-mset* (*clauses*$_{NOT}$ *U*))
$\leq$ *card* {*C*. *C* ∈# *clauses*$_{NOT}$ *S* ∧ (*tautology C* ∨ ¬*distinct-mset C*)} + *3* ^ *card* (*atms-of-ms A*)
using *rtranclp-cdcl*$_{NOT}$*-card-simple-clauses-bound*[*OF assms*(*1*−*6*)] *U* **by** *auto*
ultimately show *?thesis*
unfolding μ_{CDCL}'*-def* μ_{CDCL}'*-bound-def* **by** *linarith*
qed

lemma *rtranclp-cdcl*$_{NOT}$-μ_{CDCL}'*-bound*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite* (*atms-of-ms A*)
shows μ_{CDCL}' *A T* $\leq$ μ_{CDCL}'*-bound A S*
proof −
have μ_{CDCL}' *A* (*reduce-trail-to*$_{NOT}$ (*trail T*) *T*) = μ_{CDCL}' *A T*
unfolding μ_{CDCL}'*-def* μ_C'*-def conflicting-bj-clss-def* **by** *auto*
then show *?thesis* **using** *rtranclp-cdcl*$_{NOT}$-μ_{CDCL}'*-bound-reduce-trail-to*$_{NOT}$[*OF assms*, *of - trail T*]
state-eq$_{NOT}$*-ref* **by** *fastforce*
qed

lemma *rtranclp*-μ_{CDCL}'*-bound-decreasing*:
assumes
cdcl$_{NOT}$** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite[*simp*]: *finite* (*atms-of-ms A*)
shows μ_{CDCL}'*-bound A T* $\leq$ μ_{CDCL}'*-bound A S*
proof −
have {*C*. *C* ∈# *clauses*$_{NOT}$ *T* ∧ (*tautology C* ∨ ¬ *distinct-mset C*)}
⊆ {*C*. *C* ∈# *clauses*$_{NOT}$ *S* ∧ (*tautology C* ∨ ¬ *distinct-mset C*)} (**is** *?T* ⊆ *?S*)
proof (*rule Set.subsetI*)
fix *C* **assume** *C* ∈ *?T*
then have *C-T*: *C* ∈# *clauses*$_{NOT}$ *T* **and** *t-d*: *tautology C* ∨ ¬ *distinct-mset C*
by *auto*
then have *C* ∉ *simple-clss* (*atms-of-ms A*)
by (*auto dest*: *simple-clssE*)
then show *C* ∈ *?S*
using *C-T rtranclp-cdcl*$_{NOT}$*-clauses-bound*[*OF assms*] *t-d* **by** *force*
qed

then have *card* $\{C.\ C \in\#\ clauses_{NOT}\ T \wedge (tautology\ C \vee \neg\ distinct\text{-}mset\ C)\} \leq$
card $\{C.\ C \in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg\ distinct\text{-}mset\ C)\}$
by (*simp add*: *card-mono*)
then show *?thesis*
unfolding μ_{CDCL}*'-bound-def* **by** *auto*
qed

end — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt*

5.2.5 CDCL with restarts

Definition

locale *restart-ops* =
fixes
$cdcl_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **and**
restart :: *'st* ⇒ *'st* ⇒ *bool*
begin
inductive $cdcl_{NOT}$*-raw-restart* :: *'st* ⇒ *'st* ⇒ *bool* **where**
$cdcl_{NOT}$ *S T* ⟹ $cdcl_{NOT}$*-raw-restart S T* |
restart S T ⟹ $cdcl_{NOT}$*-raw-restart S T*

end

locale *conflict-driven-clause-learning-with-restarts* =
conflict-driven-clause-learning trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds learn-cond forget-cond
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
learn-cond forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool*
begin

lemma $cdcl_{NOT}$*-iff-*$cdcl_{NOT}$*-raw-restart-no-restarts*:
$cdcl_{NOT}$ *S T* ⟷ *restart-ops.*$cdcl_{NOT}$*-raw-restart* $cdcl_{NOT}$ (λ- -. *False*) *S T*
(**is** *?C S T* ⟷ *?R S T*)
proof
fix *S T*
assume *?C S T*
then show *?R S T* **by** (*simp add*: *restart-ops.*$cdcl_{NOT}$*-raw-restart.intros*(*1*))
next
fix *S T*
assume *?R S T*
then show *?C S T*
apply (*cases rule*: *restart-ops.*$cdcl_{NOT}$*-raw-restart.cases*)
using ‹*?R S T*› **by** *fast+*
qed

lemma $cdcl_{NOT}$*-*$cdcl_{NOT}$*-raw-restart*:

$cdcl_{NOT}$ S $T \Longrightarrow$ *restart-ops.*$cdcl_{NOT}$*-raw-restart* $cdcl_{NOT}$ *restart* S T
by (*simp add*: *restart-ops.*$cdcl_{NOT}$*-raw-restart.intros*(*1*))
end

Increasing restarts

To add restarts we needs some assumptions on the predicate (called $cdcl_{NOT}$ here):

- a function f that is strictly monotonic. The first step is actually only used as a restart to clean the state (e.g. to ensure that the trail is empty). Then we assume that $(1::'a) \leq f\ n$ for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...

- a measure μ: it should decrease under the assumptions *bound-inv*, whenever a $cdcl_{NOT}$ or a *restart* is done. A parameter is given to μ: for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.

- we also assume that the measure decrease after any $cdcl_{NOT}$ step.

- an invariant on the states $cdcl_{NOT}$*-inv* that also holds after restarts.

- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ*-bound* taking the same parameter as μ and the initial state of the considered $cdcl_{NOT}$ chain.

locale $cdcl_{NOT}$*-increasing-restarts-ops* =
  restart-ops $cdcl_{NOT}$ *restart* **for**
    restart :: $'st \Rightarrow 'st \Rightarrow bool$ **and**
    $cdcl_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ +
  fixes
    f :: $nat \Rightarrow nat$ **and**
    bound-inv :: $'bound \Rightarrow 'st \Rightarrow bool$ **and**
    μ :: $'bound \Rightarrow 'st \Rightarrow nat$ **and**
    $cdcl_{NOT}$*-inv* :: $'st \Rightarrow bool$ **and**
    μ*-bound* :: $'bound \Rightarrow 'st \Rightarrow nat$
  assumes
    f: *unbounded* f **and**
    f-ge-1:$\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \neq 0$ **and**
    bound-inv: $\bigwedge A\ S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ *bound-inv* A $S \Longrightarrow$ $cdcl_{NOT}$ S $T \Longrightarrow$ *bound-inv* A T **and**
    $cdcl_{NOT}$*-measure*: $\bigwedge A\ S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ *bound-inv* A $S \Longrightarrow$ $cdcl_{NOT}$ S $T \Longrightarrow \mu\ A\ T < \mu$
A S **and**
    measure-bound2: $\bigwedge A\ T\ U.$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* A $T \Longrightarrow$ $cdcl_{NOT}^{**}$ T U
      $\Longrightarrow \mu\ A\ U \leq \mu$*-bound* A T **and**
    measure-bound4: $\bigwedge A\ T\ U.$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* A $T \Longrightarrow$ $cdcl_{NOT}^{**}$ T U
      $\Longrightarrow \mu$*-bound* A $U \leq \mu$*-bound* A T **and**
    $cdcl_{NOT}$*-restart-inv*: $\bigwedge A\ U\ V.$ $cdcl_{NOT}$*-inv* $U \Longrightarrow$ *restart* U $V \Longrightarrow$ *bound-inv* A $U \Longrightarrow$ *bound-inv*
A V
      and
    exists-bound: $\bigwedge R\ S.$ $cdcl_{NOT}$*-inv* $R \Longrightarrow$ *restart* R $S \Longrightarrow \exists A.$ *bound-inv* A S **and**
    $cdcl_{NOT}$*-inv*: $\bigwedge S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ $cdcl_{NOT}$ S $T \Longrightarrow$ $cdcl_{NOT}$*-inv* T **and**
    $cdcl_{NOT}$*-inv-restart*: $\bigwedge S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ *restart* S $T \Longrightarrow$ $cdcl_{NOT}$*-inv* T
begin

lemma $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*:

assumes
$(cdcl_{NOT}\widehat{\ }\widehat{\ }n)$ S T **and**
$cdcl_{NOT}$*-inv* S
shows $cdcl_{NOT}$*-inv* T
using *assms* **by** (*induction n arbitrary*: T) (*auto intro*:*bound-inv* $cdcl_{NOT}$*-inv*)

lemma $cdcl_{NOT}$*-bound-inv*:
assumes
$(cdcl_{NOT}\widehat{\ }\widehat{\ }n)$ S T **and**
$cdcl_{NOT}$*-inv* S
bound-inv A S
shows *bound-inv* A T
using *assms* **by** (*induction n arbitrary*: T) (*auto intro*:*bound-inv* $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*)

lemma *rtranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*:
assumes
$cdcl_{NOT}{}^{**}$ S T **and**
$cdcl_{NOT}$*-inv* S
shows $cdcl_{NOT}$*-inv* T
using *assms* **by** *induction* (*auto intro*: $cdcl_{NOT}$*-inv*)

lemma *rtranclp-*$cdcl_{NOT}$*-bound-inv*:
assumes
$cdcl_{NOT}{}^{**}$ S T **and**
bound-inv A S **and**
$cdcl_{NOT}$*-inv* S
shows *bound-inv* A T
using *assms* **by** *induction* (*auto intro*:*bound-inv rtranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*)

lemma $cdcl_{NOT}$*-comp-n-le*:
assumes
$(cdcl_{NOT}\widehat{\ }\widehat{\ }(Suc\ n))$ S T **and**
bound-inv A S
$cdcl_{NOT}$*-inv* S
shows $\mu\ A\ T < \mu\ A\ S - n$
using *assms*
proof (*induction n arbitrary*: T)
case *0*
then show *?case* **using** $cdcl_{NOT}$*-measure* **by** *auto*
next
case (*Suc n*) **note** *IH* = *this*(*1*)[*OF - this*(*3*) *this*(*4*)] **and** *S-T* = *this*(*2*) **and** *b-inv* = *this*(*3*) **and** *c-inv* = *this*(*4*)
obtain U :: *'st* **where** *S-U*: $(cdcl_{NOT}\widehat{\ }\widehat{\ }(Suc\ n))$ S U **and** *U-T*: $cdcl_{NOT}$ U T **using** *S-T* **by** *auto*
then have $\mu\ A\ U < \mu\ A\ S - n$ **using** *IH*[*of U*] **by** *simp*
moreover
have *bound-inv* A U
using *S-U b-inv* $cdcl_{NOT}$*-bound-inv c-inv* **by** *blast*
then have $\mu\ A\ T < \mu\ A\ U$ **using** $cdcl_{NOT}$*-measure*[*OF - - U-T*] *S-U c-inv* $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*
by *auto*
ultimately show *?case* **by** *linarith*
qed

lemma *wf-*$cdcl_{NOT}$:
wf $\{(T, S).\ cdcl_{NOT}\ S\ T \wedge cdcl_{NOT}\text{-}inv\ S \wedge bound\text{-}inv\ A\ S\}$ (**is** *wf ?A*)
apply (*rule wfP-if-measure2*[*of - -* μ *A*])
using $cdcl_{NOT}$*-comp-n-le*[*of 0 - - A*] **by** *auto*

lemma *rtranclp-cdcl*$_{NOT}$*-measure*:
assumes
cdcl$_{NOT}$** *S T* **and**
bound-inv A S **and**
cdcl$_{NOT}$*-inv S*
shows μ *A T* $\leq$ μ *A S*
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step T U*) **note** *IH* = *this*(*3*)[*OF this*(*4*) *this*(*5*)] **and** *st* = *this*(*1*) **and** *cdcl*$_{NOT}$ = *this*(*2*)
and
b-inv = *this*(*4*) **and** *c-inv* = *this*(*5*)
have *bound-inv A T*
by (*meson cdcl*$_{NOT}$*-bound-inv rtranclp-imp-relpowp st step.prems*)
moreover have *cdcl*$_{NOT}$*-inv T*
using *c-inv rtranclp-cdcl*$_{NOT}$*-cdcl*$_{NOT}$*-inv st* **by** *blast*
ultimately have μ *A U* < μ *A T* **using** *cdcl*$_{NOT}$*-measure*[*OF - - cdcl*$_{NOT}$] **by** *auto*
then show *?case* **using** *IH* **by** *linarith*
qed

lemma *cdcl*$_{NOT}$*-comp-bounded*:
assumes
bound-inv A S **and** *cdcl*$_{NOT}$*-inv S* **and** $m \geq 1+\mu$ *A S*
shows ¬(*cdcl*$_{NOT}$ ^^ *m*) *S T*
using *assms cdcl*$_{NOT}$*-comp-n-le*[*of m−1 S T A*] **by** *fastforce*

- *f n* < *m* ensures that at least one step has been done.

inductive *cdcl*$_{NOT}$*-restart* **where**
restart-step: (*cdcl*$_{NOT}$^^*m*) *S T* ⟹ $m \geq f\ n$ ⟹ *restart T U*
⟹ *cdcl*$_{NOT}$*-restart* (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1 cdcl*$_{NOT}$ *S T* ⟹ *cdcl*$_{NOT}$*-restart* (*S*, *n*) (*T*, *Suc n*)

lemmas *cdcl*$_{NOT}$*-with-restart-induct* = *cdcl*$_{NOT}$*-restart.induct*[*split-format*(*complete*),
OF cdcl$_{NOT}$*-increasing-restarts-ops-axioms*]

lemma *cdcl*$_{NOT}$*-restart-cdcl*$_{NOT}$*-raw-restart*:
cdcl$_{NOT}$*-restart S T* ⟹ *cdcl*$_{NOT}$*-raw-restart*** (*fst S*) (*fst T*)
proof (*induction rule*: *cdcl*$_{NOT}$*-restart.induct*)
case (*restart-step m S T n U*)
then have *cdcl*$_{NOT}$** *S T* **by** (*meson relpowp-imp-rtranclp*)
then have *cdcl*$_{NOT}$*-raw-restart*** *S T* **using** *cdcl*$_{NOT}$*-raw-restart.intros*(*1*)
rtranclp-mono[*of cdcl*$_{NOT}$ *cdcl*$_{NOT}$*-raw-restart*] **by** *blast*
moreover have *cdcl*$_{NOT}$*-raw-restart T U*
using ‹*restart T U*› *cdcl*$_{NOT}$*-raw-restart.intros*(*2*) **by** *blast*
ultimately show *?case* **by** *auto*
next
case (*restart-full S T*)
then have *cdcl*$_{NOT}$** *S T* **unfolding** *full1-def* **by** *auto*
then show *?case* **using** *cdcl*$_{NOT}$*-raw-restart.intros*(*1*)
rtranclp-mono[*of cdcl*$_{NOT}$ *cdcl*$_{NOT}$*-raw-restart*] **by** *auto*
qed

lemma $cdcl_{NOT}$*-with-restart-bound-inv*:
  assumes
    $cdcl_{NOT}$*-restart* S T **and**
    bound-inv A (*fst* S) **and**
    $cdcl_{NOT}$*-inv* (*fst* S)
  shows *bound-inv* A (*fst* T)
  using *assms* **apply** (*induction rule*: $cdcl_{NOT}$*-restart.induct*)
    prefer *2* **apply** (*metis rtranclp-unfold fstI full1-def rtranclp-*$cdcl_{NOT}$*-bound-inv*)
  by (*metis* $cdcl_{NOT}$*-bound-inv* $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv* $cdcl_{NOT}$*-restart-inv fst-conv*)

lemma $cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*:
  assumes
    $cdcl_{NOT}$*-restart* S T **and**
    $cdcl_{NOT}$*-inv* (*fst* S)
  shows $cdcl_{NOT}$*-inv* (*fst* T)
  using *assms* **apply** *induction*
    apply (*metis* $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv* $cdcl_{NOT}$*-inv-restart fst-conv*)
   apply (*metis fstI full-def full-unfold rtranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*)
  done

lemma *rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*:
  assumes
    $cdcl_{NOT}$*-restart*** S T **and**
    $cdcl_{NOT}$*-inv* (*fst* S)
  shows $cdcl_{NOT}$*-inv* (*fst* T)
  using *assms* **by** *induction* (*auto intro*: $cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*)

lemma *rtranclp-*$cdcl_{NOT}$*-with-restart-bound-inv*:
  assumes
    $cdcl_{NOT}$*-restart*** S T **and**
    $cdcl_{NOT}$*-inv* (*fst* S) **and**
    bound-inv A (*fst* S)
  shows *bound-inv* A (*fst* T)
  using *assms* **apply** *induction*
   apply (*simp add*: $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv* $cdcl_{NOT}$*-with-restart-bound-inv*)
  using $cdcl_{NOT}$*-with-restart-bound-inv rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv* **by** *blast*

lemma $cdcl_{NOT}$*-with-restart-increasing-number*:
  $cdcl_{NOT}$*-restart* S $T \Longrightarrow$ *snd* T = *1* + *snd* S
  by (*induction rule*: $cdcl_{NOT}$*-restart.induct*) *auto*
end

locale $cdcl_{NOT}$*-increasing-restarts* =
  $cdcl_{NOT}$*-increasing-restarts-ops restart* $cdcl_{NOT}$ *f bound-inv* μ $cdcl_{NOT}$*-inv* μ*-bound* +
  dpll-state trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  for
    trail :: *'st* $\Rightarrow$ (*'v*, *unit*) *ann-lits* **and**
    $clauses_{NOT}$ :: *'st* $\Rightarrow$ *'v clauses* **and**
    prepend-trail :: (*'v*, *unit*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    tl-trail :: *'st* $\Rightarrow$*'st* **and**
    $add\text{-}cls_{NOT}$ :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    $remove\text{-}cls_{NOT}$ :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    f :: *nat* $\Rightarrow$ *nat* **and**
    restart :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
    bound-inv :: *'bound* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**

μ :: *'bound* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **and**
$cdcl_{NOT}$ :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
$cdcl_{NOT}$*-inv* :: *'st* $\Rightarrow$ *bool* **and**
μ*-bound* :: *'bound* $\Rightarrow$ *'st* $\Rightarrow$ *nat* +
assumes
measure-bound: $\bigwedge A\ T\ V\ n.$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow$ $cdcl_{NOT}$*-restart* (T, n) $(V, Suc\ n) \Longrightarrow \mu\ A\ V \leq \mu$*-bound* $A\ T$ **and**
$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}$*-restart* (T, a) $(V, b) \Longrightarrow cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu$*-bound* $A\ V \leq \mu$*-bound* $A\ T$
begin

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}$*-restart*** (T, a) $(V, b) \Longrightarrow cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu$*-bound* $A\ V \leq \mu$*-bound* $A\ T$
apply (*induction rule*: *rtranclp-induct2*)
apply *simp*
by (*metis* $cdcl_{NOT}$*-raw-restart-*μ*-bound dual-order.trans fst-conv*
rtranclp-$cdcl_{NOT}$*-with-restart-bound-inv rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*)

lemma $cdcl_{NOT}$*-raw-restart-measure-bound*:
$cdcl_{NOT}$*-restart* (T, a) $(V, b) \Longrightarrow cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu$*-bound* $A\ T$
apply (*cases rule*: $cdcl_{NOT}$*-restart.cases*)
apply *simp*
using *measure-bound relpowp-imp-rtranclp* **apply** *fastforce*
by (*metis full-def full-unfold measure-bound2 prod.inject*)

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-measure-bound*:
$cdcl_{NOT}$*-restart*** (T, a) $(V, b) \Longrightarrow cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu$*-bound* $A\ T$
apply (*induction rule*: *rtranclp-induct2*)
apply (*simp add*: *measure-bound2*)
by (*metis dual-order.trans fst-conv measure-bound2 r-into-rtranclp rtranclp.rtrancl-refl*
rtranclp-$cdcl_{NOT}$*-with-restart-bound-inv rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*
rtranclp-$cdcl_{NOT}$*-raw-restart-*μ*-bound*)

lemma *wf-*$cdcl_{NOT}$*-restart*:
wf $\{(T, S).$ $cdcl_{NOT}$*-restart* $S\ T \wedge cdcl_{NOT}$*-inv* $(fst\ S)\}$ (**is** *wf ?A*)
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then obtain g **where**
g: $\bigwedge i.$ $cdcl_{NOT}$*-restart* $(g\ i)$ $(g\ (Suc\ i))$ **and**
$cdcl_{NOT}$*-inv-g*: $\bigwedge i.$ $cdcl_{NOT}$*-inv* $(fst\ (g\ i))$
unfolding *wf-iff-no-infinite-down-chain* **by** *fast*

have *snd-g*: $\bigwedge i.$ $snd\ (g\ i) = i + snd\ (g\ 0)$
apply (*induct-tac i*)
apply *simp*
by (*metis Suc-eq-plus1-left add.commute add.left-commute*
$cdcl_{NOT}$*-with-restart-increasing-number g*)
then have *snd-g-0*: $\bigwedge i.$ $i > 0 \Longrightarrow snd\ (g\ i) = i + snd\ (g\ 0)$
by *blast*
have *unbounded-f-g*: *unbounded* $(\lambda i.\ f\ (snd\ (g\ i)))$
using f **unfolding** *bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*
not-bounded-nat-exists-larger not-le le-iff-add)

{ **fix** *i*
have *H*: $\bigwedge T\ Ta\ m.\ (cdcl_{NOT} \ \widehat{}\widehat{}\ m)\ T\ Ta \Longrightarrow$ *no-step* $cdcl_{NOT}\ T \Longrightarrow m = 0$
apply (*case-tac m*) **by** *simp* (*meson relpowp-E2*)
have $\exists\ T\ m.\ (cdcl_{NOT} \ \widehat{}\widehat{}\ m)\ (fst\ (g\ i))\ T \wedge m \geq f\ (snd\ (g\ i))$
using *g*[*of i*] **apply** (*cases rule*: $cdcl_{NOT}$*-restart.cases*)
apply *auto*[]
using *g*[*of Suc i*] *f-ge-1* **apply** (*cases rule*: $cdcl_{NOT}$*-restart.cases*)
apply (*auto simp add*: *full1-def full-def dest*: *H dest*: *tranclpD*)
using *H Suc-leI leD* **by** *blast*
} **note** *H* = *this*
obtain *A* **where** *bound-inv A* (*fst* (*g 1*))
using *g*[*of 0*] $cdcl_{NOT}$*-inv-g*[*of 0*] **apply** (*cases rule*: $cdcl_{NOT}$*-restart.cases*)
apply (*metis One-nat-def* $cdcl_{NOT}$*-inv exists-bound fst-conv relpowp-imp-rtranclp rtranclp-induct*)
using *H*[*of 1*] **unfolding** *full1-def* **by** (*metis One-nat-def Suc-eq-plus1 diff-is-0-eq′ diff-zero f-ge-1 fst-conv le-add2 relpowp-E2 snd-conv*)
let $?j = \mu$*-bound A* (*fst* (*g 1*)) + *1*
obtain *j* **where**
j: $f\ (snd\ (g\ j)) > ?j$ **and** $j > 1$
using *unbounded-f-g not-bounded-nat-exists-larger* **by** *blast*
{
fix *i j*
have $cdcl_{NOT}$*-with-restart*: $j \geq i \Longrightarrow cdcl_{NOT}\text{-}restart^{**}\ (g\ i)\ (g\ j)$
apply (*induction j*)
apply *simp*
by (*metis g le-Suc-eq rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl*)
} **note** $cdcl_{NOT}$*-restart* = *this*
have $cdcl_{NOT}$*-inv* (*fst* (*g* (*Suc 0*)))
by (*simp add*: $cdcl_{NOT}$*-inv-g*)
have $cdcl_{NOT}\text{-}restart^{**}$ (*fst* (*g 1*), *snd* (*g 1*)) (*fst* (*g j*), *snd* (*g j*))
using ‹*j*> *1*› **by** (*simp add*: $cdcl_{NOT}$*-restart*)
have μ *A* (*fst* (*g j*)) $\leq \mu$*-bound A* (*fst* (*g 1*))
apply (*rule rtranclp*-$cdcl_{NOT}$*-raw-restart-measure-bound*)
using ‹$cdcl_{NOT}\text{-}restart^{**}$ (*fst* (*g 1*), *snd* (*g 1*)) (*fst* (*g j*), *snd* (*g j*))› **apply** *blast*
apply (*simp add*: $cdcl_{NOT}$*-inv-g*)
using ‹*bound-inv A* (*fst* (*g 1*))› **apply** *simp*
done
then have μ *A* (*fst* (*g j*)) $\leq ?j$
by *auto*
have *inv*: *bound-inv A* (*fst* (*g j*))
using ‹*bound-inv A* (*fst* (*g 1*))› ‹$cdcl_{NOT}$*-inv* (*fst* (*g* (*Suc 0*)))›
‹$cdcl_{NOT}\text{-}restart^{**}$ (*fst* (*g 1*), *snd* (*g 1*)) (*fst* (*g j*), *snd* (*g j*))›
rtranclp-$cdcl_{NOT}$*-with-restart-bound-inv* **by** *auto*
obtain *T m* **where**
$cdcl_{NOT}$*-m*: $(cdcl_{NOT} \ \widehat{}\widehat{}\ m)$ (*fst* (*g j*)) *T* **and**
f-m: $f\ (snd\ (g\ j)) \leq m$
using *H*[*of j*] **by** *blast*
have $?j < m$
using *f-m j Nat.le-trans* **by** *linarith*

then show *False*
using ‹μ *A* (*fst* (*g j*)) $\leq \mu$*-bound A* (*fst* (*g 1*))›
$cdcl_{NOT}$*-comp-bounded*[*OF inv* $cdcl_{NOT}$*-inv-g*, *of*] $cdcl_{NOT}$*-inv-g* $cdcl_{NOT}$*-m*
‹$?j < m$› **by** *auto*
qed

lemma *cdcl*$_{NOT}$*-restart-steps-bigger-than-bound*:
  assumes
    cdcl$_{NOT}$*-restart S T* **and**
    bound-inv A (*fst S*) **and**
    cdcl$_{NOT}$*-inv* (*fst S*) **and**
    f (*snd S*) > *μ-bound A* (*fst S*)
  shows *full1 cdcl*$_{NOT}$ (*fst S*) (*fst T*)
  using *assms*
proof (*induction rule*: *cdcl*$_{NOT}$*-restart.induct*)
  case *restart-full*
  then show *?case* **by** *auto*
next
  case (*restart-step m S T n U*) **note** *st* = *this*(*1*) **and** *f* = *this*(*2*) **and** *bound-inv* = *this*(*4*) **and**
    cdcl$_{NOT}$*-inv* = *this*(*5*) **and** *μ* = *this*(*6*)
  then obtain *m'* **where** *m*: *m* = *Suc m'* **by** (*cases m*) *auto*
  have *μ A S* − *m'* = *0*
    using *f bound-inv cdcl*$_{NOT}$*-inv μ m rtranclp-cdcl*$_{NOT}$*-raw-restart-measure-bound* **by** *fastforce*
  then have *False* **using** *cdcl*$_{NOT}$*-comp-n-le*[*of m' S T A*] *restart-step* **unfolding** *m* **by** *simp*
  then show *?case* **by** *fast*
qed

lemma *rtranclp-cdcl*$_{NOT}$*-with-inv-inv-rtranclp-cdcl*$_{NOT}$:
  assumes
    inv: *cdcl*$_{NOT}$*-inv S* **and**
    binv: *bound-inv A S*
  shows (*λS T. cdcl*$_{NOT}$ *S T* ∧ *cdcl*$_{NOT}$*-inv S* ∧ *bound-inv A S*)** *S T* ⟷ *cdcl*$_{NOT}$** *S T*
    (**is** *?A*** *S T* ⟷ *?B*** *S T*)
  apply (*rule iffI*)
    using *rtranclp-mono*[*of ?A ?B*] **apply** *blast*
  apply (*induction rule*: *rtranclp-induct*)
    using *inv binv* **apply** *simp*
  by (*metis* (*mono-tags*, *lifting*) *binv inv rtranclp.simps rtranclp-cdcl*$_{NOT}$*-bound-inv*
    rtranclp-cdcl$_{NOT}$*-cdcl*$_{NOT}$*-inv*)

lemma *no-step-cdcl*$_{NOT}$*-restart-no-step-cdcl*$_{NOT}$:
  assumes
    n-s: *no-step cdcl*$_{NOT}$*-restart S* **and**
    inv: *cdcl*$_{NOT}$*-inv* (*fst S*) **and**
    binv: *bound-inv A* (*fst S*)
  shows *no-step cdcl*$_{NOT}$ (*fst S*)
proof (*rule ccontr*)
  assume ¬ *?thesis*
  then obtain *T* **where** *T*: *cdcl*$_{NOT}$ (*fst S*) *T*
    by *blast*
  then obtain *U* **where** *U*: *full* (*λS T. cdcl*$_{NOT}$ *S T* ∧ *cdcl*$_{NOT}$*-inv S* ∧ *bound-inv A S*) *T U*
    using *wf-exists-normal-form-full*[*OF wf-cdcl*$_{NOT}$, *of A T*] **by** *auto*
  moreover have *inv-T*: *cdcl*$_{NOT}$*-inv T*
    using ‹*cdcl*$_{NOT}$ (*fst S*) *T*› *cdcl*$_{NOT}$*-inv inv* **by** *blast*
  moreover have *b-inv-T*: *bound-inv A T*
    using ‹*cdcl*$_{NOT}$ (*fst S*) *T*› *binv bound-inv inv* **by** *blast*
  ultimately have *full cdcl*$_{NOT}$ *T U*
    using *rtranclp-cdcl*$_{NOT}$*-with-inv-inv-rtranclp-cdcl*$_{NOT}$ *rtranclp-cdcl*$_{NOT}$*-bound-inv*
    rtranclp-cdcl$_{NOT}$*-cdcl*$_{NOT}$*-inv* **unfolding** *full-def* **by** *blast*
  then have *full1 cdcl*$_{NOT}$ (*fst S*) *U*
    using *T full-fullI* **by** *metis*

then show *False* **by** (*metis n-s prod.collapse restart-full*)
qed

end

5.2.6 Merging backjump and learning

locale $cdcl_{NOT}$*-merge-bj-learn-ops* =
decide-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
forget-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *forget-cond* +
propagate-ops trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *propagate-conds*
for
trail :: $'st \Rightarrow ('v,\ unit)$ *ann-lits* **and**
$clauses_{NOT}$ :: $'st \Rightarrow\ 'v$ *clauses* **and**
prepend-trail :: $('v,\ unit)$ *ann-lit* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
propagate-conds :: $('v,\ unit)$ *ann-lit* $\Rightarrow\ 'st \Rightarrow$ *bool* **and**
forget-cond :: $'v$ *clause* $\Rightarrow\ 'st \Rightarrow$ *bool* +
fixes *backjump-l-cond* :: $'v$ *clause* $\Rightarrow\ 'v$ *clause* $\Rightarrow\ 'v$ *literal* $\Rightarrow\ 'st \Rightarrow\ 'st \Rightarrow$ *bool*
begin

We have a new backjump that combines the backjumping on the trail and the learning of the used clause (called C'' below)

inductive *backjump-l* **where**
backjump-l: *trail* $S = F'$ @ *Decided* K # F
$\Longrightarrow$ *no-dup* (*trail* S)
$\Longrightarrow T \sim$ *prepend-trail* (*Propagated* L ()) ($reduce\text{-}trail\text{-}to_{NOT}$ F ($add\text{-}cls_{NOT}$ C'' S))
$\Longrightarrow C \in\#$ $clauses_{NOT}$ S
$\Longrightarrow$ *trail* $S \models as$ *CNot* C
$\Longrightarrow$ *undefined-lit* F L
$\Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* ($clauses_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S))
$\Longrightarrow$ $clauses_{NOT}$ $S \models pm$ $C' + \{\#L\#\}$
$\Longrightarrow C'' = C' + \{\#L\#\}$
$\Longrightarrow F \models as$ *CNot* C'
$\Longrightarrow$ *backjump-l-cond* C C' L S T
$\Longrightarrow$ *backjump-l* S T

Avoid (meaningless) simplification in the theorem generated by *inductive-cases*:

declare $reduce\text{-}trail\text{-}to_{NOT}$*-length-ne*[*simp del*] *Set.Un-iff*[*simp del*] *Set.insert-iff*[*simp del*]
inductive-cases *backjump-lE*: *backjump-l* S T
thm *backjump-lE*
declare $reduce\text{-}trail\text{-}to_{NOT}$*-length-ne*[*simp*] *Set.Un-iff*[*simp*] *Set.insert-iff*[*simp*]

inductive $cdcl_{NOT}$*-merged-bj-learn* :: $'st \Rightarrow\ 'st \Rightarrow$ *bool* **for** S :: $'st$ **where**
$cdcl_{NOT}$*-merged-bj-learn-decide*$_{NOT}$: $decide_{NOT}$ S $S' \Longrightarrow cdcl_{NOT}$*-merged-bj-learn* S S' |
$cdcl_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$: $propagate_{NOT}$ S $S' \Longrightarrow cdcl_{NOT}$*-merged-bj-learn* S S' |
$cdcl_{NOT}$*-merged-bj-learn-backjump-l*: *backjump-l* S $S' \Longrightarrow cdcl_{NOT}$*-merged-bj-learn* S S' |
$cdcl_{NOT}$*-merged-bj-learn-forget*$_{NOT}$: $forget_{NOT}$ S $S' \Longrightarrow cdcl_{NOT}$*-merged-bj-learn* S S'

lemma $cdcl_{NOT}$*-merged-bj-learn-no-dup-inv*:
$cdcl_{NOT}$*-merged-bj-learn* S $T \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *no-dup* (*trail* T)
apply (*induction rule*: $cdcl_{NOT}$*-merged-bj-learn.induct*)
using *defined-lit-map* **apply** *fastforce*

using *defined-lit-map* **apply** *fastforce*
apply (*force simp*: *defined-lit-map elim*!: *backjump-lE*)[]
using $forget_{NOT}$*.simps* **apply** *auto*[*1*]
done
end

locale $cdcl_{NOT}$*-merge-bj-learn-proxy* =
$cdcl_{NOT}$*-merge-bj-learn-ops trail* $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
propagate-conds forget-cond
$\lambda C\ C'\ L'\ S\ T.$ *backjump-l-cond* $C\ C'\ L'\ S\ T$
$\wedge$ *distinct-mset* $(C' + \{\#L'\#\}) \wedge \neg$*tautology* $(C' + \{\#L'\#\})$
for
trail :: $'st \Rightarrow ('v,\ unit)$ *ann-lits* **and**
$clauses_{NOT}$ :: $'st \Rightarrow\ 'v$ *clauses* **and**
prepend-trail :: $('v,\ unit)$ *ann-lit* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'v$ *clause* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
propagate-conds :: $('v,\ unit)$ *ann-lit* $\Rightarrow\ 'st \Rightarrow$ *bool* **and**
forget-cond :: $'v$ *clause* $\Rightarrow\ 'st \Rightarrow$ *bool* **and**
backjump-l-cond :: $'v$ *clause* $\Rightarrow\ 'v$ *clause* $\Rightarrow\ 'v$ *literal* $\Rightarrow\ 'st \Rightarrow\ 'st \Rightarrow$ *bool* +
fixes
inv :: $'st \Rightarrow$ *bool*
assumes
bj-merge-can-jump:
$\bigwedge S\ C\ F'\ K\ F\ L.$
inv S
$\Longrightarrow$ *trail* $S = F'$ @ *Decided* K # F
$\Longrightarrow C \in\#$ $clauses_{NOT}$ S
$\Longrightarrow$ *trail* $S \models as$ *CNot* C
$\Longrightarrow$ *undefined-lit* $F\ L$
$\Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* $(clauses_{NOT}\ S) \cup$ *atm-of* ' (*lits-of-l* (F' @ *Decided* K # F))
$\Longrightarrow$ $clauses_{NOT}$ $S \models pm\ C' + \{\#L\#\}$
$\Longrightarrow F \models as$ *CNot* C'
$\Longrightarrow \neg$*no-step backjump-l* S **and**
cdcl-merged-inv: $\bigwedge S\ T.$ $cdcl_{NOT}$*-merged-bj-learn* $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *inv* T
begin

abbreviation *backjump-conds* :: $'v$ *clause* $\Rightarrow\ 'v$ *clause* $\Rightarrow\ 'v$ *literal* $\Rightarrow\ 'st \Rightarrow\ 'st \Rightarrow$ *bool*
where
backjump-conds $\equiv \lambda C\ C'\ L'\ S\ T.$ *distinct-mset* $(C' + \{\#L'\#\}) \wedge \neg$*tautology* $(C' + \{\#L'\#\})$

Without additional knowledge on *backjump-l-cond*, it is impossible to have the same invariant.

sublocale *dpll-with-backjumping-ops trail* $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds
proof (*unfold-locales*, *goal-cases*)
case *1*
{ **fix** $S\ S'$
assume *bj*: *backjump-l* $S\ S'$ **and** *no-dup* (*trail* S)
then obtain $F'\ K\ F\ L\ C'\ C\ D$ **where**
S': $S' \sim$ *prepend-trail* (*Propagated* L ()) ($reduce\text{-}trail\text{-}to_{NOT}$ F ($add\text{-}cls_{NOT}$ $D\ S$))
and
tr-S: *trail* $S = F'$ @ *Decided* K # F **and**
C: $C \in\#$ $clauses_{NOT}$ S **and**
tr-S-C: *trail* $S \models as$ *CNot* C **and**
undef-L: *undefined-lit* $F\ L$ **and**

atm-L:
atm-of L $\in$ *insert* (*atm-of K*) (*atms-of-mm* (*clauses*$_{NOT}$ *S*) $\cup$ *atm-of* ‘ (*lits-of-l F'* $\cup$ *lits-of-l F*))
and
cls-S-C': *clauses*$_{NOT}$ *S* $\models$*pm C'* + {#*L*#} **and**
F-C': *F* $\models$*as CNot C'* **and**
dist: *distinct-mset* (*C'* + {#*L*#}) **and**
not-tauto: $\neg$ *tautology* (*C'* + {#*L*#}) **and**
cond: *backjump-l-cond C C' L S S'*
D = *C'* + {#*L*#}
by (*elim backjump-lE*) *metis*
interpret *backjumping-ops trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
backjump-conds
by *unfold-locales*
have $\exists$ *T*. *backjump S T*
apply *rule*
apply (*rule backjump.intros*)
using *tr-S* **apply** *simp*
apply (*rule state-eq*$_{NOT}$*-ref*)
using *C* **apply** *simp*
using *tr-S-C* **apply** *simp*
using *undef-L* **apply** *simp*
using *atm-L tr-S* **apply** *simp*
using *cls-S-C'* **apply** *simp*
using *F-C'* **apply** *simp*
using *dist not-tauto cond* **apply** *simp*
done
}
then show *?case* **using** *1 bj-merge-can-jump* **by** *meson*
qed

end

locale *cdcl*$_{NOT}$*-merge-bj-learn-proxy2* =
cdcl$_{NOT}$*-merge-bj-learn-proxy trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
propagate-conds forget-cond backjump-l-cond inv
for
trail :: *'st* $\Rightarrow$ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* $\Rightarrow$ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
tl-trail :: *'st* $\Rightarrow$*'st* **and**
add-cls$_{NOT}$ :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
forget-cond :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
backjump-l-cond :: *'v clause* $\Rightarrow$ *'v clause* $\Rightarrow$ *'v literal* $\Rightarrow$ *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
inv :: *'st* $\Rightarrow$ *bool*
begin

sublocale *conflict-driven-clause-learning-ops trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$
remove-cls$_{NOT}$ *inv backjump-conds propagate-conds*
λ*C* -. *distinct-mset C* $\wedge$ $\neg$*tautology C*
forget-cond
by *unfold-locales*
end

locale *cdcl*$_{NOT}$*-merge-bj-learn* =

cdcl$_{NOT}$-merge-bj-learn-proxy2 trail clauses$_{NOT}$ prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$
propagate-conds forget-cond backjump-l-cond inv
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
forget-cond :: *'v clause* ⇒ *'st* ⇒ *bool* **and**
inv :: *'st* ⇒ *bool* +
assumes
dpll-merge-bj-inv: $\bigwedge S\ T.$ *dpll-bj S T* ⟹ *inv S* ⟹ *inv T* **and**
learn-inv: $\bigwedge S\ T.$ *learn S T* ⟹ *inv S* ⟹ *inv T*
begin

sublocale
conflict-driven-clause-learning trail clauses$_{NOT}$ prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$
inv backjump-conds propagate-conds
λC -. distinct-mset C ∧ ¬*tautology C*
forget-cond
apply *unfold-locales*
using *cdcl$_{NOT}$-merged-bj-learn-forget$_{NOT}$ cdcl-merged-inv learn-inv*
by (*auto simp add*: *cdcl$_{NOT}$.simps dpll-merge-bj-inv*)

lemma *backjump-l-learn-backjump*:
assumes *bt*: *backjump-l S T* **and** *inv*: *inv S* **and** *n-d*: *no-dup* (*trail S*)
shows ∃ *C' L D. learn S* (*add-cls$_{NOT}$ D S*)
∧ *D* = (*C'* + {#*L*#})
∧ *backjump* (*add-cls$_{NOT}$ D S*) *T*
∧ *atms-of* (*C'* + {#*L*#}) ⊆ *atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
proof −
obtain *C F' K F L l C' D* **where**
tr-S: *trail S* = *F'* @ *Decided K* # *F* **and**
T: *T* ∼ *prepend-trail* (*Propagated L l*) (*reduce-trail-to$_{NOT}$ F* (*add-cls$_{NOT}$ D S*)) **and**
C-cls-S: *C* ∈# *clauses$_{NOT}$ S* **and**
tr-S-CNot-C: *trail S* ⊨*as CNot C* **and**
undef: *undefined-lit F L* **and**
atm-L: *atm-of L* ∈ *atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*)) **and**
clss-C: *clauses$_{NOT}$ S* ⊨*pm D* **and**
D: *D* = *C'* + {#*L*#}
F ⊨*as CNot C'* **and**
distinct: *distinct-mset D* **and**
not-tauto: ¬ *tautology D*
using *bt inv* **by** (*elim backjump-lE*) *simp*
have *atms-C'*: *atms-of C'* ⊆ *atm-of* ‘ (*lits-of-l F*)
by (*metis D*(*2*) *atms-of-def image-subsetI true-annots-CNot-all-atms-defined*)
then have *atms-of* (*C'* + {#*L*#}) ⊆ *atms-of-mm* (*clauses$_{NOT}$ S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
using *atm-L tr-S* **by** *auto*
moreover have *learn*: *learn S* (*add-cls$_{NOT}$ D S*)
apply (*rule learn.intros*)
apply (*rule clss-C*)
using *atms-C' atm-L D* **apply** (*fastforce simp add*: *tr-S in-plus-implies-atm-of-on-atms-of-ms*)
apply *standard*

apply (*rule distinct*)
apply (*rule not-tauto*)
apply *simp*
done
moreover have *bj*: *backjump* (*add-cls*$_{NOT}$ *D S*) *T*
apply (*rule backjump.intros*)
using ‹*F* ⊨*as CNot C'*› *C-cls-S tr-S-CNot-C undef T distinct not-tauto n-d D*
by (*auto simp*: *tr-S state-eq*$_{NOT}$*-def simp del*: *state-simp*$_{NOT}$)
ultimately show *?thesis* **using** *D* **by** *blast*
qed

lemma *cdcl*$_{NOT}$*-merged-bj-learn-is-tranclp-cdcl*$_{NOT}$:
cdcl$_{NOT}$*-merged-bj-learn S T* ⟹ *inv S* ⟹ *no-dup* (*trail S*) ⟹ *cdcl*$_{NOT}$$^{++}$ *S T*
proof (*induction rule*: *cdcl*$_{NOT}$*-merged-bj-learn.induct*)
case (*cdcl*$_{NOT}$*-merged-bj-learn-decide*$_{NOT}$ *T*)
then have *cdcl*$_{NOT}$ *S T*
using *bj-decide*$_{NOT}$ *cdcl*$_{NOT}$*.simps* **by** *fastforce*
then show *?case* **by** *auto*
next
case (*cdcl*$_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$ *T*)
then have *cdcl*$_{NOT}$ *S T*
using *bj-propagate*$_{NOT}$ *cdcl*$_{NOT}$*.simps* **by** *fastforce*
then show *?case* **by** *auto*
next
case (*cdcl*$_{NOT}$*-merged-bj-learn-forget*$_{NOT}$ *T*)
then have *cdcl*$_{NOT}$ *S T*
using *c-forget*$_{NOT}$ **by** *blast*
then show *?case* **by** *auto*
next
case (*cdcl*$_{NOT}$*-merged-bj-learn-backjump-l T*) **note** *bt* = *this*(*1*) **and** *inv* = *this*(*2*) **and**
n-d = *this*(*3*)
obtain *C'* :: *'v clause* **and** *L* :: *'v literal* **and** *D* :: *'v clause* **where**
f3: *learn S* (*add-cls*$_{NOT}$ *D S*) ∧
backjump (*add-cls*$_{NOT}$ *D S*) *T* ∧
atms-of (*C'* + {#*L*#}) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ *lits-of-l* (*trail S*) **and**
D: *D* = *C'* + {#*L*#}
using *n-d backjump-l-learn-backjump*[*OF bt inv*] **by** *blast*
then have *f4*: *cdcl*$_{NOT}$ *S* (*add-cls*$_{NOT}$ *D S*)
using *n-d c-learn* **by** *blast*
have *cdcl*$_{NOT}$ (*add-cls*$_{NOT}$ *D S*) *T*
using *f3 n-d bj-backjump c-dpll-bj* **by** *blast*
then show *?case*
using *f4* **by** (*meson tranclp.r-into-trancl tranclp.trancl-into-trancl*)
qed

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-is-rtranclp-cdcl*$_{NOT}$*-and-inv*:
cdcl$_{NOT}$*-merged-bj-learn*** *S T* ⟹ *inv S* ⟹ *no-dup* (*trail S*) ⟹ *cdcl*$_{NOT}$** *S T* ∧ *inv T*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *cdcl*$_{NOT}$ = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*−)] **and**
inv = *this*(*4*) **and** *n-d* = *this*(*5*)
have *cdcl*$_{NOT}$** *T U*
using *cdcl*$_{NOT}$*-merged-bj-learn-is-tranclp-cdcl*$_{NOT}$[*OF cdcl*$_{NOT}$] *IH*
rtranclp-cdcl$_{NOT}$*-no-dup inv n-d* **by** *auto*

then have $cdcl_{NOT}^{**}$ *S U* **using** *IH* **by** *fastforce*
moreover have *inv U* **using** *n-d IH* ⟨$cdcl_{NOT}^{**}$ *T U*⟩ *rtranclp-*$cdcl_{NOT}$*-inv* **by** *blast*
ultimately show *?case* **using** *st* **by** *fast*
qed

lemma *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$:
$cdcl_{NOT}$*-merged-bj-learn*** *S T* ⟹ *inv S* ⟹*no-dup* (*trail S*) ⟹ $cdcl_{NOT}^{**}$ *S T*
using *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$*-and-inv* **by** *blast*

lemma *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-inv*:
$cdcl_{NOT}$*-merged-bj-learn*** *S T* ⟹ *inv S* ⟹ *no-dup* (*trail S*) ⟹ *inv T*
using *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$*-and-inv* **by** *blast*

definition $\mu_C{}'$ :: *'v clause set* ⇒ *'st* ⇒ *nat* **where**
$\mu_C{}'$ *A T* ≡ μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight T*)

definition $\mu_{CDCL}{}'$*-merged* :: *'v clause set* ⇒ *'st* ⇒ *nat* **where**
$\mu_{CDCL}{}'$*-merged A T* ≡
((*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) − $\mu_C{}'$ *A T*) ∗ *2* + *card* (*set-mset* ($clauses_{NOT}$ *T*))

lemma $cdcl_{NOT}$*-decreasing-measure'*:
assumes
$cdcl_{NOT}$*-merged-bj-learn S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-trail: *atm-of ‘ lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows $\mu_{CDCL}{}'$*-merged A T* < $\mu_{CDCL}{}'$*-merged A S*
using *assms*(*1*)
proof *induction*
case ($cdcl_{NOT}$*-merged-bj-learn-decide*$_{NOT}$ *T*)
have $clauses_{NOT}$ *S* = $clauses_{NOT}$ *T*
using $cdcl_{NOT}$*-merged-bj-learn-decide*$_{NOT}$*.hyps* **by** *auto*
moreover have
(*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight T*)
< (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight S*)
apply (*rule dpll-bj-trail-mes-decreasing-prop*)
using $cdcl_{NOT}$*-merged-bj-learn-decide*$_{NOT}$ *fin-A atm-clss atm-trail n-d inv*
by (*simp-all add*: *bj-decide*$_{NOT}$ $cdcl_{NOT}$*-merged-bj-learn-decide*$_{NOT}$*.hyps*)
ultimately show *?case*
unfolding $\mu_{CDCL}{}'$*-merged-def* $\mu_C{}'$*-def* **by** *simp*
next
case ($cdcl_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$ *T*)
have $clauses_{NOT}$ *S* = $clauses_{NOT}$ *T*
using $cdcl_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$*.hyps*
by (*simp add*: *bj-propagate*$_{NOT}$ *inv dpll-bj-clauses*)
moreover have
(*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight T*)
< (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight S*)
apply (*rule dpll-bj-trail-mes-decreasing-prop*)

using *inv n-d atm-clss atm-trail fin-A* **by** (*simp-all add*: *bj-propagate*$_{NOT}$ *cdcl*$_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$*.hyps*)
ultimately show *?case*
unfolding μ_{CDCL}*'-merged-def* μ_C*'-def* **by** *simp*
next
case (*cdcl*$_{NOT}$*-merged-bj-learn-forget*$_{NOT}$ *T*)
have *card* (*set-mset* (*clauses*$_{NOT}$ *T*)) < *card* (*set-mset* (*clauses*$_{NOT}$ *S*))
using ‹*forget*$_{NOT}$ *S T*› **by** (*metis card-Diff1-less clauses-remove-cls*$_{NOT}$ *finite-set-mset forget*$_{NOT}$*.cases linear set-mset-minus-replicate-mset*(*1*) *state-eq*$_{NOT}$*-def*)
moreover
have *trail S* = *trail T*
using ‹*forget*$_{NOT}$ *S T*› **by** (*auto elim*: *forget*$_{NOT}$*E*)
then have
(*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight T*)
= (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight S*)
by *auto*
ultimately show *?case*
unfolding μ_{CDCL}*'-merged-def* μ_C*'-def* **by** *simp*
next
case (*cdcl*$_{NOT}$*-merged-bj-learn-backjump-l T*) **note** *bj-l* = *this*(*1*)
obtain *C' L D* **where**
learn: *learn S* (*add-cls*$_{NOT}$ *D S*) **and**
bj: *backjump* (*add-cls*$_{NOT}$ *D S*) *T* **and**
atms-C: *atms-of* (*C'* + {#*L*#}) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*)) **and**
D: *D* = *C'* + {#*L*#}
using *bj-l inv backjump-l-learn-backjump* [*of S*] *n-d atm-clss atm-trail* **by** *blast*
have *card-T-S*: *card* (*set-mset* (*clauses*$_{NOT}$ *T*)) ≤ *1*+ *card* (*set-mset* (*clauses*$_{NOT}$ *S*))
using *bj-l inv* **by** (*force elim*!: *backjump-lE simp*: *card-insert-if*)
have
((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight T*))
< ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*))
(*trail-weight* (*add-cls*$_{NOT}$ *D S*)))
apply (*rule dpll-bj-trail-mes-decreasing-prop*)
using *bj bj-backjump* **apply** *blast*
using *cdcl*$_{NOT}$*.c-learn cdcl*$_{NOT}$*-inv inv learn* **apply** *blast*
using *atms-C atm-clss atm-trail D* **apply** (*simp add*: *n-d*) **apply** *fast*
using *atm-trail n-d* **apply** *simp*
apply (*simp add*: *n-d*)
using *fin-A* **apply** *simp*
done
then have ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight T*))
< ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))
− μ_C (*1* + *card* (*atms-of-ms A*)) (*2* + *card* (*atms-of-ms A*)) (*trail-weight S*))
using *n-d* **by** *auto*
then show *?case*
using *card-T-S* **unfolding** μ_{CDCL}*'-merged-def* μ_C*'-def* **by** *linarith*
qed

lemma *wf-cdcl*$_{NOT}$*-merged-bj-learn*:
assumes
fin-A: *finite A*

shows *wf* $\{(T, S)$.
$(inv\ S \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
$\wedge\ no\text{-}dup\ (trail\ S))$
$\wedge\ cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ T\}$
apply (*rule wfP-if-measure*[*of - -* $\mu_{CDCL}{}'$*-merged A*])
using $cdcl_{NOT}$*-decreasing-measure′ fin-A* **by** *simp*

lemma *tranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-tranclp*:
assumes
$cdcl_{NOT}$*-merged-bj-learn*$^{++}$ *S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* ($clauses_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
atm-trail: *atm-of ‘ lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A[*simp*]: *finite A*
shows $(T, S) \in \{(T, S)$.
$(inv\ S \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
$\wedge\ no\text{-}dup\ (trail\ S))$
$\wedge\ cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ T\}^{+}$ (**is** - $\in$ *?P*$^{+}$)
using *assms*(*1*)
proof (*induction rule*: *tranclp-induct*)
case *base*
then show *?case* **using** *n-d atm-clss atm-trail inv* **by** *auto*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** $cdcl_{NOT}$ = *this*(*2*) **and** *IH* = *this*(*3*)
have $cdcl_{NOT}{}^{**}$ *S T*
apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$)
using *st* $cdcl_{NOT}$ *inv n-d atm-clss atm-trail inv* **by** *auto*
have *inv T*
apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-inv*)
using *inv st* $cdcl_{NOT}$ *n-d atm-clss atm-trail inv* **by** *auto*
moreover have *atms-of-mm* ($clauses_{NOT}$ *T*) $\subseteq$ *atms-of-ms A*
using *rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*[*OF* ‹$cdcl_{NOT}{}^{**}$ *S T*› *inv n-d atm-clss atm-trail*]
by *fast*
moreover have *atm-of ‘* (*lits-of-l* (*trail T*))$\subseteq$ *atms-of-ms A*
using *rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*[*OF* ‹$cdcl_{NOT}{}^{**}$ *S T*› *inv n-d atm-clss atm-trail*]
by *fast*
moreover have *no-dup* (*trail T*)
using *rtranclp-*$cdcl_{NOT}$*-no-dup*[*OF* ‹$cdcl_{NOT}{}^{**}$ *S T*› *inv n-d*] **by** *fast*
ultimately have $(U, T) \in$ *?P*
using $cdcl_{NOT}$ **by** *auto*
then show *?case* **using** *IH* **by** (*simp add*: *trancl-into-trancl2*)
qed

lemma *wf-tranclp-*$cdcl_{NOT}$*-merged-bj-learn*:
assumes *finite A*
shows *wf* $\{(T, S)$.
$(inv\ S \wedge atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$
$\wedge\ no\text{-}dup\ (trail\ S))$
$\wedge\ cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn^{++}\ S\ T\}$
apply (*rule wf-subset*)
apply (*rule wf-trancl*[*OF wf-*$cdcl_{NOT}$*-merged-bj-learn*])
using *assms* **apply** *simp*
using *tranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-tranclp*[*OF - - - - - ‹finite A›*] **by** *auto*

lemma *backjump-no-step-backjump-l*:

backjump S T $\Longrightarrow$ *inv S* $\Longrightarrow$ $\neg$*no-step backjump-l S*
apply (*elim backjumpE*)
apply (*rule bj-merge-can-jump*)
apply *auto*[*7*]
by *blast*

lemma *cdcl*$_{NOT}$*-merged-bj-learn-final-state*:
fixes *A* :: *'v clause set* **and** *S T* :: *'st*
assumes
n-s: *no-step cdcl*$_{NOT}$*-merged-bj-learn S* **and**
atms-S: *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
atms-trail: *atm-of ' lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
$\vee$ (*trail S* $\models$*asm clauses*$_{NOT}$ *S* $\wedge$ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))
proof −
let *?N* = *set-mset* (*clauses*$_{NOT}$ *S*)
let *?M* = *trail S*
consider
(*sat*) *satisfiable ?N* **and** *?M* $\models$*as ?N*
| (*sat'*) *satisfiable ?N* **and** $\neg$ *?M* $\models$*as ?N*
| (*unsat*) *unsatisfiable ?N*
by *auto*
then show *?thesis*
proof *cases*
case *sat'* **note** *sat* = *this*(*1*) **and** *M* = *this*(*2*)
obtain *C* **where** *C* $\in$ *?N* **and** $\neg$*?M* $\models$*a C* **using** *M* **unfolding** *true-annots-def* **by** *auto*
obtain *I* :: *'v literal set* **where**
I $\models$*s ?N* **and**
cons: *consistent-interp I* **and**
tot: *total-over-m I ?N* **and**
atm-I-N: *atm-of 'I* $\subseteq$ *atms-of-ms ?N*
using *sat* **unfolding** *satisfiable-def-min* **by** *auto*
let *?I* = *I* $\cup$ {*P*| *P*. *P* $\in$ *lits-of-l ?M* $\wedge$ *atm-of P* $\notin$ *atm-of ' I*}
let *?O* = {*unmark L* |*L*. *is-decided L* $\wedge$ *L* $\in$ *set ?M* $\wedge$ *atm-of* (*lit-of L*) $\notin$ *atms-of-ms ?N*}
have *cons-I'*: *consistent-interp ?I*
using *cons* **using** ‹*no-dup ?M*› **unfolding** *consistent-interp-def*
by (*auto simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def*
dest!: *no-dup-cannot-not-lit-and-uminus*)
have *tot-I'*: *total-over-m ?I* (*?N* $\cup$ *unmark-l ?M*)
using *tot atms-of-s-def* **unfolding** *total-over-m-def total-over-set-def*
by (*fastforce simp*: *image-iff*)
have {*P* |*P*. *P* $\in$ *lits-of-l ?M* $\wedge$ *atm-of P* $\notin$ *atm-of ' I*} $\models$*s ?O*
using ‹*I*$\models$*s ?N*› *atm-I-N* **by** (*auto simp add*: *atm-of-eq-atm-of true-clss-def lits-of-def*)
then have *I'-N*: *?I* $\models$*s ?N* $\cup$ *?O*
using ‹*I*$\models$*s ?N*› *true-clss-union-increase* **by** *force*
have *tot'*: *total-over-m ?I* (*?N*$\cup$*?O*)
using *atm-I-N tot* **unfolding** *total-over-m-def total-over-set-def*
by (*force simp*: *lits-of-def elim*!: *is-decided-ex-Decided*)

have *atms-N-M*: *atms-of-ms ?N* $\subseteq$ *atm-of ' lits-of-l ?M*
proof (*rule ccontr*)
assume $\neg$ *?thesis*

then obtain $l :: 'v$ **where**
l-N: $l \in$ *atms-of-ms ?N* **and**
l-M: $l \notin$ *atm-of ‘ lits-of-l ?M*
by *auto*
have *undefined-lit ?M (Pos l)*
using *l-M* **by** (*metis Decided-Propagated-in-iff-in-lits-of-l*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
have $decide_{NOT}$ *S (prepend-trail (Decided (Pos l)) S)*
by (*metis ‹undefined-lit ?M (Pos l)›* $decide_{NOT}$*.intros l-N literal.sel(1)*
state-eq$_{NOT}$*-ref*)
then show *False*
using $cdcl_{NOT}$*-merged-bj-learn-decide*$_{NOT}$ *n-s* **by** *blast*
qed

have *?M* $\models$*as CNot C*
apply (*rule all-variables-defined-not-imply-cnot*)
using *atms-N-M ‹C* $\in$ *?N› ‹*$\neg$ *?M* $\models$*a C› atms-of-atms-of-ms-mono[OF ‹C* $\in$ *?N›]*
by (*auto dest: atms-of-atms-of-ms-mono*)
have $\exists\, l \in$ *set ?M. is-decided l*
proof (*rule ccontr*)
let *?O* = {*unmark L* |*L. is-decided L* $\wedge$ *L* $\in$ *set ?M* $\wedge$ *atm-of* (*lit-of L*) $\notin$ *atms-of-ms ?N*}
have ϑ[*iff*]: $\bigwedge$*I. total-over-m I* (*?N* $\cup$ *?O* $\cup$ *unmark-l ?M*)
$\longleftrightarrow$ *total-over-m I* (*?N* $\cup$*unmark-l ?M*)
unfolding *total-over-set-def total-over-m-def atms-of-ms-def* **by** *blast*
assume $\neg$ *?thesis*
then have [*simp*]:{*unmark L* |*L. is-decided L* $\wedge$ *L* $\in$ *set ?M*}
= {*unmark L* |*L. is-decided L* $\wedge$ *L* $\in$ *set ?M* $\wedge$ *atm-of* (*lit-of L*) $\notin$ *atms-of-ms ?N*}
by *auto*
then have *?N* $\cup$ *?O* $\models$*ps unmark-l ?M*
using *all-decomposition-implies-propagated-lits-are-implied[OF decomp]* **by** *auto*

then have *?I* $\models$*s unmark-l ?M*
using *cons-I′ I′-N tot-I′ ‹?I* $\models$*s ?N* $\cup$ *?O›* **unfolding** ϑ *true-clss-clss-def* **by** *blast*
then have *lits-of-l ?M* $\subseteq$ *?I*
unfolding *true-clss-def lits-of-def* **by** *auto*
then have *?M* $\models$*as ?N*
using *I′-N ‹C* $\in$ *?N› ‹*$\neg$ *?M* $\models$*a C› cons-I′ atms-N-M*
by (*meson ‹trail S* $\models$*as CNot C› consistent-CNot-not rev-subsetD sup-ge1 true-annot-def*
true-annots-def true-cls-mono-set-mset-l true-clss-def)
then show *False* **using** *M* **by** *fast*
qed
from *List.split-list-first-propE[OF this]* **obtain** *K* :: *'v literal* **and** *d* :: *unit* **and**
F F′ :: (*'v, unit*) *ann-lits* **where**
M-K: *?M* = *F′* @ *Decided K* # *F* **and**
nm: $\forall f\in$*set F′.* $\neg$*is-decided f*
unfolding *is-decided-def* **by** (*metis (full-types) old.unit.exhaust*)
let *?K* = *Decided K*::(*'v, unit*) *ann-lit*
have *?K* $\in$ *set ?M*
unfolding *M-K* **by** *auto*
let *?C* = *image-mset lit-of* {#*L*$\in$#*mset ?M. is-decided L* $\wedge$ *L*$\neq$*?K*#} :: *'v clause*
let *?C′* = *set-mset* (*image-mset* (λ*L*::*'v literal.* {#*L*#}) (*?C* + *unmark ?K*))
have *?N* $\cup$ {*unmark L* |*L. is-decided L* $\wedge$ *L* $\in$ *set ?M*} $\models$*ps unmark-l ?M*
using *all-decomposition-implies-propagated-lits-are-implied[OF decomp]* **.**
moreover have *C′*: *?C′* = {*unmark L* |*L. is-decided L* $\wedge$ *L* $\in$ *set ?M*}
unfolding *M-K* **apply** *standard*
apply *force*

by *auto*
ultimately have *N-C-M*: *?N* ∪ *?C′* ⊨*ps unmark-l ?M*
by *auto*
have *N-M-False*: *?N* ∪ (*λL. unmark L*) ' (*set ?M*) ⊨*ps* {{#}}
using *M* ‹*?M* ⊨*as CNot C*› ‹*C*∈*?N*› **unfolding** *true-clss-clss-def true-annots-def Ball-def true-annot-def* **by** (*metis consistent-CNot-not sup.orderE sup-commute true-clss-def true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase*)

have *undefined-lit F K* **using** ‹*no-dup ?M*› **unfolding** *M-K* **by** (*simp add*: *defined-lit-map*)
moreover
have *?N* ∪ *?C′* ⊨*ps* {{#}}
proof −
have *A*: *?N* ∪ *?C′* ∪ *unmark-l ?M* = *?N* ∪ *unmark-l ?M*
unfolding *M-K* **by** *auto*
show *?thesis*
using *true-clss-clss-left-right*[*OF N-C-M*, *of* {{#}}] *N-M-False* **unfolding** *A* **by** *auto*
qed
have *?N* ⊨*p image-mset uminus ?C* + {#−*K*#}
unfolding *true-clss-cls-def true-clss-clss-def total-over-m-def*
proof (*intro allI impI*)
fix *I*
assume
tot: *total-over-set I* (*atms-of-ms* (*?N* ∪ {*image-mset uminus ?C*+ {#− *K*#}})) **and**
cons: *consistent-interp I* **and**
I ⊨*s ?N*
have (*K* ∈ *I* ∧ −*K* ∉ *I*) ∨ (−*K* ∈ *I* ∧ *K* ∉ *I*)
using *cons tot* **unfolding** *consistent-interp-def* **by** (*cases K*) *auto*
have {*a* ∈ *set* (*trail S*). *is-decided a* ∧ *a* ≠ *Decided K*} =
set (*trail S*) ∩ {*L. is-decided L* ∧ *L* ≠ *Decided K*}
by *auto*
then have *tot′*: *total-over-set I*
(*atm-of* ' *lit-of* ' (*set ?M* ∩ {*L. is-decided L* ∧ *L* ≠ *Decided K*}))
using *tot* **by** (*auto simp add*: *atms-of-uminus-lit-atm-of-lit-of*)
{ **fix** *x* :: (*′v*, *unit*) *ann-lit*
assume
a3: *lit-of x* ∉ *I* **and**
a1: *x* ∈ *set ?M* **and**
a4: *is-decided x* **and**
a5: *x* ≠ *Decided K*
then have *Pos* (*atm-of* (*lit-of x*)) ∈ *I* ∨ *Neg* (*atm-of* (*lit-of x*)) ∈ *I*
using *a5 a4 tot′ a1* **unfolding** *total-over-set-def atms-of-s-def* **by** *blast*
moreover have *f6*: *Neg* (*atm-of* (*lit-of x*)) = − *Pos* (*atm-of* (*lit-of x*))
by *simp*
ultimately have − *lit-of x* ∈ *I*
using *f6 a3* **by** (*metis* (*no-types*) *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel*(*1*))
} **note** *H* = *this*

have ¬*I* ⊨*s ?C′*
using ‹*?N* ∪ *?C′* ⊨*ps* {{#}}› *tot cons* ‹*I* ⊨*s ?N*›
unfolding *true-clss-clss-def total-over-m-def*
by (*simp add*: *atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of*)
then show *I* ⊨ *image-mset uminus ?C* + {#− *K*#}
unfolding *true-clss-def true-cls-def Bex-def*
using ‹(*K* ∈ *I* ∧ −*K* ∉ *I*) ∨ (−*K* ∈ *I* ∧ *K* ∉ *I*)›
by (*auto dest*!: *H*)

qed
moreover have $F \models as$ *CNot* (*image-mset uminus ?C*)
using *nm* **unfolding** *true-annots-def CNot-def M-K* **by** (*auto simp add*: *lits-of-def*)
ultimately have *False*
using *bj-merge-can-jump*[*of S F' K F C −K*
image-mset uminus (*image-mset lit-of* {# *L* :# *mset ?M. is-decided L* ∧ *L* ≠ *Decided K*#})]
⟨*C*∈*?N*⟩ *n-s* ⟨*?M* ⊨*as CNot C*⟩ *bj-backjump inv* **unfolding** *M-K*
by (*auto simp*: $cdcl_{NOT}$*-merged-bj-learn.simps*)
then show *?thesis* **by** *fast*
qed *auto*
qed

lemma *full-*$cdcl_{NOT}$*-merged-bj-learn-final-state*:
fixes *A* :: *'v clause set* **and** *S T* :: *'st*
assumes
full: *full* $cdcl_{NOT}$*-merged-bj-learn S T* **and**
atms-S: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite A **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ *T*))
∨ (*trail T* ⊨*asm* $clauses_{NOT}$ *T* ∧ *satisfiable* (*set-mset* ($clauses_{NOT}$ *T*)))
proof −
have *st*: $cdcl_{NOT}$*-merged-bj-learn*** *S T* **and** *n-s*: *no-step* $cdcl_{NOT}$*-merged-bj-learn T*
using *full* **unfolding** *full-def* **by** *blast+*
then have *st*: $cdcl_{NOT}{}^{**}$ *S T*
using *inv rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$*-and-inv n-d* **by** *auto*
have *atms-of-mm* ($clauses_{NOT}$ *T*) ⊆ *atms-of-ms A* **and** *atm-of* ' *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
using *rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*[*OF st inv n-d atms-S atms-trail*] **by** *blast+*
moreover have *no-dup* (*trail T*)
using *rtranclp-*$cdcl_{NOT}$*-no-dup inv n-d st* **by** *blast*
moreover have *inv T*
using *rtranclp-*$cdcl_{NOT}$*-inv inv st* **by** *blast*
moreover have *all-decomposition-implies-m* ($clauses_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
using *rtranclp-*$cdcl_{NOT}$*-all-decomposition-implies inv st decomp n-d* **by** *blast*
ultimately show *?thesis*
using $cdcl_{NOT}$*-merged-bj-learn-final-state*[*of T A*] ⟨*finite A*⟩ *n-s* **by** *fast*
qed

end

5.2.7 Instantiations

In this section, we instantiate the previous locales to ensure that the assumption are not contradictory.

locale $cdcl_{NOT}$*-with-backtrack-and-restarts* =
conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt
trail $clauses_{NOT}$ *prepend-trail tl-trail* $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds learn-restrictions forget-restrictions
for
trail :: *'st* ⇒ (*'v*, *unit*) *ann-lits* **and**
$clauses_{NOT}$ :: *'st* ⇒ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**

tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
learn-restrictions forget-restrictions :: *'v clause* ⇒ *'st* ⇒ *bool*
+
fixes *f* :: *nat* ⇒ *nat*
assumes
unbounded: *unbounded f* **and** *f-ge-1*: ⋀*n*. *n* ≥ *1* ⟹ *f n* ≥ *1* **and**
inv-restart:⋀*S T*. *inv S* ⟹ *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::*'a list*) *S* ⟹ *inv T*
begin

lemma *bound-inv-inv*:
assumes
inv S **and**
n-d: *no-dup* (*trail S*) **and**
atms-clss-S-A: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atms-trail-S-A:*atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
finite A **and**
cdcl$_{NOT}$: *cdcl*$_{NOT}$ *S T*
shows
atms-of-mm (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A* **and**
finite A
proof −
have *cdcl*$_{NOT}$ *S T*
using ⟨*inv S*⟩ *cdcl*$_{NOT}$ **by** *linarith*
then have *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ *lits-of-l* (*trail S*)
using ⟨*inv S*⟩
by (*meson conflict-driven-clause-learning-ops.cdcl*$_{NOT}$*-atms-of-ms-clauses-decreasing*
conflict-driven-clause-learning-ops-axioms n-d)
then show *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
using *atms-clss-S-A atms-trail-S-A* **by** *blast*
next
show *atm-of* ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
by (*meson* ⟨*inv S*⟩ *atms-clss-S-A atms-trail-S-A cdcl*$_{NOT}$ *cdcl*$_{NOT}$*-atms-in-trail-in-set n-d*)
next
show *finite A*
using ⟨*finite A*⟩ **by** *simp*
qed

sublocale *cdcl*$_{NOT}$*-increasing-restarts-ops* λ*S T*. *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::*'a list*) *S cdcl*$_{NOT}$ *f*
λ*A S*. *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧
finite A
μ_{CDCL}*'* λ*S*. *inv S* ∧ *no-dup* (*trail S*)
μ_{CDCL}*'-bound*
apply *unfold-locales*
apply (*simp add*: *unbounded*)
using *f-ge-1* **apply** *force*
using *bound-inv-inv* **apply** *meson*
apply (*rule cdcl*$_{NOT}$*-decreasing-measure'*; *simp*)
apply (*rule rtranclp-cdcl*$_{NOT}$*-*μ_{CDCL}*'-bound*; *simp*)
apply (*rule rtranclp-*μ_{CDCL}*'-bound-decreasing*; *simp*)
apply *auto*[]

```
    apply auto[]
   using cdcl_NOT-inv cdcl_NOT-no-dup apply blast
  using inv-restart apply auto[]
  done

lemma cdcl_NOT-with-restart-μ_CDCL'-le-μ_CDCL'-bound:
  assumes
    cdcl_NOT: cdcl_NOT-restart (T, a) (V, b) and
    cdcl_NOT-inv:
      inv T
      no-dup (trail T) and
    bound-inv:
      atms-of-mm (clauses_NOT T) ⊆ atms-of-ms A
      atm-of ` lits-of-l (trail T) ⊆ atms-of-ms A
      finite A
  shows μ_CDCL' A V ≤ μ_CDCL'-bound A T
  using cdcl_NOT-inv bound-inv
proof (induction rule: cdcl_NOT-with-restart-induct[OF cdcl_NOT])
  case (1 m S T n U) note U = this(3)
  show ?case
    apply (rule rtranclp-cdcl_NOT-μ_CDCL'-bound-reduce-trail-to_NOT[of S T])
         using ‹(cdcl_NOT ^^ m) S T› apply (fastforce dest!: relpowp-imp-rtranclp)
        using 1 by auto
next
  case (2 S T n) note full = this(2)
  show ?case
    apply (rule rtranclp-cdcl_NOT-μ_CDCL'-bound)
    using full 2 unfolding full1-def by force+
qed

lemma cdcl_NOT-with-restart-μ_CDCL'-bound-le-μ_CDCL'-bound:
  assumes
    cdcl_NOT: cdcl_NOT-restart (T, a) (V, b) and
    cdcl_NOT-inv:
      inv T
      no-dup (trail T) and
    bound-inv:
      atms-of-mm (clauses_NOT T) ⊆ atms-of-ms A
      atm-of ` lits-of-l (trail T) ⊆ atms-of-ms A
      finite A
  shows μ_CDCL'-bound A V ≤ μ_CDCL'-bound A T
  using cdcl_NOT-inv bound-inv
proof (induction rule: cdcl_NOT-with-restart-induct[OF cdcl_NOT])
  case (1 m S T n U) note U = this(3)
  have μ_CDCL'-bound A T ≤ μ_CDCL'-bound A S
     apply (rule rtranclp-μ_CDCL'-bound-decreasing)
         using ‹(cdcl_NOT ^^ m) S T› apply (fastforce dest: relpowp-imp-rtranclp)
        using 1 by auto
  then show ?case using U unfolding μ_CDCL'-bound-def by auto
next
  case (2 S T n) note full = this(2)
  show ?case
    apply (rule rtranclp-μ_CDCL'-bound-decreasing)
    using full 2 unfolding full1-def by force+
qed
```

sublocale *cdcl*$_{NOT}$*-increasing-restarts* - - - - - -
f
$\lambda S\ T.\ T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S*
$\lambda A\ S.$ *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* $\wedge$ *finite A*
$\mu_{CDCL}{}'$ *cdcl*$_{NOT}$
$\lambda S.$ *inv S* $\wedge$ *no-dup* (*trail S*)
$\mu_{CDCL}{}'$*-bound*
apply *unfold-locales*
using *cdcl*$_{NOT}$*-with-restart-*$\mu_{CDCL}{}'$*-le-*$\mu_{CDCL}{}'$*-bound* **apply** *simp*
using *cdcl*$_{NOT}$*-with-restart-*$\mu_{CDCL}{}'$*-bound-le-*$\mu_{CDCL}{}'$*-bound* **apply** *simp*
done

lemma *cdcl*$_{NOT}$*-restart-all-decomposition-implies*:
assumes *cdcl*$_{NOT}$*-restart S T* **and**
inv (*fst S*) **and**
no-dup (*trail* (*fst S*))
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst S*)) (*get-all-ann-decomposition* (*trail* (*fst S*)))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst T*)) (*get-all-ann-decomposition* (*trail* (*fst T*)))
using *assms* **apply** (*induction*)
using *rtranclp-cdcl*$_{NOT}$*-all-decomposition-implies* **by** (*auto dest*!: *tranclp-into-rtranclp*
simp: *full1-def*)

lemma *rtranclp-cdcl*$_{NOT}$*-restart-all-decomposition-implies*:
assumes *cdcl*$_{NOT}$*-restart*** *S T* **and**
inv: *inv* (*fst S*) **and**
n-d: *no-dup* (*trail* (*fst S*)) **and**
decomp:
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst S*)) (*get-all-ann-decomposition* (*trail* (*fst S*)))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst T*)) (*get-all-ann-decomposition* (*trail* (*fst T*)))
using *assms*(*1*)
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **using** *decomp* **by** *simp*
next
case (*step T u*) **note** *st* = *this*(*1*) **and** *r* = *this*(*2*) **and** *IH* = *this*(*3*)
have *inv* (*fst T*)
using *rtranclp-cdcl*$_{NOT}$*-with-restart-cdcl*$_{NOT}$*-inv*[*OF st*] *inv n-d* **by** *blast*
moreover have *no-dup* (*trail* (*fst T*))
using *rtranclp-cdcl*$_{NOT}$*-with-restart-cdcl*$_{NOT}$*-inv*[*OF st*] *inv n-d* **by** *blast*
ultimately show *?case*
using *cdcl*$_{NOT}$*-restart-all-decomposition-implies r IH n-d* **by** *fast*
qed

lemma *cdcl*$_{NOT}$*-restart-sat-ext-iff*:
assumes
st: *cdcl*$_{NOT}$*-restart S T* **and**
n-d: *no-dup* (*trail* (*fst S*)) **and**
inv: *inv* (*fst S*)
shows *I* $\models$*sextm clauses*$_{NOT}$ (*fst S*) $\longleftrightarrow$ *I* $\models$*sextm clauses*$_{NOT}$ (*fst T*)
using *assms*
proof (*induction*)
case (*restart-step m S T n U*)
then show *?case*

  using *rtranclp-cdcl*$_{NOT}$*-bj-sat-ext-iff n-d* **by** (*fastforce dest*!: *relpowp-imp-rtranclp*)
next
 case *restart-full*
 then show *?case* **using** *rtranclp-cdcl*$_{NOT}$*-bj-sat-ext-iff* **unfolding** *full1-def*
 by (*fastforce dest*!: *tranclp-into-rtranclp*)
qed

lemma *rtranclp-cdcl*$_{NOT}$*-restart-sat-ext-iff*:
 fixes *S T* :: *'st* × *nat*
 assumes
  st: *cdcl*$_{NOT}$*-restart*** *S T* **and**
  n-d: *no-dup* (*trail* (*fst S*)) **and**
  inv: *inv* (*fst S*)
 shows *I* ⊨*sextm clauses*$_{NOT}$ (*fst S*) ⟷ *I* ⊨*sextm clauses*$_{NOT}$ (*fst T*)
 using *st*
proof (*induction*)
 case *base*
 then show *?case* **by** *simp*
next
 case (*step T U*) **note** *st* = *this*(*1*) **and** *r* = *this*(*2*) **and** *IH* = *this*(*3*)
 have *inv* (*fst T*)
  using *rtranclp-cdcl*$_{NOT}$*-with-restart-cdcl*$_{NOT}$*-inv*[*OF st*] *inv n-d* **by** *blast+*
 moreover have *no-dup* (*trail* (*fst T*))
  using *rtranclp-cdcl*$_{NOT}$*-with-restart-cdcl*$_{NOT}$*-inv rtranclp-cdcl*$_{NOT}$*-no-dup st inv n-d* **by** *blast*
 ultimately show *?case*
  using *cdcl*$_{NOT}$*-restart-sat-ext-iff*[*OF r*] *IH* **by** *blast*
qed

theorem *full-cdcl*$_{NOT}$*-restart-backjump-final-state*:
 fixes *A* :: *'v clause set* **and** *S T* :: *'st*
 assumes
  full: *full cdcl*$_{NOT}$*-restart* (*S*, *n*) (*T*, *m*) **and**
  atms-S: *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
  atms-trail: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
  n-d: *no-dup* (*trail S*) **and**
  fin-A[*simp*]: *finite A* **and**
  inv: *inv S* **and**
  decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
 shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
  ∨ (*lits-of-l* (*trail T*) ⊨*sextm clauses*$_{NOT}$ *S* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))
proof −
 have *st*: *cdcl*$_{NOT}$*-restart*** (*S*, *n*) (*T*, *m*) **and**
  n-s: *no-step cdcl*$_{NOT}$*-restart* (*T*, *m*)
  using *full* **unfolding** *full-def* **by** *fast+*
 have *binv-T*: *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
  atm-of ‘ *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
  using *rtranclp-cdcl*$_{NOT}$*-with-restart-bound-inv*[*OF st*, *of A*] *inv n-d atms-S atms-trail*
  by *auto*
 moreover have *inv-T*: *no-dup* (*trail T*) *inv T*
  using *rtranclp-cdcl*$_{NOT}$*-with-restart-cdcl*$_{NOT}$*-inv*[*OF st*] *inv n-d* **by** *auto*
 moreover have *all-decomposition-implies-m* (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
  using *rtranclp-cdcl*$_{NOT}$*-restart-all-decomposition-implies*[*OF st*] *inv n-d*
  decomp **by** *auto*
 ultimately have *T*: *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*))
  ∨ (*trail T* ⊨*asm clauses*$_{NOT}$ *T* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*)))
  using *no-step-cdcl*$_{NOT}$*-restart-no-step-cdcl*$_{NOT}$[*of* (*T*, *m*) *A*] *n-s*

cdcl$_{NOT}$*-final-state*[*of T A*] **unfolding** *cdcl*$_{NOT}$*-NOT-all-inv-def* **by** *auto*
have *eq-sat-S-T*:$\bigwedge$*I. I* $\models$*sextm clauses*$_{NOT}$ *S* $\longleftrightarrow$ *I* $\models$*sextm clauses*$_{NOT}$ *T*
using *rtranclp-cdcl*$_{NOT}$*-restart-sat-ext-iff*[*OF st*] *inv n-d atms-S*
atms-trail **by** *auto*
have *cons-T*: *consistent-interp* (*lits-of-l* (*trail T*))
using *inv-T*(*1*) *distinct-consistent-interp* **by** *blast*
consider
(*unsat*) *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*))
| (*sat*) *trail T* $\models$*asm clauses*$_{NOT}$ *T* **and** *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *T*))
using *T* **by** *blast*
then show *?thesis*
proof *cases*
case *unsat*
then have *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
using *eq-sat-S-T consistent-true-clss-ext-satisfiable true-clss-imp-true-cls-ext*
unfolding *satisfiable-def* **by** *blast*
then show *?thesis* **by** *fast*
next
case *sat*
then have *lits-of-l* (*trail T*) $\models$*sextm clauses*$_{NOT}$ *S*
using *rtranclp-cdcl*$_{NOT}$*-restart-sat-ext-iff*[*OF st*] *inv n-d atms-S*
atms-trail **by** (*auto simp*: *true-clss-imp-true-cls-ext true-annots-true-cls*)
moreover then have *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
using *cons-T consistent-true-clss-ext-satisfiable* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed
qed
end — end of *cdcl*$_{NOT}$*-with-backtrack-and-restarts* locale

The restart does only reset the trail, contrary to Weidenbach's version where forget and restart are always combined. But there is a forget rule.

locale *cdcl*$_{NOT}$*-merge-bj-learn-with-backtrack-restarts* =
cdcl$_{NOT}$*-merge-bj-learn trail clauses*$_{NOT}$ *prepend-trail tl-trail add-cls*$_{NOT}$ *remove-cls*$_{NOT}$
λ*C C' L' S T. distinct-mset* (*C'* + {#*L'*#}) $\wedge$ *backjump-l-cond C C' L' S T*
propagate-conds forget-conds inv
for
trail :: *'st* $\Rightarrow$ (*'v*, *unit*) *ann-lits* **and**
clauses$_{NOT}$ :: *'st* $\Rightarrow$ *'v clauses* **and**
prepend-trail :: (*'v*, *unit*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
tl-trail :: *'st* $\Rightarrow$*'st* **and**
add-cls$_{NOT}$ :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
remove-cls$_{NOT}$ :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
propagate-conds :: (*'v*, *unit*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
inv :: *'st* $\Rightarrow$ *bool* **and**
forget-conds :: *'v clause* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
backjump-l-cond :: *'v clause* $\Rightarrow$ *'v clause* $\Rightarrow$ *'v literal* $\Rightarrow$ *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool*
+
fixes *f* :: *nat* $\Rightarrow$ *nat*
assumes
unbounded: *unbounded f* **and** *f-ge-1*: $\bigwedge$*n. n* $\geq$ *1* $\Longrightarrow$ *f n* $\geq$ *1* **and**
inv-restart:$\bigwedge$*S T. inv S* $\Longrightarrow$ *T* $\sim$ *reduce-trail-to*$_{NOT}$ [] *S* $\Longrightarrow$ *inv T*
begin

definition *not-simplified-cls* :: *'b literal multiset multiset* $\Rightarrow$ *'b literal multiset multiset*
where
not-simplified-cls A $\equiv$ {#*C* $\in$# *A. C* $\notin$ *simple-clss* (*atms-of-mm A*)#}

lemma *not-simplified-cls-tautology-distinct-mset*:
not-simplified-cls $A = \{\#C \in\# A.$ *tautology* $C \vee \neg$*distinct-mset* $C\#\}$
unfolding *not-simplified-cls-def* **by** (*rule filter-mset-cong*) (*auto simp*: *simple-clss-def*)

lemma *simple-clss-or-not-simplified-cls*:
assumes *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
$x \in\#$ *clauses*$_{NOT}$ S **and** *finite* A
shows $x \in$ *simple-clss* (*atms-of-ms* A) $\vee$ $x \in\#$ *not-simplified-cls* (*clauses*$_{NOT}$ S)
proof −
consider
(*simpl*) $\neg$*tautology* x **and** *distinct-mset* x
| (*n-simp*) *tautology* x $\vee$ $\neg$*distinct-mset* x
by *auto*
then show *?thesis*
proof *cases*
case *simpl*
then have $x \in$ *simple-clss* (*atms-of-ms* A)
by (*meson assms atms-of-atms-of-ms-mono atms-of-ms-finite simple-clss-mono distinct-mset-not-tautology-implies-in-simple-clss finite-subset subsetCE*)
then show *?thesis* **by** *blast*
next
case *n-simp*
then have $x \in\#$ *not-simplified-cls* (*clauses*$_{NOT}$ S)
using ‹$x \in\#$ *clauses*$_{NOT}$ S› **unfolding** *not-simplified-cls-tautology-distinct-mset* **by** *auto*
then show *?thesis* **by** *blast*
qed
qed

lemma *cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound*:
assumes
cdcl$_{NOT}$*-merged-bj-learn* S T **and**
inv: *inv* S **and**
atms-clss: *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atms-trail: *atm-of* ‘(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
fin-A[*simp*]: *finite* A
shows *set-mset* (*clauses*$_{NOT}$ T) $\subseteq$ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ S))
$\cup$ *simple-clss* (*atms-of-ms* A)
using *assms*
proof (*induction rule*: *cdcl*$_{NOT}$*-merged-bj-learn.induct*)
case *cdcl*$_{NOT}$*-merged-bj-learn-decide*$_{NOT}$
then show *?case* **using** *dpll-bj-clauses* **by** (*force dest*!: *simple-clss-or-not-simplified-cls*)
next
case *cdcl*$_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$
then show *?case* **using** *dpll-bj-clauses* **by** (*force dest*!: *simple-clss-or-not-simplified-cls*)
next
case *cdcl*$_{NOT}$*-merged-bj-learn-forget*$_{NOT}$
then show *?case* **using** *clauses-remove-cls*$_{NOT}$ **unfolding** *state-eq*$_{NOT}$*-def*
by (*force elim*!: *forget*$_{NOT}$*E dest*: *simple-clss-or-not-simplified-cls*)
next
case (*cdcl*$_{NOT}$*-merged-bj-learn-backjump-l* T) **note** *bj* = *this*(*1*) **and** *inv* = *this*(*2*) **and**
atms-clss = *this*(*3*) **and** *atms-trail* = *this*(*4*) **and** *n-d* = *this*(*5*)

have *cdcl*$_{NOT}$** S T

apply (*rule rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-is-rtranclp-cdcl*$_{NOT}$)
using *bj inv cdcl*$_{NOT}$*-merged-bj-learn.simps n-d* **by** *blast+*
have *atm-of* '(*lits-of-l* (*trail T*)) ⊆ *atms-of-ms A*
using *rtranclp-cdcl*$_{NOT}$*-trail-clauses-bound*[*OF* ‹*cdcl*$_{NOT}$** *S T*›] *inv atms-trail atms-clss n-d* **by** *auto*
have *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
using *rtranclp-cdcl*$_{NOT}$*-trail-clauses-bound*[*OF* ‹*cdcl*$_{NOT}$** *S T*› *inv n-d atms-clss atms-trail*]
by *fast*
moreover have *no-dup* (*trail T*)
using *rtranclp-cdcl*$_{NOT}$*-no-dup*[*OF* ‹*cdcl*$_{NOT}$** *S T*› *inv n-d*] **by** *fast*

obtain *F' K F L l C' C D* **where**
tr-S: *trail S* = *F'* @ *Decided K* # *F* **and**
T: *T* ∼ *prepend-trail* (*Propagated L l*) (*reduce-trail-to*$_{NOT}$ *F* (*add-cls*$_{NOT}$ *D S*)) **and**
C ∈# *clauses*$_{NOT}$ *S* **and**
trail S ⊨*as CNot C* **and**
undef: *undefined-lit F L* **and**
clauses$_{NOT}$ *S* ⊨*pm C'* + {#*L*#} **and**
F ⊨*as CNot C'* **and**
D: *D* = *C'* + {#*L*#} **and**
dist: *distinct-mset* (*C'* + {#*L*#}) **and**
tauto: ¬ *tautology* (*C'* + {#*L*#}) **and**
backjump-l-cond C C' L S T
using ‹*backjump-l S T*› **apply** (*elim backjump-lE*) **by** *auto*

have *atms-of C'* ⊆ *atm-of* ' (*lits-of-l F*)
using ‹*F* ⊨*as CNot C'*› **by** (*simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def image-subset-iff in-CNot-implies-uminus*(*2*))
then have *atms-of* (*C'*+{#*L*#}) ⊆ *atms-of-ms A*
using *T* ‹*atm-of* ' *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*› *tr-S undef n-d* **by** *auto*
then have *simple-clss* (*atms-of* (*C'* + {#*L*#})) ⊆ *simple-clss* (*atms-of-ms A*)
apply − **by** (*rule simple-clss-mono*) (*simp-all*)
then have *C'* + {#*L*#} ∈ *simple-clss* (*atms-of-ms A*)
using *distinct-mset-not-tautology-implies-in-simple-clss*[*OF dist tauto*]
by *auto*
then show *?case*
using *T inv atms-clss undef tr-S n-d D* **by** (*force dest*!: *simple-clss-or-not-simplified-cls*)
qed

lemma *cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl*$_{NOT}$*-merged-bj-learn S T*
shows *not-simplified-cls* (*clauses*$_{NOT}$ *T*) ⊆# *not-simplified-cls* (*clauses*$_{NOT}$ *S*)
using *assms* **apply** *induction*
prefer *4*
unfolding *not-simplified-cls-tautology-distinct-mset* **apply** (*auto elim*!: *backjump-lE forget*$_{NOT}$*E*)[*3*]
by (*elim backjump-lE*) *auto*

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl*$_{NOT}$*-merged-bj-learn*** *S T*
shows *not-simplified-cls* (*clauses*$_{NOT}$ *T*) ⊆# *not-simplified-cls* (*clauses*$_{NOT}$ *S*)
using *assms* **apply** *induction*
apply *simp*
by (*drule cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*) *auto*

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound*:
assumes

$cdcl_{NOT}$*-merged-bj-learn*** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite[*simp*]: *finite A*
shows *set-mset* (*clauses*$_{NOT}$ *T*) ⊆ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ *S*))
∪ *simple-clss* (*atms-of-ms A*)
using *assms*(*1−5*)
proof *induction*
case *base*
then show *?case* **by** (*auto dest*!: *simple-clss-or-not-simplified-cls*)
next
case (*step T U*) **note** *st* = *this*(*1*) **and** $cdcl_{NOT}$ = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4−7*)] **and**
inv = *this*(*4*) **and** *atms-clss-S* = *this*(*5*) **and** *atms-trail-S* = *this*(*6*) **and** *finite-cls-S* = *this*(*7*)
have *st'*: $cdcl_{NOT}{}^{**}$ *S T*
using *inv rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$*-and-inv st n-d* **by** *blast*
have *inv T*
using *inv rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-inv st n-d* **by** *blast*
moreover
have *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A* **and**
atm-of ' *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
using *rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*[*OF st'*] *inv atms-clss-S atms-trail-S n-d*
by *blast*+
moreover moreover have *no-dup* (*trail T*)
using *rtranclp-*$cdcl_{NOT}$*-no-dup*[*OF* ‹$cdcl_{NOT}{}^{**}$ *S T*› *inv n-d*] **by** *fast*
ultimately have *set-mset* (*clauses*$_{NOT}$ *U*)
⊆ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ *T*)) ∪ *simple-clss* (*atms-of-ms A*)
using $cdcl_{NOT}$ *finite* $cdcl_{NOT}$*-merged-bj-learn-clauses-bound*
by (*auto intro*!: $cdcl_{NOT}$*-merged-bj-learn-clauses-bound*)
moreover have *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ *T*))
⊆ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ *S*))
using *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-not-simplified-decreasing*[*OF st*] **by** *auto*
ultimately show *?case* **using** *IH inv atms-clss-S*
by (*auto dest*!: *simple-clss-or-not-simplified-cls*)
qed

abbreviation $\mu_{CDCL}{}'$*-bound* **where**
$\mu_{CDCL}{}'$*-bound A T* ≡ ((*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))) ∗ *2*
+ *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ *T*)))
+ *3* ^ *card* (*atms-of-ms A*)

lemma *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-clauses-bound-card*:
assumes
$cdcl_{NOT}$*-merged-bj-learn*** *S T* **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
atm-of '(*lits-of-l* (*trail S*)) ⊆ *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows $\mu_{CDCL}{}'$*-merged A T* ≤ $\mu_{CDCL}{}'$*-bound A S*
proof −
have *set-mset* (*clauses*$_{NOT}$ *T*) ⊆ *set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ *S*))
∪ *simple-clss* (*atms-of-ms A*)
using *rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-clauses-bound*[*OF assms*] **.**
moreover have *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ *S*))

$\cup$ *simple-clss* (*atms-of-ms A*))
$\leq$ *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ *S*))) + *3* ^ *card* (*atms-of-ms A*)
by (*meson Nat.le-trans atms-of-ms-finite simple-clss-card card-Un-le finite nat-add-left-cancel-le*)
ultimately have *card* (*set-mset* (*clauses*$_{NOT}$ *T*))
$\leq$ *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ *S*))) + *3* ^ *card* (*atms-of-ms A*)
by (*meson Nat.le-trans atms-of-ms-finite simple-clss-finite card-mono finite-UnI finite-set-mset local.finite*)
moreover have ((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) − μ_C' *A T*) ∗ *2*
$\leq$ (*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*)) ∗ *2*
by *auto*
ultimately show *?thesis* **unfolding** μ_{CDCL}'*-merged-def* **by** *auto*
qed

sublocale *cdcl*$_{NOT}$*-increasing-restarts-ops* λ*S T*. *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S*
cdcl$_{NOT}$*-merged-bj-learn f*
λ*A S*. *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* $\wedge$ *finite A*
μ_{CDCL}'*-merged*
λ*S*. *inv S* $\wedge$ *no-dup* (*trail S*)
μ_{CDCL}'*-bound*
apply *unfold-locales*
using *unbounded* **apply** *simp*
using *f-ge-1* **apply** *force*
apply (*blast dest*!: *cdcl*$_{NOT}$*-merged-bj-learn-is-tranclp-cdcl*$_{NOT}$ *tranclp-into-rtranclp rtranclp-cdcl*$_{NOT}$*-trail-clauses-bound*)
apply (*simp add*: *cdcl*$_{NOT}$*-decreasing-measure*')
using *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound-card* **apply** *blast*
apply (*drule rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*)
apply (*auto simp*: *card-mono set-mset-mono*)[]
apply *simp*
apply *auto*[]
using *cdcl*$_{NOT}$*-merged-bj-learn-no-dup-inv cdcl-merged-inv* **apply** *blast*
apply (*auto simp*: *inv-restart*)[]
done

lemma *cdcl*$_{NOT}$*-restart-*μ_{CDCL}'*-merged-le-*μ_{CDCL}'*-bound*:
assumes
cdcl$_{NOT}$*-restart T V*
inv (*fst T*) **and**
no-dup (*trail* (*fst T*)) **and**
atms-of-mm (*clauses*$_{NOT}$ (*fst T*)) $\subseteq$ *atms-of-ms A* **and**
atm-of ' *lits-of-l* (*trail* (*fst T*)) $\subseteq$ *atms-of-ms A* **and**
finite A
shows μ_{CDCL}'*-merged A* (*fst V*) $\leq$ μ_{CDCL}'*-bound A* (*fst T*)
using *assms*
proof *induction*
case (*restart-full S T n*)
show *?case*
unfolding *fst-conv*
apply (*rule rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound-card*)
using *restart-full* **unfolding** *full1-def* **by** (*force dest*!: *tranclp-into-rtranclp*)+
next
case (*restart-step m S T n U*) **note** *st* = *this*(*1*) **and** *U* = *this*(*3*) **and** *inv* = *this*(*4*) **and** *n-d* = *this*(*5*) **and** *atms-clss* = *this*(*6*) **and** *atms-trail* = *this*(*7*) **and** *finite* = *this*(*8*)
then have *st'*: *cdcl*$_{NOT}$*-merged-bj-learn*** *S T*

by (*blast dest*: *relpowp-imp-rtranclp*)
then have *st''*: $cdcl_{NOT}{}^{**}$ *S T*
using *inv n-d* **apply** − **by** (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$) *auto*
have *inv T*
apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-inv*)
using *inv st' n-d* **by** *auto*
then have *inv U*
using *U* **by** (*auto simp*: *inv-restart*)
have *atms-of-mm* ($clauses_{NOT}$ *T*) ⊆ *atms-of-ms A*
using *rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*[*OF st''*] *inv atms-clss atms-trail n-d*
by *simp*
then have *atms-of-mm* ($clauses_{NOT}$ *U*) ⊆ *atms-of-ms A*
using *U* **by** *simp*
have *not-simplified-cls* ($clauses_{NOT}$ *U*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *T*)
using ‹*U* ∼ *reduce-trail-to*$_{NOT}$ [] *T*› **by** *auto*
moreover have *not-simplified-cls* ($clauses_{NOT}$ *T*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *S*)
apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-not-simplified-decreasing*)
using ‹($cdcl_{NOT}$*-merged-bj-learn* ^^ *m*) *S T*› **by** (*auto dest*!: *relpowp-imp-rtranclp*)
ultimately have *U-S*: *not-simplified-cls* ($clauses_{NOT}$ *U*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *S*)
by *auto*

have (*set-mset* ($clauses_{NOT}$ *U*))
⊆ *set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *U*)) ∪ *simple-clss* (*atms-of-ms A*)
apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-clauses-bound*)
apply *simp*
using ‹*inv U*› **apply** *simp*
using ‹*atms-of-mm* ($clauses_{NOT}$ *U*) ⊆ *atms-of-ms A*› **apply** *simp*
using *U* **apply** *simp*
using *U* **apply** *simp*
using *finite* **apply** *simp*
done
then have *f1*: *card* (*set-mset* ($clauses_{NOT}$ *U*)) ≤ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *U*))
∪ *simple-clss* (*atms-of-ms A*))
by (*simp add*: *simple-clss-finite card-mono local.finite*)

moreover have *set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *U*)) ∪ *simple-clss* (*atms-of-ms A*)
⊆ *set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *S*)) ∪ *simple-clss* (*atms-of-ms A*)
using *U-S* **by** *auto*
then have *f2*:
card (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *U*)) ∪ *simple-clss* (*atms-of-ms A*))
≤ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *S*)) ∪ *simple-clss* (*atms-of-ms A*))
by (*simp add*: *simple-clss-finite card-mono local.finite*)

moreover have *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *S*))
∪ *simple-clss* (*atms-of-ms A*))
≤ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *S*))) + *card* (*simple-clss* (*atms-of-ms A*))
using *card-Un-le* **by** *blast*
moreover have *card* (*simple-clss* (*atms-of-ms A*)) ≤ *3* ^ *card* (*atms-of-ms A*)
using *atms-of-ms-finite simple-clss-card local.finite* **by** *blast*
ultimately have *card* (*set-mset* ($clauses_{NOT}$ *U*))
≤ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ *S*))) + *3* ^ *card* (*atms-of-ms A*)
by *linarith*
then show *?case* **unfolding** $\mu_{CDCL}{}'$*-merged-def* **by** *auto*
qed

lemma $cdcl_{NOT}$*-restart-*$\mu_{CDCL}{}'$*-bound-le-*$\mu_{CDCL}{}'$*-bound*:

assumes
  $cdcl_{NOT}$*-restart T V* **and**
  no-dup (*trail* (*fst T*)) **and**
  inv (*fst T*) **and**
  fin: *finite A*
shows μ_{CDCL}*'-bound A* (*fst V*) $\leq$ μ_{CDCL}*'-bound A* (*fst T*)
using *assms*(*1*−*3*)
proof *induction*
  case (*restart-full S T n*)
  have *not-simplified-cls* ($clauses_{NOT}$ *T*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *S*)
    apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-not-simplified-decreasing*)
    using ‹*full1* $cdcl_{NOT}$*-merged-bj-learn S T*› **unfolding** *full1-def*
    by (*auto dest*: *tranclp-into-rtranclp*)
  then show *?case* **by** (*auto simp*: *card-mono set-mset-mono*)
next
  case (*restart-step m S T n U*) **note** *st* = *this*(*1*) **and** *U* = *this*(*3*) **and** *n-d* = *this*(*4*) **and**
    inv = *this*(*5*)
  then have *st'*: $cdcl_{NOT}$*-merged-bj-learn*** *S T*
    by (*blast dest*: *relpowp-imp-rtranclp*)
  then have *st''*: $cdcl_{NOT}{}^{**}$ *S T*
    using *inv n-d* **apply** − **by** (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$) *auto*
  have *inv T*
    apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-inv*)
      using *inv st' n-d* **by** *auto*
  then have *inv U*
    using *U* **by** (*auto simp*: *inv-restart*)
  have *not-simplified-cls* ($clauses_{NOT}$ *U*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *T*)
    using ‹*U* ∼ *reduce-trail-to*$_{NOT}$ [] *T*› **by** *auto*
  moreover have *not-simplified-cls* ($clauses_{NOT}$ *T*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *S*)
    apply (*rule rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-not-simplified-decreasing*)
    using ‹($cdcl_{NOT}$*-merged-bj-learn* ^^ *m*) *S T*› **by** (*auto dest*!: *relpowp-imp-rtranclp*)
  ultimately have *U-S*: *not-simplified-cls* ($clauses_{NOT}$ *U*) ⊆# *not-simplified-cls* ($clauses_{NOT}$ *S*)
    by *auto*
  then show *?case* **by** (*auto simp*: *card-mono set-mset-mono*)
qed

sublocale $cdcl_{NOT}$*-increasing-restarts* - - - - - - *f*
  λ*S T*. *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S*
  λ*A S*. *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A*
    ∧ *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧ *finite A*
  μ_{CDCL}*'-merged* $cdcl_{NOT}$*-merged-bj-learn*
  λ*S*. *inv S* ∧ *no-dup* (*trail S*)
  λ*A T*. ((*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))) ∗ *2*
    + *card* (*set-mset* (*not-simplified-cls*($clauses_{NOT}$ *T*)))
    + *3* ^ *card* (*atms-of-ms A*)
  apply *unfold-locales*
    using $cdcl_{NOT}$*-restart-*μ_{CDCL}*'-merged-le-*μ_{CDCL}*'-bound* **apply** *force*
  using $cdcl_{NOT}$*-restart-*μ_{CDCL}*'-bound-le-*μ_{CDCL}*'-bound* **by** *fastforce*

lemma $cdcl_{NOT}$*-restart-eq-sat-iff*:
  assumes
    $cdcl_{NOT}$*-restart S T* **and**
    no-dup (*trail* (*fst S*))
    inv (*fst S*)
  shows *I*⊨*sextm* $clauses_{NOT}$ (*fst S*) ⟷ *I* ⊨*sextm* $clauses_{NOT}$ (*fst T*)

using *assms*
proof (*induction rule*: $cdcl_{NOT}$-*restart.induct*)
case (*restart-full S T n*)
then have $cdcl_{NOT}$-*merged-bj-learn*** *S T*
by (*simp add*: *tranclp-into-rtranclp full1-def*)
then show *?case*
using *rtranclp-*$cdcl_{NOT}$*-bj-sat-ext-iff restart-full.prems*(*1*,*2*)
rtranclp-$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$ **by** *auto*
next
case (*restart-step m S T n U*)
then have $cdcl_{NOT}$-*merged-bj-learn*** *S T*
by (*auto simp*: *tranclp-into-rtranclp full1-def dest*!: *relpowp-imp-rtranclp*)
then have $I \models sextm\ clauses_{NOT}\ S \longleftrightarrow I \models sextm\ clauses_{NOT}\ T$
using *rtranclp-*$cdcl_{NOT}$*-bj-sat-ext-iff restart-step.prems*(*1*,*2*)
rtranclp-$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$ **by** *auto*
moreover have $I \models sextm\ clauses_{NOT}\ T \longleftrightarrow I \models sextm\ clauses_{NOT}\ U$
using *restart-step.hyps*(*3*) **by** *auto*
ultimately show *?case* **by** *auto*
qed

lemma *rtranclp-*$cdcl_{NOT}$*-restart-eq-sat-iff*:
assumes
$cdcl_{NOT}$-*restart*** *S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*))
shows $I \models sextm\ clauses_{NOT}\ (fst\ S) \longleftrightarrow I \models sextm\ clauses_{NOT}\ (fst\ T)$
using *assms*(*1*)
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *cdcl* = *this*(*2*) **and** *IH* = *this*(*3*)
have *inv* (*fst T*) **and** *no-dup* (*trail* (*fst T*))
using *rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv* **using** *st inv n-d* **by** *blast+*
then have $I \models sextm\ clauses_{NOT}\ (fst\ T) \longleftrightarrow I \models sextm\ clauses_{NOT}\ (fst\ U)$
using $cdcl_{NOT}$*-restart-eq-sat-iff cdcl* **by** *blast*
then show *?case* **using** *IH* **by** *blast*
qed

lemma $cdcl_{NOT}$*-restart-all-decomposition-implies-m*:
assumes
$cdcl_{NOT}$*-restart S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**
all-decomposition-implies-m ($clauses_{NOT}$ (*fst S*))
(*get-all-ann-decomposition* (*trail* (*fst S*)))
shows *all-decomposition-implies-m* ($clauses_{NOT}$ (*fst T*))
(*get-all-ann-decomposition* (*trail* (*fst T*)))
using *assms*
proof *induction*
case (*restart-full S T n*) **note** *full* = *this*(*1*) **and** *inv* = *this*(*2*) **and** *n-d* = *this*(*3*) **and**
decomp = *this*(*4*)
have *st*: $cdcl_{NOT}$-*merged-bj-learn*** *S T* **and**
n-s: *no-step* $cdcl_{NOT}$*-merged-bj-learn T*
using *full* **unfolding** *full1-def* **by** (*fast dest*: *tranclp-into-rtranclp*)+
have *st'*: $cdcl_{NOT}$*** *S T*
using *inv rtranclp-*$cdcl_{NOT}$*-merged-bj-learn-is-rtranclp-*$cdcl_{NOT}$*-and-inv st n-d* **by** *auto*
have *inv T*

using *rtranclp-cdcl$_{NOT}$-cdcl$_{NOT}$-inv*[*OF st*] *inv n-d* **by** *auto*
then show *?case*
using *rtranclp-cdcl$_{NOT}$-all-decomposition-implies*[*OF - - n-d decomp*] *st' inv* **by** *auto*
next
case (*restart-step m S T n U*) **note** *st* = *this*(*1*) **and** *U* = *this*(*3*) **and** *inv* = *this*(*4*) **and** *n-d* = *this*(*5*) **and** *decomp* = *this*(*6*)
show *?case* **using** *U* **by** *auto*
qed

lemma *rtranclp-cdcl$_{NOT}$-restart-all-decomposition-implies-m*:
assumes
cdcl$_{NOT}$-restart** S T **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**
decomp: *all-decomposition-implies-m* (*clauses$_{NOT}$* (*fst S*))
(*get-all-ann-decomposition* (*trail* (*fst S*)))
shows *all-decomposition-implies-m* (*clauses$_{NOT}$* (*fst T*))
(*get-all-ann-decomposition* (*trail* (*fst T*)))
using *assms*
proof *induction*
case *base*
then show *?case* **using** *decomp* **by** *simp*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *cdcl* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*−)] **and** *inv* = *this*(*4*) **and** *n-d* = *this*(*5*) **and** *decomp* = *this*(*6*)
have *inv* (*fst T*) **and** *no-dup* (*trail* (*fst T*))
using *rtranclp-cdcl$_{NOT}$-with-restart-cdcl$_{NOT}$-inv* **using** *st inv n-d* **by** *blast+*
then show *?case*
using *cdcl$_{NOT}$-restart-all-decomposition-implies-m*[*OF cdcl*] *IH* **by** *auto*
qed

lemma *full-cdcl$_{NOT}$-restart-normal-form*:
assumes
full: *full cdcl$_{NOT}$-restart S T* **and**
inv: *inv* (*fst S*) **and** *n-d*: *no-dup*(*trail* (*fst S*)) **and**
decomp: *all-decomposition-implies-m* (*clauses$_{NOT}$* (*fst S*))
(*get-all-ann-decomposition* (*trail* (*fst S*))) **and**
atms-cls: *atms-of-mm* (*clauses$_{NOT}$* (*fst S*)) ⊆ *atms-of-ms A* **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail* (*fst S*)) ⊆ *atms-of-ms A* **and**
fin: *finite A*
shows *unsatisfiable* (*set-mset* (*clauses$_{NOT}$* (*fst S*)))
∨ *lits-of-l* (*trail* (*fst T*)) ⊨*sextm clauses$_{NOT}$* (*fst S*) ∧
satisfiable (*set-mset* (*clauses$_{NOT}$* (*fst S*)))
proof −
have *inv-T*: *inv* (*fst T*) **and** *n-d-T*: *no-dup* (*trail* (*fst T*))
using *rtranclp-cdcl$_{NOT}$-with-restart-cdcl$_{NOT}$-inv* **using** *full inv n-d* **unfolding** *full-def* **by** *blast+*
moreover have
atms-cls-T: *atms-of-mm* (*clauses$_{NOT}$* (*fst T*)) ⊆ *atms-of-ms A* **and**
atms-trail-T: *atm-of* ‘ *lits-of-l* (*trail* (*fst T*)) ⊆ *atms-of-ms A*
using *rtranclp-cdcl$_{NOT}$-with-restart-bound-inv*[*of S T A*] *full atms-cls atms-trail fin inv n-d*
unfolding *full-def* **by** *blast+*
ultimately have *no-step cdcl$_{NOT}$-merged-bj-learn* (*fst T*)
apply −
apply (*rule no-step-cdcl$_{NOT}$-restart-no-step-cdcl$_{NOT}$*[*of - A*])
using *full* **unfolding** *full-def* **apply** *simp*
apply *simp*
using *fin* **apply** *simp*

    done
  moreover have *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst T*))
    (*get-all-ann-decomposition* (*trail* (*fst T*)))
    using *rtranclp-cdcl*$_{NOT}$*-restart-all-decomposition-implies-m*[*of S T*] *inv n-d decomp*
    full **unfolding** *full-def* **by** *auto*
  ultimately have *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst T*)))
    ∨ *trail* (*fst T*) ⊨*asm clauses*$_{NOT}$ (*fst T*) ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst T*)))
    apply −
    apply (*rule cdcl*$_{NOT}$*-merged-bj-learn-final-state*)
    using *atms-cls-T atms-trail-T fin n-d-T fin inv-T* **by** *blast*+
  then consider
      (*unsat*) *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst T*)))
    | (*sat*) *trail* (*fst T*) ⊨*asm clauses*$_{NOT}$ (*fst T*) **and** *satisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst T*)))
    by *auto*
  then show *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst S*)))
    ∨ *lits-of-l* (*trail* (*fst T*)) ⊨*sextm clauses*$_{NOT}$ (*fst S*) ∧
      satisfiable (*set-mset* (*clauses*$_{NOT}$ (*fst S*)))
    proof *cases*
      case *unsat*
      then have *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst S*)))
        unfolding *satisfiable-def* **apply** *auto*
        using *rtranclp-cdcl*$_{NOT}$*-restart-eq-sat-iff*[*of S T*] *full inv n-d*
        consistent-true-clss-ext-satisfiable true-clss-imp-true-cls-ext
        unfolding *satisfiable-def full-def* **by** *blast*
      then show *?thesis* **by** *blast*
    next
      case *sat*
      then have *lits-of-l* (*trail* (*fst T*)) ⊨*sextm clauses*$_{NOT}$ (*fst T*)
        using *true-clss-imp-true-cls-ext* **by** (*auto simp*: *true-annots-true-cls*)
      then have *lits-of-l* (*trail* (*fst T*)) ⊨*sextm clauses*$_{NOT}$ (*fst S*)
        using *rtranclp-cdcl*$_{NOT}$*-restart-eq-sat-iff*[*of S T*] *full inv n-d* **unfolding** *full-def* **by** *blast*
      moreover then have *satisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst S*)))
        using *consistent-true-clss-ext-satisfiable distinct-consistent-interp n-d-T* **by** *fast*
      ultimately show *?thesis* **by** *fast*
    qed
qed

corollary *full-cdcl*$_{NOT}$*-restart-normal-form-init-state*:
  assumes
    init-state: *trail S* = [] *clauses*$_{NOT}$ *S* = *N* **and**
    full: *full cdcl*$_{NOT}$*-restart* (*S*, *0*) *T* **and**
    inv: *inv S*
  shows *unsatisfiable* (*set-mset N*)
    ∨ *lits-of-l* (*trail* (*fst T*)) ⊨*sextm N* ∧ *satisfiable* (*set-mset N*)
  using *full-cdcl*$_{NOT}$*-restart-normal-form*[*of* (*S*, *0*) *T*] *assms* **by** *auto*

end

end
theory *DPLL-NOT*
imports *CDCL-NOT*
begin

5.3 DPLL as an instance of NOT

5.3.1 DPLL with simple backtrack

We are using a concrete couple instead of an abstract state.

locale *dpll-with-backtrack*
begin
inductive *backtrack* :: $('v, unit)$ *ann-lits* $\times$ $'v$ *clauses*
  $\Rightarrow$ $('v, unit)$ *ann-lits* $\times$ $'v$ *clauses* $\Rightarrow$ *bool* **where**
backtrack-split $(fst\ S) = (M', L\ \#\ M) \Longrightarrow$ *is-decided* $L \Longrightarrow D \in\#\ snd\ S$
  $\Longrightarrow fst\ S \models as\ CNot\ D \Longrightarrow backtrack\ S\ (Propagated\ (-\ (lit\text{-}of\ L))\ ()\ \#\ M,\ snd\ S)$

inductive-cases *backtrackE*[*elim*]: *backtrack* (M, N) (M', N')
lemma *backtrack-is-backjump*:
  fixes $M\ M'$ :: $('v, unit)$ *ann-lits*
  assumes
    backtrack: *backtrack* (M, N) (M', N') **and**
    no-dup: $(no\text{-}dup \circ fst)$ (M, N) **and**
    decomp: *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
    shows
      $\exists\ C\ F'\ K\ F\ L\ l\ C'$.
        $M = F'\ @\ Decided\ K\ \#\ F\ \wedge$
        $M' = Propagated\ L\ l\ \#\ F \wedge N = N' \wedge C \in\#\ N \wedge F'\ @\ Decided\ K\ \#\ F \models as\ CNot\ C\ \wedge$
        $undefined\text{-}lit\ F\ L \wedge atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ N \cup atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F)\ \wedge$
        $N \models pm\ C' + \{\#L\#\} \wedge F \models as\ CNot\ C'$
proof $-$
  let $?S = (M, N)$
  let $?T = (M', N')$
  obtain $F\ F'\ P\ L\ D$ **where**
    b-sp: *backtrack-split* $M = (F', L\ \#\ F)$ **and**
    is-decided L **and**
    $D \in\#\ snd\ ?S$ **and**
    $M \models as\ CNot\ D$ **and**
    bt: *backtrack* $?S$ $(Propagated\ (-\ (lit\text{-}of\ L))\ P\ \#\ F,\ N)$ **and**
    M': $M' = Propagated\ (-\ (lit\text{-}of\ L))\ P\ \#\ F$ **and**
    [*simp*]: $N' = N$
  using *backtrackE*[*OF backtrack*] **by** (*metis backtrack fstI sndI*)
  let $?K = lit\text{-}of\ L$
  let $?C = image\text{-}mset\ lit\text{-}of\ \{\#K{\in}\#mset\ M.\ is\text{-}decided\ K \wedge K{\neq}L\#\}$ :: $'v$ *clause*
  let $?C' = set\text{-}mset\ (image\text{-}mset\ single\ (?C{+}\{\#?K\#\}))$
  obtain K **where** L: $L = Decided\ K$ **using** ‹*is-decided* L› **by** (*cases* L) *auto*

  have M: $M = F'\ @\ Decided\ K\ \#\ F$
    using *b-sp* **by** (*metis L backtrack-split-list-eq fst-conv snd-conv*)
  moreover have $F'\ @\ Decided\ K\ \#\ F \models as\ CNot\ D$
    using ‹$M \models as\ CNot\ D$› **unfolding** M **.**
  moreover have *undefined-lit* F $(-?K)$
    using *no-dup* **unfolding** $M\ L$ **by** (*simp add*: *defined-lit-map*)
  moreover have $atm\text{-}of\ (-K) \in atms\text{-}of\text{-}mm\ N \cup atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ \#\ F)$
    by *auto*
  moreover
    have $set\text{-}mset\ N \cup ?C' \models ps\ \{\{\#\}\}$
      proof $-$
        have A: $set\text{-}mset\ N \cup ?C' \cup unmark\text{-}l\ M =$
          $set\text{-}mset\ \ N \cup unmark\text{-}l\ M$

unfolding *M L* **by** *auto*
have *set-mset* $N \cup \{\{\#lit\text{-}of\ L\#\} \mid L.\ is\text{-}decided\ L \wedge L \in set\ M\}$
$\models ps\ unmark\text{-}l\ M$
using *all-decomposition-implies-propagated-lits-are-implied*[*OF decomp*] **.**
moreover have C': $?C' = \{\{\#lit\text{-}of\ L\#\} \mid L.\ is\text{-}decided\ L \wedge L \in set\ M\}$
unfolding *M L* **apply** *standard*
apply *force*
using *IntI* **by** *auto*
ultimately have *N-C-M*: *set-mset* $N \cup ?C' \models ps\ unmark\text{-}l\ M$
by *auto*
have *set-mset* $N \cup (\lambda L.\ \{\#lit\text{-}of\ L\#\})$ ‘ $(set\ M) \models ps\ \{\{\#\}\}$
unfolding *true-clss-clss-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 I*) **note** *tot* = *this*(*1*) **and** *cons* = *this*(*2*) **and** *I-N-M* = *this*(*3*)
have $I \models D$
using *I-N-M* ‹$D \in\# snd\ ?S$› **unfolding** *true-clss-def* **by** *auto*
moreover have $I \models s\ CNot\ D$
using ‹$M \models as\ CNot\ D$› **unfolding** *M* **by** (*metis 1*(*3*) ‹$M \models as\ CNot\ D$›
true-annots-true-cls true-cls-mono-set-mset-l true-clss-def
true-clss-singleton-lit-of-implies-incl true-clss-union)
ultimately show *?case* **using** *cons consistent-CNot-not* **by** *blast*
qed
then show *?thesis*
using *true-clss-clss-left-right*[*OF N-C-M*, *of* $\{\{\#\}\}$] **unfolding** *A* **by** *auto*
qed
have $N \models pm\ image\text{-}mset\ uminus\ ?C + \{\#-?K\#\}$
unfolding *true-clss-cls-def true-clss-clss-def total-over-m-def*
proof (*intro allI impI*)
fix *I*
assume
tot: *total-over-set* I (*atms-of-ms* (*set-mset* $N \cup \{image\text{-}mset\ uminus\ ?C + \{\#-\ ?K\#\}\}$)) **and**
cons: *consistent-interp* I **and**
$I \models sm\ N$
have $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$
using *cons tot* **unfolding** *consistent-interp-def L* **by** (*cases K*) *auto*
have $\{a \in set\ M.\ is\text{-}decided\ a \wedge a \neq Decided\ K\} =$
$set\ M \cap \{L.\ is\text{-}decided\ L \wedge L \neq Decided\ K\}$
by *auto*
then have
tI: *total-over-set* I (*atm-of* ‘ *lit-of* ‘ $(set\ M \cap \{L.\ is\text{-}decided\ L \wedge L \neq Decided\ K\})$)
using *tot* **by** (*auto simp add*: *L atms-of-uminus-lit-atm-of-lit-of*)

then have *H*: $\bigwedge x.$
$lit\text{-}of\ x \notin I \Longrightarrow x \in set\ M \Longrightarrow is\text{-}decided\ x$
$\Longrightarrow x \neq Decided\ K \Longrightarrow -lit\text{-}of\ x \in I$
proof −
fix x :: ($'v$, *unit*) *ann-lit*
assume *a1*: $x \neq Decided\ K$
assume *a2*: *is-decided* x
assume *a3*: $x \in set\ M$
assume *a4*: $lit\text{-}of\ x \notin I$
have *atm-of* (*lit-of* x) $\in$ *atm-of* ‘ *lit-of* ‘
$(set\ M \cap \{m.\ is\text{-}decided\ m \wedge m \neq Decided\ K\})$
using *a3 a2 a1* **by** *blast*
then have *Pos* (*atm-of* (*lit-of* x)) $\in I \vee$ *Neg* (*atm-of* (*lit-of* x)) $\in I$
using *tI* **unfolding** *total-over-set-def* **by** *blast*

then show $-$ *lit-of* $x \in I$
using *a4* **by** (*metis* (*no-types*) *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel*(*1*,*2*))
qed
have $\neg I \models s$ *?C'*
using ‹*set-mset N* ∪ *?C'* ⊨*ps* {{#}}› *tot cons* ‹*I* ⊨*sm N*›
unfolding *true-clss-clss-def total-over-m-def*
by (*simp add*: *atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of*)
then show *I* ⊨ *image-mset uminus ?C* + {#− *lit-of L*#}
unfolding *true-clss-def true-cls-def*
using ‹$(K \in I \land -K \notin I) \lor (-K \in I \land K \notin I)$›
unfolding *L* **by** (*auto dest*!: *H*)
qed
moreover
have *set F'* ∩ {*K*. *is-decided K* ∧ *K* ≠ *L*} = {}
using *backtrack-split-fst-not-decided*[*of - M*] *b-sp* **by** *auto*
then have *F* ⊨*as CNot* (*image-mset uminus ?C*)
unfolding *M CNot-def true-annots-def* **by** (*auto simp add*: *L lits-of-def*)
ultimately show *?thesis*
using *M'* ‹*D* ∈# *snd ?S*› *L* **by** *force*
qed

lemma *backtrack-is-backjump'*:
fixes *M M'* :: (*'v*, *unit*) *ann-lits*
assumes
backtrack: *backtrack S T* **and**
no-dup: (*no-dup* ∘ *fst*) *S* **and**
decomp: *all-decomposition-implies-m* (*snd S*) (*get-all-ann-decomposition* (*fst S*))
shows
∃ *C F' K F L l C'*.
fst S = *F'* @ *Decided K* # *F* ∧
T = (*Propagated L l* # *F*, *snd S*) ∧ *C* ∈# *snd S* ∧ *fst S* ⊨*as CNot C*
∧ *undefined-lit F L* ∧ *atm-of L* ∈ *atms-of-mm* (*snd S*) ∪ *atm-of* ‘ *lits-of-l* (*fst S*) ∧
snd S ⊨*pm C'* + {#*L*#} ∧ *F* ⊨*as CNot C'*
apply (*cases S*, *cases T*)
using *backtrack-is-backjump*[*of fst S snd S fst T snd T*] *assms* **by** *fastforce*

sublocale *dpll-state*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*) λ(*M*, *N*). (*tl M*, *N*)
λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*)
by *unfold-locales* (*auto simp*: *ac-simps*)

sublocale *backjumping-ops*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*) λ(*M*, *N*). (*tl M*, *N*)
λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*) λ- - - *S T*. *backtrack S T*
by *unfold-locales*
thm *reduce-trail-to$_{NOT}$-clauses*

lemma *reduce-trail-to$_{NOT}$*:
reduce-trail-to$_{NOT}$ F S =
(*if length* (*fst S*) ≥ *length F*
then drop (*length* (*fst S*) − *length F*) (*fst S*)
else [],
snd S) (**is** *?R* = *?C*)
proof −
have *?R* = (*fst ?R*, *snd ?R*)

by (*cases reduce-trail-to*$_{NOT}$ *F S*) *auto*
also have (*fst ?R*, *snd ?R*) = *?C*
by (*auto simp*: *trail-reduce-trail-to*$_{NOT}$*-drop*)
finally show *?thesis* **.**
qed

lemma *backtrack-is-backjump''*:
fixes *M M'* :: (*'v*, *unit*) *ann-lits*
assumes
backtrack: *backtrack S T* **and**
no-dup: (*no-dup* ∘ *fst*) *S* **and**
decomp: *all-decomposition-implies-m* (*snd S*) (*get-all-ann-decomposition* (*fst S*))
shows *backjump S T*
proof −
obtain *C F' K F L l C'* **where**
1: *fst S* = *F'* @ *Decided K* # *F* **and**
2: *T* = (*Propagated L l* # *F*, *snd S*) **and**
3: *C* ∈# *snd S* **and**
4: *fst S* ⊨*as CNot C* **and**
5: *undefined-lit F L* **and**
6: *atm-of L* ∈ *atms-of-mm* (*snd S*) ∪ *atm-of* ' *lits-of-l* (*fst S*) **and**
7: *snd S* ⊨*pm C'* + {#*L*#} **and**
8: *F* ⊨*as CNot C'*
using *backtrack-is-backjump'*[*OF assms*] **by** *force*
show *?thesis*
apply (*cases S*)
using *backjump.intros*[*OF 1 - - 4 5 - - 8*, *of T*] *2 backtrack 1 5 3 6 7*
by (*auto simp*: *state-eq*$_{NOT}$*-def trail-reduce-trail-to*$_{NOT}$*-drop*
reduce-trail-to$_{NOT}$ *simp del*: *state-simp*$_{NOT}$)
qed

lemma *can-do-bt-step*:
assumes
M: *fst S* = *F'* @ *Decided K* # *F* **and**
C ∈# *snd S* **and**
C: *fst S* ⊨*as CNot C*
shows ¬ *no-step backtrack S*
proof −
obtain *L G' G* **where**
backtrack-split (*fst S*) = (*G'*, *L* # *G*)
unfolding *M* **by** (*induction F' rule*: *ann-lit-list-induct*) *auto*
moreover then have *is-decided L*
by (*metis backtrack-split-snd-hd-decided list.distinct*(*1*) *list.sel*(*1*) *snd-conv*)
ultimately show *?thesis*
using *backtrack.intros*[*of S G' L G C*] ‹*C* ∈# *snd S*› *C* **unfolding** *M* **by** *auto*
qed

end

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping-ops*
fst snd λ*L* (*M*, *N*). (*L* # *M*, *N*)
λ(*M*, *N*). (*tl M*, *N*) λ*C* (*M*, *N*). (*M*, {#*C*#} + *N*) λ*C* (*M*, *N*). (*M*, *removeAll-mset C N*)
λ(*M*, *N*). *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)
λ- - - *S T*. *backtrack S T*
λ- -. *True*
apply *unfold-locales*

by (*metis* (*mono-tags*, *lifting*) *case-prod-beta comp-def dpll-with-backtrack.backtrack-is-backjump''* *dpll-with-backtrack.can-do-bt-step*)

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping*
fst snd λL (M, N). $(L \# M, N)$
$\lambda(M, N)$. (*tl* M, N) λC (M, N). $(M, \{\#C\#\} + N)$ λC (M, N). $(M,$ *removeAll-mset* C $N)$
$\lambda(M, N)$. *no-dup* $M \wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True*
apply *unfold-locales*
using *dpll-bj-no-dup dpll-bj-all-decomposition-implies-inv* **apply** *fastforce*
done

context *dpll-with-backtrack*
begin
lemma *wf-tranclp-dpll-inital-state*:
assumes *fin*: *finite A*
shows *wf* $\{((M'::('v, unit)\ ann\text{-}lits, N'::'v\ clauses), ([], N))|M'\ N'\ N.$
dpll-bj^{++} $([], N)$ (M', N') $\wedge$ *atms-of-mm* $N \subseteq$ *atms-of-ms* $A\}$
using *wf-tranclp-dpll-bj*[*OF assms*(*1*)] **by** (*rule wf-subset*) *auto*

corollary *full-dpll-final-state-conclusive*:
fixes M M' :: $('v, unit)$ *ann-lits*
assumes
full: *full dpll-bj* $([], N)$ (M', N')
shows *unsatisfiable* (*set-mset* N) $\vee$ $(M' \models asm$ $N \wedge$ *satisfiable* (*set-mset* N))
using *assms full-dpll-backjump-final-state*[*of* $([],N)$ (M', N') *set-mset* N] **by** *auto*

corollary *full-dpll-normal-form-from-init-state*:
fixes M M' :: $('v, unit)$ *ann-lits*
assumes
full: *full dpll-bj* $([], N)$ (M', N')
shows $M' \models asm$ $N \longleftrightarrow$ *satisfiable* (*set-mset* N)
proof −
have *no-dup* M'
using *rtranclp-dpll-bj-no-dup*[*of* $([], N)$ (M', N')]
full **unfolding** *full-def* **by** *auto*
then have $M' \models asm$ $N \Longrightarrow$ *satisfiable* (*set-mset* N)
using *distinct-consistent-interp satisfiable-carac' true-annots-true-cls* **by** *blast*
then show *?thesis*
using *full-dpll-final-state-conclusive*[*OF full*] **by** *auto*
qed

interpretation *conflict-driven-clause-learning-ops*
fst snd λL (M, N). $(L \# M, N)$
$\lambda(M, N)$. (*tl* M, N) λC (M, N). $(M, \{\#C\#\} + N)$ λC (M, N). $(M,$ *removeAll-mset* C $N)$
$\lambda(M, N)$. *no-dup* $M \wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True* λ- -. *False* λ- -. *False*
by *unfold-locales*

interpretation *conflict-driven-clause-learning*
fst snd λL (M, N). $(L \# M, N)$
$\lambda(M, N)$. (*tl* M, N) λC (M, N). $(M, \{\#C\#\} + N)$ λC (M, N). $(M,$ *removeAll-mset* C $N)$
$\lambda(M, N)$. *no-dup* $M \wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T

λ- -. *True* λ- -. *False* λ- -. *False*
apply *unfold-locales*
using $cdcl_{NOT}$*-all-decomposition-implies* $cdcl_{NOT}$*-no-dup* **by** *fastforce*

lemma $cdcl_{NOT}$*-is-dpll*:
$cdcl_{NOT}$ S $T \longleftrightarrow$ *dpll-bj* S T
by (*auto simp*: $cdcl_{NOT}$*.simps learn.simps* $forget_{NOT}$*.simps*)

Another proof of termination:

lemma *wf* $\{(T, S)$. *dpll-bj* S $T \wedge$ $cdcl_{NOT}$*-NOT-all-inv* A $S\}$
unfolding $cdcl_{NOT}$*-is-dpll*[*symmetric*]
by (*rule wf-*$cdcl_{NOT}$*-no-learn-and-forget-infinite-chain*)
(*auto simp*: *learn.simps* $forget_{NOT}$*.simps*)
end

5.3.2 Adding restarts

This was mainly a test whether it was possible to instantiate the assumption of the locale.

locale *dpll-withbacktrack-and-restarts* =
dpll-with-backtrack +
fixes f :: *nat* $\Rightarrow$ *nat*
assumes *unbounded*: *unbounded f* **and** *f-ge-1*:$\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$
begin
sublocale $cdcl_{NOT}$*-increasing-restarts*
fst snd λL (M, N). $(L \# M, N)$ $\lambda(M, N)$. (*tl* M, N)
λC (M, N). $(M, \{\#C\#\} + N)$ λC (M, N). (M, *removeAll-mset* C N) f λ(-, N) S. S = ([], N)
λA (M, N). *atms-of-mm* $N \subseteq$ *atms-of-ms* $A \wedge$ *atm-of* ' *lits-of-l* $M \subseteq$ *atms-of-ms* $A \wedge$ *finite* A
$\wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λA T. (*2+card* (*atms-of-ms* A)) ^ (*1+card* (*atms-of-ms* A))
$-$ μ_C (*1+card* (*atms-of-ms* A)) (*2+card* (*atms-of-ms* A)) (*trail-weight* T) *dpll-bj*
$\lambda(M, N)$. *no-dup* $M \wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λA -. (*2+card* (*atms-of-ms* A)) ^ (*1+card* (*atms-of-ms* A))
apply *unfold-locales*
apply (*rule unbounded*)
using *f-ge-1* **apply** *fastforce*
apply (*smt dpll-bj-all-decomposition-implies-inv dpll-bj-atms-in-trail-in-set*
dpll-bj-clauses id-apply prod.case-eq-if)
apply (*rule dpll-bj-trail-mes-decreasing-prop*; *auto*)
apply (*rename-tac* A T U, *case-tac* T, *simp*)
apply (*rename-tac* A T U, *case-tac* U, *simp*)
using *dpll-bj-clauses dpll-bj-all-decomposition-implies-inv dpll-bj-no-dup* **by** *fastforce+*
end

end
theory *DPLL-W*
imports *Main Partial-Clausal-Logic Partial-Annotated-Clausal-Logic List-More Wellfounded-More DPLL-NOT*
begin

5.4 Weidenbach's DPLL

5.4.1 Rules

type-synonym *'a* $dpll_W$*-ann-lit* = (*'a*, *unit*) *ann-lit*
type-synonym *'a* $dpll_W$*-ann-lits* = (*'a*, *unit*) *ann-lits*

type-synonym *'v dpll$_W$-state* = *'v dpll$_W$-ann-lits* × *'v clauses*

abbreviation *trail* :: *'v dpll$_W$-state* ⇒ *'v dpll$_W$-ann-lits* **where**
trail ≡ *fst*
abbreviation *clauses* :: *'v dpll$_W$-state* ⇒ *'v clauses* **where**
clauses ≡ *snd*

inductive *dpll$_W$* :: *'v dpll$_W$-state* ⇒ *'v dpll$_W$-state* ⇒ *bool* **where**
propagate: *C* + {#*L*#} ∈# *clauses S* ⟹ *trail S* ⊨*as CNot C* ⟹ *undefined-lit* (*trail S*) *L*
⟹ *dpll$_W$ S* (*Propagated L* () # *trail S*, *clauses S*) |
decided: *undefined-lit* (*trail S*) *L* ⟹ *atm-of L* ∈ *atms-of-mm* (*clauses S*)
⟹ *dpll$_W$ S* (*Decided L* # *trail S*, *clauses S*) |
backtrack: *backtrack-split* (*trail S*) = (*M'*, *L* # *M*) ⟹ *is-decided L* ⟹ *D* ∈# *clauses S*
⟹ *trail S* ⊨*as CNot D* ⟹ *dpll$_W$ S* (*Propagated* (− (*lit-of L*)) () # *M*, *clauses S*)

5.4.2 Invariants

lemma *dpll$_W$-distinct-inv*:
assumes *dpll$_W$ S S'*
and *no-dup* (*trail S*)
shows *no-dup* (*trail S'*)
using *assms*
proof (*induct rule*: *dpll$_W$.induct*)
case (*decided L S*)
then show *?case* **using** *defined-lit-map* **by** *force*
next
case (*propagate C L S*)
then show *?case* **using** *defined-lit-map* **by** *force*
next
case (*backtrack S M' L M D*) **note** *extracted* = *this*(*1*) **and** *no-dup* = *this*(*5*)
show *?case*
using *no-dup backtrack-split-list-eq*[*of trail S*, *symmetric*] **unfolding** *extracted* **by** *auto*
qed

lemma *dpll$_W$-consistent-interp-inv*:
assumes *dpll$_W$ S S'*
and *consistent-interp* (*lits-of-l* (*trail S*))
and *no-dup* (*trail S*)
shows *consistent-interp* (*lits-of-l* (*trail S'*))
using *assms*
proof (*induct rule*: *dpll$_W$.induct*)
case (*backtrack S M' L M D*) **note** *extracted* = *this*(*1*) **and** *decided* = *this*(*2*) **and** *D* = *this*(*4*) **and**
cons = *this*(*5*) **and** *no-dup* = *this*(*6*)
have *no-dup'*: *no-dup M*
by (*metis* (*no-types*) *backtrack-split-list-eq distinct.simps*(*2*) *distinct-append extracted*
list.simps(*9*) *map-append no-dup snd-conv*)
then have *insert* (*lit-of L*) (*lits-of-l M*) ⊆ *lits-of-l* (*trail S*)
using *backtrack-split-list-eq*[*of trail S*, *symmetric*] **unfolding** *extracted* **by** *auto*
then have *cons*: *consistent-interp* (*insert* (*lit-of L*) (*lits-of-l M*))
using *consistent-interp-subset cons* **by** *blast*
moreover
have *lit-of L* ∉ *lits-of-l M*
using *no-dup backtrack-split-list-eq*[*of trail S*, *symmetric*] *extracted*
unfolding *lits-of-def* **by** *force*
moreover
have *atm-of* (−*lit-of L*) ∉ (λ*m*. *atm-of* (*lit-of m*)) ‘ *set M*

using *no-dup backtrack-split-list-eq*[*of trail S*, *symmetric*] **unfolding** *extracted* **by** *force*
then have $-$*lit-of L* $\notin$ *lits-of-l M*
unfolding *lits-of-def* **by** *force*
ultimately show *?case* **by** *simp*
qed (*auto intro*: *consistent-add-undefined-lit-consistent*)

lemma $dpll_W$*-vars-in-snd-inv*:
assumes $dpll_W$ *S S'*
and *atm-of* ' (*lits-of-l* (*trail S*)) $\subseteq$ *atms-of-mm* (*clauses S*)
shows *atm-of* ' (*lits-of-l* (*trail S'*)) $\subseteq$ *atms-of-mm* (*clauses S'*)
using *assms*
proof (*induct rule*: $dpll_W$*.induct*)
case (*backtrack S M' L M D*)
then have *atm-of* (*lit-of L*) $\in$ *atms-of-mm* (*clauses S*)
using *backtrack-split-list-eq*[*of trail S*, *symmetric*] **by** *auto*
moreover
have *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-mm* (*clauses S*)
using *backtrack*(*5*) **by** *simp*
then have $\bigwedge$*xb. xb* $\in$ *set M* $\Longrightarrow$ *atm-of* (*lit-of xb*) $\in$ *atms-of-mm* (*clauses S*)
using *backtrack-split-list-eq*[*symmetric*, *of trail S*] *backtrack.hyps*(*1*)
unfolding *lits-of-def* **by** *auto*
ultimately show *?case* **by** (*auto simp* : *lits-of-def*)
qed (*auto simp*: *in-plus-implies-atm-of-on-atms-of-ms*)

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms* ((λ*a.* {#*lit-of a*#}) ' *c*) = *atm-of* ' *lit-of* ' *c*
unfolding *atms-of-ms-def* **using** *image-iff* **by** *force*

theorem 2.8.2 page 73 of Weidenbach's book

lemma $dpll_W$*-propagate-is-conclusion*:
assumes $dpll_W$ *S S'*
and *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-mm* (*clauses S*)
shows *all-decomposition-implies-m* (*clauses S'*) (*get-all-ann-decomposition* (*trail S'*))
using *assms*
proof (*induct rule*: $dpll_W$*.induct*)
case (*decided L S*)
then show *?case* **unfolding** *all-decomposition-implies-def* **by** *simp*
next
case (*propagate C L S*) **note** *inS* = *this*(*1*) **and** *cnot* = *this*(*2*) **and** *IH* = *this*(*4*) **and** *undef* = *this*(*3*) **and** *atms-incl* = *this*(*5*)
let *?I* = *set* (*map* (λ*a.* {#*lit-of a*#}) (*trail S*)) $\cup$ *set-mset* (*clauses S*)
have *?I* $\models$*p C* + {#*L*#} **by** (*auto simp add*: *inS*)
moreover have *?I* $\models$*ps CNot C* **using** *true-annots-true-clss-cls cnot* **by** *fastforce*
ultimately have *?I* $\models$*p* {#*L*#} **using** *true-clss-cls-plus-CNot*[*of ?I C L*] *inS* **by** *blast*
{
assume *get-all-ann-decomposition* (*trail S*) = []
then have *?case* **by** *blast*
}
moreover {
assume *n*: *get-all-ann-decomposition* (*trail S*) $\neq$ []
have *1*: $\bigwedge$*a b.* (*a*, *b*) $\in$ *set* (*tl* (*get-all-ann-decomposition* (*trail S*)))
$\Longrightarrow$ (*unmark-l a* $\cup$ *set-mset* (*clauses S*)) $\models$*ps unmark-l b*
using *IH* **unfolding** *all-decomposition-implies-def* **by** (*fastforce simp add*: *list.set-sel*(*2*) *n*)
moreover have *2*: $\bigwedge$*a c. hd* (*get-all-ann-decomposition* (*trail S*)) = (*a*, *c*)
$\Longrightarrow$ (*unmark-l a* $\cup$ *set-mset* (*clauses S*)) $\models$*ps* (*unmark-l c*)
by (*metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single*

list.collapse n)
moreover have *3*: $\bigwedge a\ c.$ *hd* (*get-all-ann-decomposition* (*trail S*)) = (*a*, *c*)
$\Longrightarrow$ (*unmark-l a* ∪ *set-mset* (*clauses S*)) ⊨*p* {#*L*#}
proof −
fix *a c*
assume *h*: *hd* (*get-all-ann-decomposition* (*trail S*)) = (*a*, *c*)
have *h′*: *trail S* = *c* @ *a* **using** *get-all-ann-decomposition-decomp h* **by** *blast*
have *I*: *set* (*map* (λ*a*. {#*lit-of a*#}) *a*) ∪ *set-mset* (*clauses S*)
∪ *unmark-l c* ⊨*ps CNot C*
using ‹*?I* ⊨*ps CNot C*› **unfolding** *h′* **by** (*simp add*: *Un-commute Un-left-commute*)
have
atms-of-ms (*CNot C*) ⊆ *atms-of-ms* (*set* (*map* (λ*a*. {#*lit-of a*#}) *a*) ∪ *set-mset* (*clauses S*))
and
atms-of-ms (*unmark-l c*) ⊆ *atms-of-ms* (*set* (*map* (λ*a*. {#*lit-of a*#}) *a*)
∪ *set-mset* (*clauses S*))
apply (*metis CNot-plus Un-subset-iff atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of atms-of-ms-union inS sup.coboundedI2*)
using *inS atms-of-atms-of-ms-mono atms-incl* **by** (*fastforce simp*: *h′*)

then have *unmark-l a* ∪ *set-mset* (*clauses S*) ⊨*ps CNot C*
using *true-clss-clss-left-right*[*OF - I*] *h 2* **by** *auto*
then show *unmark-l a* ∪ *set-mset* (*clauses S*) ⊨*p* {#*L*#}
by (*metis* (*no-types*) *Un-insert-right inS insertI1 mk-disjoint-insert inS true-clss-cls-in true-clss-cls-plus-CNot*)
qed
ultimately have *?case*
by (*cases hd* (*get-all-ann-decomposition* (*trail S*)))
(*auto simp*: *all-decomposition-implies-def*)
}
ultimately show *?case* **by** *auto*
next
case (*backtrack S M′ L M D*) **note** *extracted* = *this*(*1*) **and** *decided* = *this*(*2*) **and** *D* = *this*(*3*) **and** *cnot* = *this*(*4*) **and** *cons* = *this*(*4*) **and** *IH* = *this*(*5*) **and** *atms-incl* = *this*(*6*)
have *S*: *trail S* = *M′* @ *L* # *M*
using *backtrack-split-list-eq*[*of trail S*] **unfolding** *extracted* **by** *auto*
have *M′*: ∀ *l* ∈ *set M′*. ¬*is-decided l*
using *extracted backtrack-split-fst-not-decided*[*of - trail S*] **by** *simp*
have *n*: *get-all-ann-decomposition* (*trail S*) ≠ [] **by** *auto*
then have *all-decomposition-implies-m* (*clauses S*) ((*L* # *M*, *M′*)
tl (*get-all-ann-decomposition* (*trail S*)))
by (*metis* (*no-types*) *IH extracted get-all-ann-decomposition-backtrack-split list.exhaust-sel*)
then have *1*: *unmark-l* (*L* # *M*) ∪ *set-mset* (*clauses S*) ⊨*ps*(λ*a*.{#*lit-of a*#}) ‘ *set M′*
by *simp*
moreover
have *unmark-l* (*L* # *M*) ∪ *unmark-l M′* ⊨*ps CNot D*
by (*metis* (*mono-tags*, *lifting*) *S Un-commute cons image-Un set-append true-annots-true-clss-clss*)
then have *2*: *unmark-l* (*L* # *M*) ∪ *set-mset* (*clauses S*) ∪ *unmark-l M′*
⊨*ps CNot D*
by (*metis* (*no-types*, *lifting*) *Un-assoc Un-left-commute true-clss-clss-union-l-r*)
ultimately
have *set* (*map* (λ*a*. {#*lit-of a*#}) (*L* # *M*)) ∪ *set-mset* (*clauses S*) ⊨*ps CNot D*
using *true-clss-clss-left-right* **by** *fastforce*
then have *set* (*map* (λ*a*. {#*lit-of a*#}) (*L* # *M*)) ∪ *set-mset* (*clauses S*) ⊨*p* {#}
by (*metis* (*mono-tags*, *lifting*) *D Un-def mem-Collect-eq true-clss-clss-contradiction-true-clss-cls-false*)

then have *IL*: *unmark-l M* ∪ *set-mset* (*clauses S*) ⊨*p* {#−*lit-of L*#}
using *true-clss-clss-false-left-right* **by** *auto*
show *?case* **unfolding** *S all-decomposition-implies-def*
proof
fix *x P level*
assume *x*: *x* ∈ *set* (*get-all-ann-decomposition*
(*fst* (*Propagated* (− *lit-of L*) *P* # *M*, *clauses S*)))
let *?M′* = *Propagated* (− *lit-of L*) *P* # *M*
let *?hd* = *hd* (*get-all-ann-decomposition ?M′*)
let *?tl* = *tl* (*get-all-ann-decomposition ?M′*)
have *x* = *?hd* ∨ *x* ∈ *set ?tl*
using *x*
by (*cases get-all-ann-decomposition ?M′*)
auto
moreover {
assume *x′*: *x* ∈ *set ?tl*
have *L′*: *Decided* (*lit-of L*) = *L* **using** *decided* **by** (*cases L*, *auto*)
have *x* ∈ *set* (*get-all-ann-decomposition* (*M′* @ *L* # *M*))
using *x′ get-all-ann-decomposition-except-last-choice-equal*[*of M′ lit-of L P M*]
L′ **by** (*metis* (*no-types*) *M′ list.set-sel*(*2*) *tl-Nil*)
then have *case x of* (*Ls*, *seen*) ⇒ *unmark-l Ls* ∪ *set-mset* (*clauses S*)
⊨*ps unmark-l seen*
using *decided IH* **by** (*cases L*) (*auto simp add*: *S all-decomposition-implies-def*)
}
moreover {
assume *x′*: *x* = *?hd*
have *tl*: *tl* (*get-all-ann-decomposition* (*M′* @ *L* # *M*)) ≠ []
proof −
have *f1*: ⋀*ms. length* (*get-all-ann-decomposition* (*M′* @ *ms*))
= *length* (*get-all-ann-decomposition ms*)
by (*simp add*: *M′ get-all-ann-decomposition-remove-undecided-length*)
have *Suc* (*length* (*get-all-ann-decomposition M*)) ≠ *Suc 0*
by *blast*
then show *?thesis*
using *f1 decided* **by** (*metis* (*no-types*) *get-all-ann-decomposition.simps*(*1*) *length-tl*
list.sel(*3*) *list.size*(*3*) *ann-lit.collapse*(*1*))
qed
obtain *M0′ M0* **where**
L0: *hd* (*tl* (*get-all-ann-decomposition* (*M′* @ *L* # *M*))) = (*M0*, *M0′*)
by (*cases hd* (*tl* (*get-all-ann-decomposition* (*M′* @ *L* # *M*))))
have *x″*: *x* = (*M0*, *Propagated* (−*lit-of L*) *P* # *M0′*)
unfolding *x′* **using** *get-all-ann-decomposition-last-choice tl M′ L0*
by (*metis decided ann-lit.collapse*(*1*))
obtain *l-get-all-ann-decomposition* **where**
get-all-ann-decomposition (*trail S*) = (*L* # *M*, *M′*) # (*M0*, *M0′*) #
l-get-all-ann-decomposition
using *get-all-ann-decomposition-backtrack-split extracted* **by** (*metis* (*no-types*) *L0 S*
hd-Cons-tl n tl)
then have *M* = *M0′* @ *M0* **using** *get-all-ann-decomposition-hd-hd* **by** *fastforce*
then have *IL′*: *unmark-l M0* ∪ *set-mset* (*clauses S*)
∪ *unmark-l M0′* ⊨*ps* {{#− *lit-of L*#}}
using *IL* **by** (*simp add*: *Un-commute Un-left-commute image-Un*)
moreover have *H*: *unmark-l M0* ∪ *set-mset* (*clauses S*)
⊨*ps unmark-l M0′*
using *IH x″* **unfolding** *all-decomposition-implies-def* **by** (*metis* (*no-types*, *lifting*) *L0 S*
list.set-sel(*1*) *list.set-sel*(*2*) *old.prod.case tl tl-Nil*)

ultimately have *case x of* (*Ls*, *seen*) ⇒ *unmark-l Ls* ∪ *set-mset* (*clauses S*)
⊨*ps unmark-l seen*
using *true-clss-clss-left-right* **unfolding** x'' **by** *auto*
}
ultimately show *case x of* (*Ls*, *seen*) ⇒
unmark-l Ls ∪ *set-mset* (*snd* (*?M′*, *clauses S*))
⊨*ps unmark-l seen*
unfolding *snd-conv* **by** *blast*
qed
qed

theorem 2.8.3 page 73 of Weidenbach's book

theorem *dpll$_W$-propagate-is-conclusion-of-decided*:
assumes *dpll$_W$ S S′*
and *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
shows *set-mset* (*clauses S′*) ∪ {{#*lit-of L*#} |*L*. *is-decided L* ∧ *L* ∈ *set* (*trail S′*)}
⊨*ps* (λ*a*. {#*lit-of a*#}) ‘ ⋃(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition* (*trail S′*)))
using *all-decomposition-implies-trail-is-implied*[*OF dpll$_W$-propagate-is-conclusion*[*OF assms*]] **.**

theorem 2.8.4 page 73 of Weidenbach's book

lemma *only-propagated-vars-unsat*:
assumes *decided*: ∀ *x* ∈ *set M*. ¬ *is-decided x*
and *DN*: *D* ∈ *N* **and** *D*: *M* ⊨*as CNot D*
and *inv*: *all-decomposition-implies N* (*get-all-ann-decomposition M*)
and *atm-incl*: *atm-of* ‘ *lits-of-l M* ⊆ *atms-of-ms N*
shows *unsatisfiable N*
proof (*rule ccontr*)
assume ¬ *unsatisfiable N*
then obtain *I* **where**
I: *I* ⊨*s N* **and**
cons: *consistent-interp I* **and**
tot: *total-over-m I N*
unfolding *satisfiable-def* **by** *auto*
then have *I-D*: *I* ⊨ *D*
using *DN* **unfolding** *true-clss-def* **by** *auto*

have *l0*: {{#*lit-of L*#} |*L*. *is-decided L* ∧ *L* ∈ *set M*} = {} **using** *decided* **by** *auto*
have *atms-of-ms* (*N* ∪ *unmark-l M*) = *atms-of-ms N*
using *atm-incl* **unfolding** *atms-of-ms-def lits-of-def* **by** *auto*

then have *total-over-m I* (*N* ∪ (λ*a*. {#*lit-of a*#}) ‘ (*set M*))
using *tot* **unfolding** *total-over-m-def* **by** *auto*
then have *I* ⊨*s* (λ*a*. {#*lit-of a*#}) ‘ (*set M*)
using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons I*
unfolding *true-clss-clss-def l0* **by** *auto*
then have *IM*: *I* ⊨*s unmark-l M* **by** *auto*
{
fix *K*
assume *K* ∈# *D*
then have −*K* ∈ *lits-of-l M*
by (*auto split*: *if-split-asm*
intro: *allE*[*OF D*[*unfolded true-annots-def Ball-def*], *of* {#−*K*#}])
then have −*K* ∈ *I* **using** *IM true-clss-singleton-lit-of-implies-incl* **by** *fastforce*
}
then have ¬ *I* ⊨ *D* **using** *cons* **unfolding** *true-cls-def consistent-interp-def* **by** *auto*

then show *False* **using** *I-D* **by** *blast*
qed

lemma $dpll_W$*-same-clauses*:
  assumes $dpll_W$ S S'
  shows *clauses* S = *clauses* S'
  using *assms* **by** (*induct rule*: $dpll_W$*.induct*, *auto*)

lemma *rtranclp-*$dpll_W$*-inv*:
  assumes *rtranclp* $dpll_W$ S S'
  and *inv*: *all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (*trail* S))
  and *atm-incl*: *atm-of* ‘ *lits-of-l* (*trail* S) ⊆ *atms-of-mm* (*clauses* S)
  and *consistent-interp* (*lits-of-l* (*trail* S))
  and *no-dup* (*trail* S)
  shows *all-decomposition-implies-m* (*clauses* S') (*get-all-ann-decomposition* (*trail* S'))
  and *atm-of* ‘ *lits-of-l* (*trail* S') ⊆ *atms-of-mm* (*clauses* S')
  and *clauses* S = *clauses* S'
  and *consistent-interp* (*lits-of-l* (*trail* S'))
  and *no-dup* (*trail* S')
  using *assms*
proof (*induct rule*: *rtranclp-induct*)
  case *base*
  show
    all-decomposition-implies-m (*clauses* S) (*get-all-ann-decomposition* (*trail* S)) **and**
    atm-of ‘ *lits-of-l* (*trail* S) ⊆ *atms-of-mm* (*clauses* S) **and**
    clauses S = *clauses* S **and**
    consistent-interp (*lits-of-l* (*trail* S)) **and**
    no-dup (*trail* S) **using** *assms* **by** *auto*
next
  case (*step* S' S'') **note** $dpll_W$*Star* = *this*(*1*) **and** *IH* = *this*(*3*,*4*,*5*,*6*,*7*) **and**
    $dpll_W$ = *this*(*2*)
  moreover
    assume
      inv: *all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (*trail* S)) **and**
      atm-incl: *atm-of* ‘ *lits-of-l* (*trail* S) ⊆ *atms-of-mm* (*clauses* S) **and**
      cons: *consistent-interp* (*lits-of-l* (*trail* S)) **and**
      no-dup (*trail* S)
  ultimately have *decomp*: *all-decomposition-implies-m* (*clauses* S')
    (*get-all-ann-decomposition* (*trail* S')) **and**
    atm-incl': *atm-of* ‘ *lits-of-l* (*trail* S') ⊆ *atms-of-mm* (*clauses* S') **and**
    snd: *clauses* S = *clauses* S' **and**
    cons': *consistent-interp* (*lits-of-l* (*trail* S')) **and**
    no-dup': *no-dup* (*trail* S') **by** *blast+*
  show *clauses* S = *clauses* S'' **using** $dpll_W$*-same-clauses*[*OF* $dpll_W$] *snd* **by** *metis*

  show *all-decomposition-implies-m* (*clauses* S'') (*get-all-ann-decomposition* (*trail* S''))
    using $dpll_W$*-propagate-is-conclusion*[*OF* $dpll_W$] *decomp atm-incl'* **by** *auto*
  show *atm-of* ‘ *lits-of-l* (*trail* S'') ⊆ *atms-of-mm* (*clauses* S'')
    using $dpll_W$*-vars-in-snd-inv*[*OF* $dpll_W$] *atm-incl atm-incl'* **by** *auto*
  show *no-dup* (*trail* S'') **using** $dpll_W$*-distinct-inv*[*OF* $dpll_W$] *no-dup'* $dpll_W$ **by** *auto*
  show *consistent-interp* (*lits-of-l* (*trail* S''))
    using *cons' no-dup'* $dpll_W$*-consistent-interp-inv*[*OF* $dpll_W$] **by** *auto*
qed

definition $dpll_W$*-all-inv* S ≡
  (*all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (*trail* S))

$\wedge$ *atm-of ‘ lits-of-l* (*trail S*) $\subseteq$ *atms-of-mm* (*clauses S*)
$\wedge$ *consistent-interp* (*lits-of-l* (*trail S*))
$\wedge$ *no-dup* (*trail S*))

lemma *dpll*$_W$*-all-inv-dest*[*dest*]:
assumes *dpll*$_W$*-all-inv S*
shows *all-decomposition-implies-m* (*clauses S*) (*get-all-ann-decomposition* (*trail S*))
and *atm-of ‘ lits-of-l* (*trail S*) $\subseteq$ *atms-of-mm* (*clauses S*)
and *consistent-interp* (*lits-of-l* (*trail S*)) $\wedge$ *no-dup* (*trail S*)
using *assms* **unfolding** *dpll*$_W$*-all-inv-def lits-of-def* **by** *auto*

lemma *rtranclp-dpll*$_W$*-all-inv*:
assumes *rtranclp dpll*$_W$ *S S′*
and *dpll*$_W$*-all-inv S*
shows *dpll*$_W$*-all-inv S′*
using *assms rtranclp-dpll*$_W$*-inv*[*OF assms*(*1*)] **unfolding** *dpll*$_W$*-all-inv-def lits-of-def* **by** *blast*

lemma *dpll*$_W$*-all-inv*:
assumes *dpll*$_W$ *S S′*
and *dpll*$_W$*-all-inv S*
shows *dpll*$_W$*-all-inv S′*
using *assms rtranclp-dpll*$_W$*-all-inv* **by** *blast*

lemma *rtranclp-dpll*$_W$*-inv-starting-from-0*:
assumes *rtranclp dpll*$_W$ *S S′*
and *inv*: *trail S* = []
shows *dpll*$_W$*-all-inv S′*
proof −
have *dpll*$_W$*-all-inv S*
using *assms* **unfolding** *all-decomposition-implies-def dpll*$_W$*-all-inv-def* **by** *auto*
then show *?thesis* **using** *rtranclp-dpll*$_W$*-all-inv*[*OF assms*(*1*)] **by** *blast*
qed

lemma *dpll*$_W$*-can-do-step*:
assumes *consistent-interp* (*set M*)
and *distinct M*
and *atm-of ‘* (*set M*) $\subseteq$ *atms-of-mm N*
shows *rtranclp dpll*$_W$ ([], *N*) (*map Decided M*, *N*)
using *assms*
proof (*induct M*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons L M*)
then have *undefined-lit* (*map Decided M*) *L*
unfolding *defined-lit-def consistent-interp-def* **by** *auto*
moreover have *atm-of L* $\in$ *atms-of-mm N* **using** *Cons.prems*(*3*) **by** *auto*
ultimately have *dpll*$_W$ (*map Decided M*, *N*) (*map Decided* (*L* # *M*), *N*)
using *dpll*$_W$*.decided* **by** *auto*
moreover have *consistent-interp* (*set M*) **and** *distinct M* **and** *atm-of ‘ set M* $\subseteq$ *atms-of-mm N*
using *Cons.prems* **unfolding** *consistent-interp-def* **by** *auto*
ultimately show *?case* **using** *Cons.hyps* **by** *auto*
qed

definition *conclusive-dpll*$_W$*-state* (*S*:: *′v dpll*$_W$*-state*) $\longleftrightarrow$
(*trail S* $\models$*asm clauses S* $\vee$ (($\forall$ *L* $\in$ *set* (*trail S*). $\neg$*is-decided L*)

$\wedge$ ($\exists$ *C* $\in\#$ *clauses S*. *trail S* $\models$*as CNot C*)))

theorem 2.8.6 page 74 of Weidenbach's book

lemma *dpll$_W$-strong-completeness*:
  assumes *set M* $\models$*sm N*
  and *consistent-interp* (*set M*)
  and *distinct M*
  and *atm-of* ‘ (*set M*) $\subseteq$ *atms-of-mm N*
  shows *dpll$_W$*** ([], *N*) (*map Decided M*, *N*)
  and *conclusive-dpll$_W$-state* (*map Decided M*, *N*)
proof –
  show *rtranclp dpll$_W$* ([], *N*) (*map Decided M*, *N*) **using** *dpll$_W$-can-do-step assms* **by** *auto*
  have *map Decided M* $\models$*asm N* **using** *assms*(*1*) *true-annots-decided-true-cls* **by** *auto*
  then show *conclusive-dpll$_W$-state* (*map Decided M*, *N*)
    unfolding *conclusive-dpll$_W$-state-def* **by** *auto*
qed

theorem 2.8.5 page 73 of Weidenbach's book

lemma *dpll$_W$-sound*:
  assumes
    rtranclp dpll$_W$ ([], *N*) (*M*, *N*) **and**
    $\forall$ *S*. $\neg$*dpll$_W$* (*M*, *N*) *S*
  shows *M* $\models$*asm N* $\longleftrightarrow$ *satisfiable* (*set-mset N*) (**is** *?A* $\longleftrightarrow$ *?B*)
proof
  let *?M′*= *lits-of-l M*
  assume *?A*
  then have *?M′* $\models$*sm N* **by** (*simp add*: *true-annots-true-cls*)
  moreover have *consistent-interp ?M′*
    using *rtranclp-dpll$_W$-inv-starting-from-0*[*OF assms*(*1*)] **by** *auto*
  ultimately show *?B* **by** *auto*
next
  assume *?B*
  show *?A*
    proof (*rule ccontr*)
      assume *n*: $\neg$ *?A*
      have ($\exists$ *L*. *undefined-lit M L* $\wedge$ *atm-of L* $\in$ *atms-of-mm N*) $\vee$ ($\exists$ *D*$\in\#$*N*. *M* $\models$*as CNot D*)
        proof –
          obtain *D* :: *′a clause* **where** *D*: *D* $\in\#$ *N* **and** $\neg$ *M* $\models$*a D*
            using *n* **unfolding** *true-annots-def Ball-def* **by** *auto*
          then have ($\exists$ *L*. *undefined-lit M L* $\wedge$ *atm-of L* $\in$ *atms-of D*) $\vee$ *M* $\models$*as CNot D*
             unfolding *true-annots-def Ball-def CNot-def true-annot-def*
             using *atm-of-lit-in-atms-of true-annot-iff-decided-or-true-lit true-cls-def* **by** *blast*
          then show *?thesis*
            by (*metis Bex-def D atms-of-atms-of-ms-mono rev-subsetD*)
        qed
      moreover {
        assume $\exists$ *L*. *undefined-lit M L* $\wedge$ *atm-of L* $\in$ *atms-of-mm N*
        then have *False* **using** *assms*(*2*) *decided* **by** *fastforce*
      }
      moreover {
        assume $\exists$ *D*$\in\#$*N*. *M* $\models$*as CNot D*
        then obtain *D* **where** *DN*: *D* $\in\#$ *N* **and** *MD*: *M* $\models$*as CNot D* **by** *auto*
        {
          assume $\forall$ *l* $\in$ *set M*. $\neg$ *is-decided l*
          moreover have *dpll$_W$-all-inv* ([], *N*)
            using *assms* **unfolding** *all-decomposition-implies-def dpll$_W$-all-inv-def* **by** *auto*

ultimately have *unsatisfiable* (*set-mset N*)
using *only-propagated-vars-unsat*[*of M D set-mset N*] *DN MD*
rtranclp-dpll$_W$*-all-inv*[*OF assms*(*1*)] **by** *force*
then have *False* **using** ⟨*?B*⟩ **by** *blast*
}
moreover {
assume *l*: ∃ *l* ∈ *set M*. *is-decided l*
then have *False*
using *backtrack*[*of* (*M*, *N*) - - - *D*] *DN MD assms*(*2*)
backtrack-split-some-is-decided-then-snd-has-hd[*OF l*]
by (*metis backtrack-split-snd-hd-decided fst-conv list.distinct*(*1*) *list.sel*(*1*) *snd-conv*)
}
ultimately have *False* **by** *blast*
}
ultimately show *False* **by** *blast*
qed
qed

5.4.3 Termination

definition *dpll*$_W$*-mes M n* =
map (λ*l*. *if is-decided l then 2 else* (*1*::*nat*)) (*rev M*) @ *replicate* (*n* − *length M*) *3*

lemma *length-dpll*$_W$*-mes*:
assumes *length M* ≤ *n*
shows *length* (*dpll*$_W$*-mes M n*) = *n*
using *assms* **unfolding** *dpll*$_W$*-mes-def* **by** *auto*

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

lemma *dpll*$_W$*-card-decrease*:
assumes *dpll*: *dpll*$_W$ *S S′* **and** *length* (*trail S′*) ≤ *card vars*
and *length* (*trail S*) ≤ *card vars*
shows (*dpll*$_W$*-mes* (*trail S′*) (*card vars*), *dpll*$_W$*-mes* (*trail S*) (*card vars*))
∈ *lexn* {(*a*, *b*). *a* < *b*} (*card vars*)
using *assms*
proof (*induct rule*: *dpll*$_W$*.induct*)
case (*propagate C L S*)
have *m*: *map* (λ*l*. *if is-decided l then 2 else 1*) (*rev* (*trail S*))
@ *replicate* (*card vars* − *length* (*trail S*)) *3*
= *map* (λ*l*. *if is-decided l then 2 else 1*) (*rev* (*trail S*)) @ *3*
replicate (*card vars* − *Suc* (*length* (*trail S*))) *3*
using *propagate.prems*[*simplified*] **using** *Suc-diff-le* **by** *fastforce*
then show *?case*
using *propagate.prems*(*1*) **unfolding** *dpll*$_W$*-mes-def* **by** (*fastforce simp add*: *lexn-conv assms*(*2*))
next
case (*decided S L*)
have *m*: *map* (λ*l*. *if is-decided l then 2 else 1*) (*rev* (*trail S*))
@ *replicate* (*card vars* − *length* (*trail S*)) *3*
= *map* (λ*l*. *if is-decided l then 2 else 1*) (*rev* (*trail S*)) @ *3*
replicate (*card vars* − *Suc* (*length* (*trail S*))) *3*
using *decided.prems*[*simplified*] **using** *Suc-diff-le* **by** *fastforce*
then show *?case*

    using *decided.prems* **unfolding** $dpll_W$*-mes-def* **by** (*force simp add*: *lexn-conv assms*(*2*))
next
  case (*backtrack S M' L M D*)
  have *L*: *is-decided L* **using** *backtrack.hyps*(*2*) **by** *auto*
  have *S*: *trail S* = *M'* @ *L* # *M*
    using *backtrack.hyps*(*1*) *backtrack-split-list-eq*[*of trail S*] **by** *auto*
  show *?case*
    using *backtrack.prems L* **unfolding** $dpll_W$*-mes-def S* **by** (*fastforce simp add*: *lexn-conv assms*(*2*))
qed

theorem 2.8.7 page 74 of Weidenbach's book

lemma $dpll_W$*-card-decrease'*:
  assumes *dpll*: $dpll_W$ *S S'*
  and *atm-incl*: *atm-of ' lits-of-l* (*trail S*) ⊆ *atms-of-mm* (*clauses S*)
  and *no-dup*: *no-dup* (*trail S*)
  shows ($dpll_W$*-mes* (*trail S'*) (*card* (*atms-of-mm* (*clauses S'*))),
        $dpll_W$*-mes* (*trail S*) (*card* (*atms-of-mm* (*clauses S*)))) ∈ *lex* {(*a*, *b*). *a* < *b*}
proof −
  have *finite* (*atms-of-mm* (*clauses S*)) **unfolding** *atms-of-ms-def* **by** *auto*
  then have *1*: *length* (*trail S*) ≤ *card* (*atms-of-mm* (*clauses S*))
    using *distinctcard-atm-of-lit-of-eq-length*[*OF no-dup*] *atm-incl card-mono* **by** *metis*

  moreover
    have *no-dup'*: *no-dup* (*trail S'*) **using** *dpll* $dpll_W$*-distinct-inv no-dup* **by** *blast*
    have *SS'*: *clauses S'* = *clauses S* **using** *dpll* **by** (*auto dest*!: $dpll_W$*-same-clauses*)
    have *atm-incl'*: *atm-of ' lits-of-l* (*trail S'*) ⊆ *atms-of-mm* (*clauses S'*)
      using *atm-incl dpll* $dpll_W$*-vars-in-snd-inv*[*OF dpll*] **by** *force*
    have *finite* (*atms-of-mm* (*clauses S'*))
      unfolding *atms-of-ms-def* **by** *auto*
    then have *2*: *length* (*trail S'*) ≤ *card* (*atms-of-mm* (*clauses S*))
      using *distinctcard-atm-of-lit-of-eq-length*[*OF no-dup'*] *atm-incl' card-mono SS'* **by** *metis*

  ultimately have ($dpll_W$*-mes* (*trail S'*) (*card* (*atms-of-mm* (*clauses S*))),
      $dpll_W$*-mes* (*trail S*) (*card* (*atms-of-mm* (*clauses S*))))
    ∈ *lexn* {(*a*, *b*). *a* < *b*} (*card* (*atms-of-mm* (*clauses S*)))
    using $dpll_W$*-card-decrease*[*OF assms*(*1*), *of atms-of-mm* (*clauses S*)] **by** *blast*
  then have ($dpll_W$*-mes* (*trail S'*) (*card* (*atms-of-mm* (*clauses S*))),
        $dpll_W$*-mes* (*trail S*) (*card* (*atms-of-mm* (*clauses S*)))) ∈ *lex* {(*a*, *b*). *a* < *b*}
    unfolding *lex-def* **by** *auto*
  then show ($dpll_W$*-mes* (*trail S'*) (*card* (*atms-of-mm* (*clauses S'*))),
        $dpll_W$*-mes* (*trail S*) (*card* (*atms-of-mm* (*clauses S*)))) ∈ *lex* {(*a*, *b*). *a* < *b*}
    using $dpll_W$*-same-clauses*[*OF assms*(*1*)] **by** *auto*
qed

lemma *wf-lexn*: *wf* (*lexn* {(*a*, *b*). (*a*::*nat*) < *b*} (*card* (*atms-of-mm* (*clauses S*))))
proof −
  have *m*: {(*a*, *b*). *a* < *b*} = *measure id* **by** *auto*
  show *?thesis* **apply** (*rule wf-lexn*) **unfolding** *m* **by** *auto*
qed

lemma $dpll_W$*-wf*:
  wf {(*S'*, *S*). $dpll_W$*-all-inv S* ∧ $dpll_W$ *S S'*}
  apply (*rule wf-wf-if-measure'*[*OF wf-lex-less*, *of - -*
        λ*S*. $dpll_W$*-mes* (*trail S*) (*card* (*atms-of-mm* (*clauses S*)))])
  using $dpll_W$*-card-decrease'* **by** *fast*

lemma $dpll_W$*-tranclp-star-commute*:
$\{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}^+ = \{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge tranclp\ dpll_W\ S\ S'\}$
(**is** $?A = ?B$)
proof
{ **fix** $S\ S'$
assume $(S, S') \in ?A$
then have $(S, S') \in ?B$
by (*induct rule*: *trancl.induct*, *auto*)
}
then show $?A \subseteq ?B$ **by** *blast*
{ **fix** $S\ S'$
assume $(S, S') \in ?B$
then have $dpll_W{}^{++}\ S'\ S$ **and** $dpll_W$*-all-inv* S' **by** *auto*
then have $(S, S') \in ?A$
proof (*induct rule*: *tranclp.induct*)
case *r-into-trancl*
then show *?case* **by** (*simp-all add*: *r-into-trancl′*)
next
case (*trancl-into-trancl* $S\ S'\ S''$)
then have $(S', S) \in \{a.\ case\ a\ of\ (S', S) \Rightarrow dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}^+$ **by** *blast*
moreover have $dpll_W$*-all-inv* S'
using *rtranclp-*$dpll_W$*-all-inv*[*OF tranclp-into-rtranclp*[*OF trancl-into-trancl.hyps*(*1*)]]
trancl-into-trancl.prems **by** *auto*
ultimately have $(S'', S') \in \{(pa, p).\ dpll_W\text{-}all\text{-}inv\ p \wedge dpll_W\ p\ pa\}^+$
using ‹$dpll_W$*-all-inv* S'› *trancl-into-trancl.hyps*(*3*) **by** *blast*
then show *?case*
using ‹$(S', S) \in \{a.\ case\ a\ of\ (S', S) \Rightarrow dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}^+$› **by** *auto*
qed
}
then show $?B \subseteq ?A$ **by** *blast*
qed

lemma $dpll_W$*-wf-tranclp*: *wf* $\{(S', S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W{}^{++}\ S\ S'\}$
unfolding $dpll_W$*-tranclp-star-commute*[*symmetric*] **by** (*simp add*: $dpll_W$*-wf wf-trancl*)

lemma $dpll_W$*-wf-plus*:
shows *wf* $\{(S', ([], N))|\ S'.\ dpll_W{}^{++}\ ([], N)\ S'\}$ (**is** *wf ?P*)
apply (*rule wf-subset*[*OF* $dpll_W$*-wf-tranclp*, *of ?P*])
using *assms* **unfolding** $dpll_W$*-all-inv-def* **by** *auto*

5.4.4 Final States

Proposition 2.8.1: final states are the normal forms of $dpll_W$

lemma $dpll_W$*-no-more-step-is-a-conclusive-state*:
assumes $\forall S'.\ \neg dpll_W\ S\ S'$
shows *conclusive-*$dpll_W$*-state* S
proof −
have *vars*: $\forall s \in atms\text{-}of\text{-}mm\ (clauses\ S).\ s \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S)$
proof (*rule ccontr*)
assume $\neg (\forall s{\in}atms\text{-}of\text{-}mm\ (clauses\ S).\ s \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S))$
then obtain L **where**
L-in-atms: $L \in atms\text{-}of\text{-}mm\ (clauses\ S)$ **and**
L-notin-trail: $L \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S)$ **by** *metis*
obtain L' **where** L': $atm\text{-}of\ L' = L$ **by** (*meson literal.sel*(*2*))

then have *undefined-lit* (*trail S*) L'
  unfolding *Decided-Propagated-in-iff-in-lits-of-l* **by** (*metis L-notin-trail atm-of-uminus imageI*)
 then show *False* **using** $dpll_W$*.decided assms*(*1*) *L-in-atms* L' **by** *blast*
 qed
 show *?thesis*
 proof (*rule ccontr*)
 assume *not-final*: ¬ *?thesis*
 then have
 ¬ *trail S* ⊨*asm clauses S* **and**
 (∃ *L*∈*set* (*trail S*). *is-decided L*) ∨ (∀ *C*∈#*clauses S*. ¬*trail S* ⊨*as CNot C*)
 unfolding *conclusive-*$dpll_W$*-state-def* **by** *auto*
 moreover {
 assume ∃ *L*∈*set* (*trail S*). *is-decided L*
 then obtain *L* M' *M* **where** *L*: *backtrack-split* (*trail S*) = (M', *L* # *M*)
 using *backtrack-split-some-is-decided-then-snd-has-hd* **by** *blast*
 obtain *D* **where** *D* ∈# *clauses S* **and** ¬ *trail S* ⊨*a D*
 using ‹¬ *trail S* ⊨*asm clauses S*› **unfolding** *true-annots-def* **by** *auto*
 then have ∀ *s*∈*atms-of-ms* {*D*}. *s* ∈ *atm-of* ‘ *lits-of-l* (*trail S*)
 using *vars* **unfolding** *atms-of-ms-def* **by** *auto*
 then have *trail S* ⊨*as CNot D*
 using *all-variables-defined-not-imply-cnot*[*of D*] ‹¬ *trail S* ⊨*a D*› **by** *auto*
 moreover have *is-decided L*
 using *L* **by** (*metis backtrack-split-snd-hd-decided list.distinct*(*1*) *list.sel*(*1*) *snd-conv*)
 ultimately have *False*
 using *assms*(*1*) $dpll_W$*.backtrack L* ‹*D* ∈# *clauses S*› ‹*trail S* ⊨*as CNot D*› **by** *blast*
 }
 moreover {
 assume *tr*: ∀ *C*∈#*clauses S*. ¬*trail S* ⊨*as CNot C*
 obtain *C* **where** *C-in-cls*: *C* ∈# *clauses S* **and** *trC*: ¬ *trail S* ⊨*a C*
 using ‹¬ *trail S* ⊨*asm clauses S*› **unfolding** *true-annots-def* **by** *auto*
 have ∀ *s*∈*atms-of-ms* {*C*}. *s* ∈ *atm-of* ‘ *lits-of-l* (*trail S*)
 using *vars* ‹*C* ∈# *clauses S*› **unfolding** *atms-of-ms-def* **by** *auto*
 then have *trail S* ⊨*as CNot C*
 by (*meson C-in-cls tr trC all-variables-defined-not-imply-cnot*)
 then have *False* **using** *tr C-in-cls* **by** *auto*
 }
 ultimately show *False* **by** *blast*
 qed
qed

lemma $dpll_W$*-conclusive-state-correct*:
 assumes $dpll_W{}^{**}$ ([], *N*) (*M*, *N*) **and** *conclusive-*$dpll_W$*-state* (*M*, *N*)
 shows *M* ⊨*asm N* ⟷ *satisfiable* (*set-mset N*) (**is** *?A* ⟷ *?B*)
proof
 let *?M'*= *lits-of-l M*
 assume *?A*
 then have *?M'* ⊨*sm N* **by** (*simp add*: *true-annots-true-cls*)
 moreover have *consistent-interp ?M'*
 using *rtranclp-*$dpll_W$*-inv-starting-from-0*[*OF assms*(*1*)] **by** *auto*
 ultimately show *?B* **by** *auto*
next
 assume *?B*
 show *?A*
 proof (*rule ccontr*)
 assume *n*: ¬ *?A*
 have *no-mark*: ∀ *L*∈*set M*. ¬ *is-decided L* ∃ *C* ∈# *N*. *M* ⊨*as CNot C*

using *n assms(2)* **unfolding** *conclusive-dpll$_W$-state-def* **by** *auto*
moreover obtain *D* **where** *DN*: $D \in\# N$ **and** *MD*: $M \models as$ *CNot D* **using** *no-mark* **by** *auto*
ultimately have *unsatisfiable (set-mset N)*
using *only-propagated-vars-unsat rtranclp-dpll$_W$-all-inv[OF assms(1)]*
unfolding *dpll$_W$-all-inv-def* **by** *force*
then show *False* **using** ‹*?B*› **by** *blast*
qed
qed

5.4.5 Link with NOT's DPLL

interpretation *dpll$_W$-$_{NOT}$*: *dpll-with-backtrack* **.**

declare *dpll$_W$-$_{NOT}$.state-simp$_{NOT}$[simp del]*
lemma *state-eq$_{NOT}$-iff-eq[iff, simp]*: *dpll$_W$-$_{NOT}$.state-eq$_{NOT}$ S T* $\longleftrightarrow S = T$
unfolding *dpll$_W$-$_{NOT}$.state-eq$_{NOT}$-def* **by** *(cases S, cases T) auto*
lemma *dpll$_W$-dpll$_W$-bj*:
assumes *inv*: *dpll$_W$-all-inv S* **and** *dpll*: *dpll$_W$ S T*
shows *dpll$_W$-$_{NOT}$.dpll-bj S T*
using *dpll inv*
apply *(induction rule: dpll$_W$.induct)*
apply *(rule dpll$_W$-$_{NOT}$.bj-propagate$_{NOT}$)*
apply *(rule dpll$_W$-$_{NOT}$.propagate$_{NOT}$.propagate$_{NOT}$; simp?)*
apply *fastforce*
apply *(rule dpll$_W$-$_{NOT}$.bj-decide$_{NOT}$)*
apply *(rule dpll$_W$-$_{NOT}$.decide$_{NOT}$.decide$_{NOT}$; simp?)*
apply *fastforce*
apply *(frule dpll$_W$-$_{NOT}$.backtrack.intros[of - - - - -], simp-all)*
apply *(rule dpll$_W$-$_{NOT}$.dpll-bj.bj-backjump)*
apply *(rule dpll$_W$-$_{NOT}$.backtrack-is-backjump″,*
simp-all add: dpll$_W$-all-inv-def)
done

lemma *dpll$_W$-bj-dpll*:
assumes *inv*: *dpll$_W$-all-inv S* **and** *dpll*: *dpll$_W$-$_{NOT}$.dpll-bj S T*
shows *dpll$_W$ S T*
using *dpll*
apply *(induction rule: dpll$_W$-$_{NOT}$.dpll-bj.induct)*
apply *(elim dpll$_W$-$_{NOT}$.decide$_{NOT}$E, cases S)*
apply *(frule decided; simp)*

apply *(elim dpll$_W$-$_{NOT}$.propagate$_{NOT}$E, cases S)*
apply *(auto intro!: propagate[of - - (-, -), simplified])[]*
apply *(elim dpll$_W$-$_{NOT}$.backjumpE, cases S)*
by *(simp add: dpll$_W$.simps dpll-with-backtrack.backtrack.simps)*

lemma *rtranclp-dpll$_W$-rtranclp-dpll$_W$-$_{NOT}$*:
assumes *dpll$_W$** S T* **and** *dpll$_W$-all-inv S*
shows *dpll$_W$-$_{NOT}$.dpll-bj** S T*
using *assms* **apply** *(induction)*
apply *simp*
by *(auto intro: rtranclp-dpll$_W$-all-inv dpll$_W$-dpll$_W$-bj rtranclp.rtrancl-into-rtrancl)*

lemma *rtranclp-dpll-rtranclp-dpll$_W$*:
assumes *dpll$_W$-$_{NOT}$.dpll-bj** S T* **and** *dpll$_W$-all-inv S*
shows *dpll$_W$** S T*

using *assms* **apply** (*induction*)
apply *simp*
by (*auto intro*: $dpll_W$*-bj-dpll rtranclp.rtrancl-into-rtrancl rtranclp-*$dpll_W$*-all-inv*)

lemma *dpll-conclusive-state-correctness*:
assumes $dpll_W$*-*$_{NOT}$*.dpll-bj*** ([], *N*) (*M*, *N*) **and** *conclusive-*$dpll_W$*-state* (*M*, *N*)
shows *M* $\models$*asm N* $\longleftrightarrow$ *satisfiable* (*set-mset N*)
proof −
have $dpll_W$*-all-inv* ([], *N*)
unfolding $dpll_W$*-all-inv-def* **by** *auto*
show *?thesis*
apply (*rule* $dpll_W$*-conclusive-state-correct*)
apply (*simp add*: ‹$dpll_W$*-all-inv* ([], *N*)› *assms*(*1*) *rtranclp-dpll-rtranclp-*$dpll_W$)
using *assms*(*2*) **by** *simp*
qed

end
theory *CDCL-W-Level*
imports *Partial-Annotated-Clausal-Logic*
begin

Level of literals and clauses

Getting the level of a variable, implies that the list has to be reversed. Here is the function after reversing.

abbreviation *count-decided* :: ('*v*, '*m*) *ann-lits* ⇒ *nat* **where**
count-decided l ≡ *length* (*filter is-decided l*)

abbreviation *get-level* :: ('*v*, '*m*) *ann-lits* ⇒ '*v literal* ⇒ *nat* **where**
get-level S L ≡ *length* (*filter is-decided* (*dropWhile* (λ*S*. *atm-of* (*lit-of S*) ≠ *atm-of L*) *S*))

lemma *get-level-uminus*: *get-level M* (−*L*) = *get-level M L*
by *auto*

lemma *atm-of-notin-get-rev-level-eq-0*[*simp*]:
assumes *atm-of L* ∉ *atm-of* ‘ *lits-of-l M*
shows *get-level M L* = *0*
using *assms* **by** (*induct M rule*: *ann-lit-list-induct*) *auto*

lemma *get-level-ge-0-atm-of-in*:
assumes *get-level M L* > *n*
shows *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*
using *assms* **by** (*induct M arbitrary*: *n rule*: *ann-lit-list-induct*) *fastforce+*

In *get-level* (resp. *get-level*), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

lemma *get-rev-level-skip*[*simp*]:
assumes *atm-of L* ∉ *atm-of* ‘ *lits-of-l M*
shows *get-level* (*M* @ *M'*) *L* = *get-level M' L*
using *assms* **by** (*induct M rule*: *ann-lit-list-induct*) *auto*

If the literal is at the beginning, then the end can be skipped

lemma *get-rev-level-skip-end*[*simp*]:
assumes *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*

shows *get-level* $(M\ @\ M')$ L = *get-level* M L + *length* (*filter is-decided* M')
using *assms* **by** (*induct* M' *rule*: *ann-lit-list-induct*) (*auto simp*: *lits-of-def*)

lemma *get-level-skip-beginning*:
assumes *atm-of* $L' \neq$ *atm-of* (*lit-of* K)
shows *get-level* $(K\ \#\ M)$ L' = *get-level* M L'
using *assms* **by** *auto*

lemma *get-level-skip-beginning-not-decided*[*simp*]:
assumes *atm-of* $L \notin$ *atm-of* ‘ *lits-of-l* S
and $\forall s \in$ *set* S. $\neg$*is-decided* s
shows *get-level* $(M\ @\ S)$ L = *get-level* M L
using *assms* **apply** (*induction* S *rule*: *ann-lit-list-induct*)
apply *auto*[*2*]
apply (*case-tac atm-of* $L \in$ *atm-of* ‘ *lits-of-l* M)
apply (*auto simp*: *image-iff lits-of-def filter-empty-conv dest*: *set-dropWhileD*)
done

lemma *get-level-skip-in-all-not-decided*:
fixes M :: $('a, 'b)$ *ann-lits* **and** L :: $'a$ *literal*
assumes $\forall m \in$ *set* M. $\neg$ *is-decided* m
and *atm-of* $L \in$ *atm-of* ‘ *lits-of-l* M
shows *get-level* M L = *0*
using *assms* **by** (*induction* M *rule*: *ann-lit-list-induct*) *auto*

lemma *get-level-skip-all-not-decided*[*simp*]:
fixes M
assumes $\forall m \in$ *set* M. $\neg$ *is-decided* m
shows *get-level* M L = *0*
using *assms* **by** (*auto simp*: *filter-empty-conv dest*: *set-dropWhileD*)

abbreviation *MMax* $M \equiv$ *Max* (*set-mset* M)

the $\{\#0::'a\#\}$ is there to ensures that the set is not empty.

definition *get-maximum-level* :: $('a, 'b)$ *ann-lits* $\Rightarrow$ $'a$ *literal multiset* $\Rightarrow$ *nat*
where
get-maximum-level M D = *MMax* $(\{\#0\#\}$ + *image-mset* (*get-level* M) D)

lemma *get-maximum-level-ge-get-level*:
$L \in\#\ D \Longrightarrow$ *get-maximum-level* M $D \geq$ *get-level* M L
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-empty*[*simp*]:
get-maximum-level M $\{\#\}$ = *0*
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-exists-lit-of-max-level*:
$D \neq \{\#\} \Longrightarrow \exists L \in\#\ D$. *get-level* M L = *get-maximum-level* M D
unfolding *get-maximum-level-def*
apply (*induct* D)
apply *simp*
by (*rename-tac* D x, *case-tac* $D = \{\#\}$) (*auto simp add*: *max-def*)

lemma *get-maximum-level-empty-list*[*simp*]:
get-maximum-level [] D = *0*
unfolding *get-maximum-level-def* **by** (*simp add*: *image-constant-conv*)

lemma *get-maximum-level-single*[*simp*]:
get-maximum-level M {#*L*#} = *get-level M L*
unfolding *get-maximum-level-def* **by** *simp*

lemma *get-maximum-level-plus*:
get-maximum-level M (*D* + *D'*) = *max* (*get-maximum-level M D*) (*get-maximum-level M D'*)
by (*induct D*) (*auto simp add*: *get-maximum-level-def*)

lemma *get-maximum-level-exists-lit*:
assumes *n*: *n* > *0*
and *max*: *get-maximum-level M D* = *n*
shows ∃ *L* ∈#*D*. *get-level M L* = *n*
proof −
have *f*: *finite* (*insert 0* ((λ*L*. *get-level M L*) ' *set-mset D*)) **by** *auto*
then have *n* ∈ ((λ*L*. *get-level M L*) ' *set-mset D*)
using *n max Max-in*[*OF f*] **unfolding** *get-maximum-level-def* **by** *simp*
then show ∃ *L* ∈# *D*. *get-level M L* = *n* **by** *auto*
qed

lemma *get-maximum-level-skip-first*[*simp*]:
assumes *atm-of L* ∉ *atms-of D*
shows *get-maximum-level* (*Propagated L C* # *M*) *D* = *get-maximum-level M D*
using *assms* **unfolding** *get-maximum-level-def atms-of-def*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
by (*smt atm-of-in-atm-of-set-in-uminus get-level-skip-beginning image-iff ann-lit.sel*(*2*)
multiset.map-cong0)

lemma *get-maximum-level-skip-beginning*:
assumes *DH*: ∀ *x* ∈ *atms-of D*. *x* ∉ *atm-of* ' *lits-of-l c*
shows *get-maximum-level* (*c* @ *H*) *D* = *get-maximum-level H D*
proof −
have (*get-level* (*c* @ *H*)) ' *set-mset D* = (*get-level H*) ' *set-mset D*
apply (*rule image-cong*)
apply *simp*
using *DH* **unfolding** *atms-of-def* **by** *auto*
then show *?thesis* **using** *DH* **unfolding** *get-maximum-level-def* **by** *auto*
qed

lemma *get-maximum-level-D-single-propagated*:
get-maximum-level [*Propagated x21 x22*] *D* = *0*
unfolding *get-maximum-level-def* **by** (*simp add*: *image-constant-conv*)

lemma *get-maximum-level-skip-un-decided-not-present*:
assumes
∀ *L*∈#*D*. *atm-of L* ∉ *atm-of* ' *lits-of-l M* **and**
∀ *m*∈*set M*. ¬ *is-decided m*
shows *get-maximum-level* (*M* @ *aa*) *D* = *get-maximum-level aa D*
using *assms* **unfolding** *get-maximum-level-def* **by** *simp*

lemma *get-maximum-level-union-mset*:
get-maximum-level M (*A* #∪ *B*) = *get-maximum-level M* (*A* + *B*)
unfolding *get-maximum-level-def* **by** (*auto simp*: *image-Un*)

lemma *count-decided-rev*[*simp*]:
count-decided (*rev M*) = *count-decided M*

by (*auto simp*: *rev-filter*[*symmetric*])

lemma *count-decided-ge-get-level*[*simp*]:
count-decided M $\geq$ *get-level M L*
by (*induct M rule*: *ann-lit-list-induct*) (*auto simp add*: *le-max-iff-disj*)

lemma *count-decided-ge-get-maximum-level*:
count-decided M $\geq$ *get-maximum-level M D*
using *get-maximum-level-exists-lit-of-max-level* **unfolding** *Bex-def*
by (*metis get-maximum-level-empty count-decided-ge-get-level le0*)

fun *get-all-mark-of-propagated* **where**
get-all-mark-of-propagated [] = [] |
get-all-mark-of-propagated (*Decided* - # *L*) = *get-all-mark-of-propagated L* |
get-all-mark-of-propagated (*Propagated* - *mark* # *L*) = *mark* # *get-all-mark-of-propagated L*

lemma *get-all-mark-of-propagated-append*[*simp*]:
get-all-mark-of-propagated (*A* @ *B*) = *get-all-mark-of-propagated A* @ *get-all-mark-of-propagated B*
by (*induct A rule*: *ann-lit-list-induct*) *auto*

Properties about the levels

lemma *atm-lit-of-set-lits-of-l*:
($\lambda l.$ *atm-of* (*lit-of l*)) ' *set xs* = *atm-of* ' *lits-of-l xs*
unfolding *lits-of-def* **by** *auto*

lemma *le-count-decided-decomp*:
assumes *no-dup M*
shows*i* < *count-decided M* $\longleftrightarrow$ ($\exists$ *c K c'*. *M* = *c* @ *Decided K* # *c'* $\wedge$ *get-level M K* = *Suc i*)
(**is** *?A* $\longleftrightarrow$ *?B*)
proof
assume *?B*
then obtain *c K c'* **where**
M = *c* @ *Decided K* # *c'* **and** *get-level M K* = *Suc i*
by *blast*
then show *?A* **using** *count-decided-ge-get-level*[*of K M*] **by** *auto*
next
assume *?A*
then show *?B*
using ‹*no-dup M*›
proof (*induction M rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Decided L M*) **note** *IH* = *this*(*1*) **and** *i* = *this*(*2*) **and** *n-d* = *this*(*3*)
then have *n-d-M*: *no-dup M* **by** *simp*
show *?case*
proof (*cases i* < *count-decided M*)
case *True*
then obtain *c K c'* **where**
M: *M* = *c* @ *Decided K* # *c'* **and** *lev-K*: *get-level M K* = *Suc i*
using *IH n-d-M* **by** *blast*
show *?thesis*
apply (*rule exI*[*of* - *Decided L* # *c*])
apply (*rule exI*[*of* - *K*])
apply (*rule exI*[*of* - *c'*])

using *lev-K n-d* **unfolding** *M* **by** *auto*
next
case *False*
show *?thesis*
apply (*rule exI*[*of* - []])
apply (*rule exI*[*of* - *L*])
apply (*rule exI*[*of* - *M*])
using *False i* **by** *auto*
qed
next
case (*Propagated L mark′ M*) **note** *i* = *this*(*2*) **and** *n-d* = *this*(*3*) **and** *IH* = *this*(*1*)
then obtain *c K c′* **where**
M: *M* = *c* @ *Decided K* # *c′* **and** *lev-K*: *get-level M K* = *Suc i*
by *auto*
show *?case*
apply (*rule exI*[*of* - *Propagated L mark′* # *c*])
apply (*rule exI*[*of* - *K*])
apply (*rule exI*[*of* - *c′*])
using *lev-K n-d* **unfolding** *M* **by** (*auto simp*: *atm-lit-of-set-lits-of-l*)
qed
qed

end
theory *CDCL-W*
imports *List-More CDCL-W-Level Wellfounded-More Partial-Annotated-Clausal-Logic*

begin

Chapter 6

Weidenbach's CDCL

The organisation of the development is the following:

- `CDCL_W.thy` contains the specification of the rules: the rules and the strategy are defined, and we proof the correctness of CDCL.
- `CDCL_W_Termination.thy` contains the proof of termination.
- `CDCL_W_Merge.thy` contains a variant of the calculus: some rules of the raw calculus are always applied together (like the rules analysing the conflict and then backtracking). We define an equivalent version of the calculus where these rules are applied together. This is useful for implementations.
- `CDCL_WNOT.thy` proves the inclusion of Weidenbach's version of CDCL in NOT's version. We use here the version defined in `CDCL_W_Merge.thy`. We need this, because NOT's backjump corresponds to multiple applications of three rules in Weidenbach's calculus. We show also the termination of the calculus without strategy.

We have some variants build on the top of Weidenbach's CDCL calculus:

- `CDCL_W_Incremental.thy` adds incrementality on the top of `CDCL_W.thy`. The way we are doing it is not compatible with `CDCL_W_Merge.thy` , because we add conflicts and the `CDCL_W_Merge.thy` cannot analyse conflicts added externally, because the conflict and analyse are merged.
- `CDCL_W_Restart.thy` adds restart. It is built on the top of `CDCL_W_Merge.thy`.

6.1 Weidenbach's CDCL with Multisets

declare *upt.simps(2)[simp del]*

6.1.1 The State

We will abstract the representation of clause and clauses via two locales. We here use multisets, contrary to `CDCL_W_Abstract_State.thy` where we assume only the existence of a conversion to the state.

locale $state_W$*-ops* =

fixes
trail :: $'st \Rightarrow ('v, 'v\ clause)$ *ann-lits* **and**
init-clss :: $'st \Rightarrow 'v$ *clauses* **and**
learned-clss :: $'st \Rightarrow 'v$ *clauses* **and**
backtrack-lvl :: $'st \Rightarrow$ *nat* **and**
conflicting :: $'st \Rightarrow 'v$ *clause option* **and**

cons-trail :: $('v, 'v\ clause)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-learned-cls :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
update-backtrack-lvl :: *nat* $\Rightarrow 'st \Rightarrow 'st$ **and**
update-conflicting :: $'v$ *clause option* $\Rightarrow 'st \Rightarrow 'st$ **and**

init-state :: $'v$ *clauses* $\Rightarrow 'st$
begin
abbreviation *hd-trail* :: $'st \Rightarrow ('v, 'v\ clause)$ *ann-lit* **where**
hd-trail $S \equiv$ *hd* (*trail* S)

definition *clauses* :: $'st \Rightarrow 'v$ *clauses* **where**
clauses S = *init-clss* S + *learned-clss* S

abbreviation *resolve-cls* **where**
resolve-cls $L\ D'\ E \equiv$ *remove1-mset* $(-L)\ D'$ #∪ *remove1-mset* $L\ E$

abbreviation *state* :: $'st \Rightarrow ('v, 'v\ clause)$ *ann-lits* $\times\ 'v$ *clauses* $\times\ 'v$ *clauses*
$\times$ *nat* $\times\ 'v$ *clause option* **where**
state $S \equiv$ (*trail* S, *init-clss* S, *learned-clss* S, *backtrack-lvl* S, *conflicting* S)
end

We are using an abstract state to abstract away the detail of the implementation: we do not need to know how the clauses are represented internally, we just need to know that they can be converted to multisets.

Weidenbach state is a five-tuple composed of:

1. the trail is a list of decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause ($'v$ *Partial-Clausal-Logic.clause*, standing for conflicting clause) and one for the initial and learned clauses ($'v$ *Partial-Clausal-Logic.clause*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to $'v$ *Partial-Clausal-Logic.clause* is enough (needed for function *hd-trail* below).

There are several axioms to state the independance of the different fields of the state: for example, adding a clause to the learned clauses does not change the trail.

locale $state_W$ =

$state_W$*-ops*

— functions about the state:
— getter:
trail init-clss learned-clss backtrack-lvl conflicting
— setter:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— Some specific states:
init-state
for
trail :: $'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lits$ **and**
init-clss :: $'st \Rightarrow 'v\ clauses$ **and**
learned-clss :: $'st \Rightarrow 'v\ clauses$ **and**
backtrack-lvl :: $'st \Rightarrow nat$ **and**
conflicting :: $'st \Rightarrow 'v\ clause\ option$ **and**

cons-trail :: $('v, 'v\ clause)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-learned-cls :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls :: $'v\ clause \Rightarrow 'st \Rightarrow 'st$ **and**
update-backtrack-lvl :: $nat \Rightarrow 'st \Rightarrow 'st$ **and**
update-conflicting :: $'v\ clause\ option \Rightarrow 'st \Rightarrow 'st$ **and**

init-state :: $'v\ clauses \Rightarrow 'st$ +
assumes
cons-trail:
$\bigwedge S'.\ state\ st = (M, S') \Longrightarrow$
$state\ (cons\text{-}trail\ L\ st) = (L \# M, S')$ **and**

tl-trail:
$\bigwedge S'.\ state\ st = (M, S') \Longrightarrow state\ (tl\text{-}trail\ st) = (tl\ M, S')$ **and**

remove-cls:
$\bigwedge S'.\ state\ st = (M, N, U, S') \Longrightarrow$
$state\ (remove\text{-}cls\ C\ st) =$
$(M, removeAll\text{-}mset\ C\ N, removeAll\text{-}mset\ C\ U, S')$ **and**

add-learned-cls:
$\bigwedge S'.\ state\ st = (M, N, U, S') \Longrightarrow$
$state\ (add\text{-}learned\text{-}cls\ C\ st) = (M, N, \{\#C\#\} + U, S')$ **and**

update-backtrack-lvl:
$\bigwedge S'.\ state\ st = (M, N, U, k, S') \Longrightarrow$
$state\ (update\text{-}backtrack\text{-}lvl\ k'\ st) = (M, N, U, k', S')$ **and**

update-conflicting:
$state\ st = (M, N, U, k, D) \Longrightarrow$
$state\ (update\text{-}conflicting\ E\ st) = (M, N, U, k, E)$ **and**

init-state:
$state\ (init\text{-}state\ N) = ([], N, \{\#\}, 0, None)$
begin
lemma
trail-cons-trail[*simp*]:

trail (*cons-trail L st*) = *L* # *trail st* **and**
trail-tl-trail[*simp*]: *trail* (*tl-trail st*) = *tl* (*trail st*) **and**
trail-add-learned-cls[*simp*]:
trail (*add-learned-cls C st*) = *trail st* **and**
trail-remove-cls[*simp*]:
trail (*remove-cls C st*) = *trail st* **and**
trail-update-backtrack-lvl[*simp*]: *trail* (*update-backtrack-lvl k st*) = *trail st* **and**
trail-update-conflicting[*simp*]: *trail* (*update-conflicting E st*) = *trail st* **and**

init-clss-cons-trail[*simp*]:
init-clss (*cons-trail M st*) = *init-clss st*
and
init-clss-tl-trail[*simp*]:
init-clss (*tl-trail st*) = *init-clss st* **and**
init-clss-add-learned-cls[*simp*]:
init-clss (*add-learned-cls C st*) = *init-clss st* **and**
init-clss-remove-cls[*simp*]:
init-clss (*remove-cls C st*) = *removeAll-mset C* (*init-clss st*) **and**
init-clss-update-backtrack-lvl[*simp*]:
init-clss (*update-backtrack-lvl k st*) = *init-clss st* **and**
init-clss-update-conflicting[*simp*]:
init-clss (*update-conflicting E st*) = *init-clss st* **and**

learned-clss-cons-trail[*simp*]:
learned-clss (*cons-trail M st*) = *learned-clss st* **and**
learned-clss-tl-trail[*simp*]:
learned-clss (*tl-trail st*) = *learned-clss st* **and**
learned-clss-add-learned-cls[*simp*]:
learned-clss (*add-learned-cls C st*) = {#*C*#} + *learned-clss st* **and**
learned-clss-remove-cls[*simp*]:
learned-clss (*remove-cls C st*) = *removeAll-mset C* (*learned-clss st*) **and**
learned-clss-update-backtrack-lvl[*simp*]:
learned-clss (*update-backtrack-lvl k st*) = *learned-clss st* **and**
learned-clss-update-conflicting[*simp*]:
learned-clss (*update-conflicting E st*) = *learned-clss st* **and**

backtrack-lvl-cons-trail[*simp*]:
backtrack-lvl (*cons-trail M st*) = *backtrack-lvl st* **and**
backtrack-lvl-tl-trail[*simp*]:
backtrack-lvl (*tl-trail st*) = *backtrack-lvl st* **and**
backtrack-lvl-add-learned-cls[*simp*]:
backtrack-lvl (*add-learned-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-remove-cls[*simp*]:
backtrack-lvl (*remove-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-update-backtrack-lvl[*simp*]:
backtrack-lvl (*update-backtrack-lvl k st*) = *k* **and**
backtrack-lvl-update-conflicting[*simp*]:
backtrack-lvl (*update-conflicting E st*) = *backtrack-lvl st* **and**

conflicting-cons-trail[*simp*]:
conflicting (*cons-trail M st*) = *conflicting st* **and**
conflicting-tl-trail[*simp*]:
conflicting (*tl-trail st*) = *conflicting st* **and**
conflicting-add-learned-cls[*simp*]:
conflicting (*add-learned-cls C st*) = *conflicting st*
and

conflicting-remove-cls[*simp*]:
conflicting (*remove-cls C st*) = *conflicting st* **and**
conflicting-update-backtrack-lvl[*simp*]:
conflicting (*update-backtrack-lvl k st*) = *conflicting st* **and**
conflicting-update-conflicting[*simp*]:
conflicting (*update-conflicting E st*) = *E* **and**

init-state-trail[*simp*]: *trail* (*init-state N*) = [] **and**
init-state-clss[*simp*]: *init-clss* (*init-state N*) = *N* **and**
init-state-learned-clss[*simp*]: *learned-clss* (*init-state N*) = {#} **and**
init-state-backtrack-lvl[*simp*]: *backtrack-lvl* (*init-state N*) = *0* **and**
init-state-conflicting[*simp*]: *conflicting* (*init-state N*) = *None*

using *cons-trail*[*of st*] *tl-trail*[*of st*] *add-learned-cls*[*of st - - - - C*]
update-backtrack-lvl[*of st - - - - - k*] *update-conflicting*[*of st - - - - - E*]
remove-cls[*of st - - - - C*]
init-state[*of N*]
by (*cases state st*; *auto simp*:)+

lemma
shows
clauses-cons-trail[*simp*]:
clauses (*cons-trail M S*) = *clauses S* **and**

clss-tl-trail[*simp*]: *clauses* (*tl-trail S*) = *clauses S* **and**
clauses-add-learned-cls-unfolded:
clauses (*add-learned-cls U S*) = {#*U*#} + *learned-clss S* + *init-clss S*
and
clauses-update-backtrack-lvl[*simp*]: *clauses* (*update-backtrack-lvl k S*) = *clauses S* **and**
clauses-update-conflicting[*simp*]: *clauses* (*update-conflicting D S*) = *clauses S* **and**
clauses-remove-cls[*simp*]:
clauses (*remove-cls C S*) = *removeAll-mset C* (*clauses S*) **and**
clauses-add-learned-cls[*simp*]:
clauses (*add-learned-cls C S*) = {#*C*#} + *clauses S* **and**
clauses-init-state[*simp*]: *clauses* (*init-state N*) = *N*
by (*auto simp*: *ac-simps replicate-mset-plus clauses-def intro*: *multiset-eqI*)

abbreviation *incr-lvl* :: *'st* ⇒ *'st* **where**
incr-lvl S ≡ *update-backtrack-lvl* (*backtrack-lvl S* + *1*) *S*

definition *state-eq* :: *'st* ⇒ *'st* ⇒ *bool* (**infix** ∼ *50*) **where**
S ∼ *T* ⟷ *state S* = *state T*

lemma *state-eq-ref*[*simp*, *intro*]:
S ∼ *S*
unfolding *state-eq-def* **by** *auto*

lemma *state-eq-sym*:
S ∼ *T* ⟷ *T* ∼ *S*
unfolding *state-eq-def* **by** *auto*

lemma *state-eq-trans*:
S ∼ *T* ⟹ *T* ∼ *U* ⟹ *S* ∼ *U*
unfolding *state-eq-def* **by** *auto*

lemma

shows
state-eq-trail: $S \sim T \Longrightarrow$ *trail* $S =$ *trail* T **and**
state-eq-init-clss: $S \sim T \Longrightarrow$ *init-clss* $S =$ *init-clss* T **and**
state-eq-learned-clss: $S \sim T \Longrightarrow$ *learned-clss* $S =$ *learned-clss* T **and**
state-eq-backtrack-lvl: $S \sim T \Longrightarrow$ *backtrack-lvl* $S =$ *backtrack-lvl* T **and**
state-eq-conflicting: $S \sim T \Longrightarrow$ *conflicting* $S =$ *conflicting* T **and**
state-eq-clauses: $S \sim T \Longrightarrow$ *clauses* $S =$ *clauses* T **and**
state-eq-undefined-lit: $S \sim T \Longrightarrow$ *undefined-lit* (*trail* S) $L =$ *undefined-lit* (*trail* T) L
unfolding *state-eq-def clauses-def* **by** *auto*

lemma *state-eq-conflicting-None*:
$S \sim T \Longrightarrow$ *conflicting* $T =$ *None* $\Longrightarrow$ *conflicting* $S =$ *None*
unfolding *state-eq-def clauses-def* **by** *auto*

We combine all simplification rules about *op* $\sim$ in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

lemmas *state-simp*[*simp*] = *state-eq-trail state-eq-init-clss state-eq-learned-clss*
state-eq-backtrack-lvl state-eq-conflicting state-eq-clauses state-eq-undefined-lit
state-eq-conflicting-None

function *reduce-trail-to* :: *'a list* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **where**
reduce-trail-to F S =
(*if length* (*trail* S) = *length* F $\vee$ *trail* S = [] *then* S *else reduce-trail-to* F (*tl-trail* S))
by *fast+*
termination
by (*relation measure* (λ(-, S). *length* (*trail* S))) *simp-all*

declare *reduce-trail-to.simps*[*simp del*]

lemma
shows
reduce-trail-to-Nil[*simp*]: *trail* S = [] $\Longrightarrow$ *reduce-trail-to* F S = S **and**
reduce-trail-to-eq-length[*simp*]: *length* (*trail* S) = *length* F $\Longrightarrow$ *reduce-trail-to* F S = S
by (*auto simp*: *reduce-trail-to.simps*)

lemma *reduce-trail-to-length-ne*:
length (*trail* S) $\neq$ *length* F $\Longrightarrow$ *trail* S $\neq$ [] $\Longrightarrow$
reduce-trail-to F S = *reduce-trail-to* F (*tl-trail* S)
by (*auto simp*: *reduce-trail-to.simps*)

lemma *trail-reduce-trail-to-length-le*:
assumes *length* F > *length* (*trail* S)
shows *trail* (*reduce-trail-to* F S) = []
using *assms* **apply** (*induction* F S *rule*: *reduce-trail-to.induct*)
by (*metis* (*no-types*, *hide-lams*) *length-tl less-imp-diff-less less-irrefl trail-tl-trail*
reduce-trail-to.simps)

lemma *trail-reduce-trail-to-Nil*[*simp*]:
trail (*reduce-trail-to* [] S) = []
apply (*induction* []::(*'v*, *'v clause*) *ann-lits* S *rule*: *reduce-trail-to.induct*)
by (*metis length-0-conv reduce-trail-to-length-ne reduce-trail-to-Nil*)

lemma *clauses-reduce-trail-to-Nil*:
clauses (*reduce-trail-to* [] S) = *clauses* S

proof (*induction* [] *S rule*: *reduce-trail-to.induct*)
  case (*1 Sa*)
  then have *clauses* (*reduce-trail-to* ([]::'*a list*) (*tl-trail Sa*)) = *clauses* (*tl-trail Sa*)
    ∨ *trail Sa* = []
    by *fastforce*
  then show *clauses* (*reduce-trail-to* ([]::'*a list*) *Sa*) = *clauses Sa*
    by (*metis* (*no-types*) *length-0-conv reduce-trail-to-eq-length clss-tl-trail*
      reduce-trail-to-length-ne)
qed

lemma *reduce-trail-to-skip-beginning*:
  assumes *trail* $S = F'$ @ F
  shows *trail* (*reduce-trail-to F S*) = *F*
  using *assms* **by** (*induction* F' *arbitrary*: *S*) (*auto simp*: *reduce-trail-to-length-ne*)

lemma *clauses-reduce-trail-to*[*simp*]:
  clauses (*reduce-trail-to F S*) = *clauses S*
  apply (*induction F S rule*: *reduce-trail-to.induct*)
  by (*metis clss-tl-trail reduce-trail-to.simps*)

lemma *conflicting-update-trail*[*simp*]:
  conflicting (*reduce-trail-to F S*) = *conflicting S*
  apply (*induction F S rule*: *reduce-trail-to.induct*)
  by (*metis conflicting-tl-trail reduce-trail-to.simps*)

lemma *backtrack-lvl-update-trail*[*simp*]:
  backtrack-lvl (*reduce-trail-to F S*) = *backtrack-lvl S*
  apply (*induction F S rule*: *reduce-trail-to.induct*)
  by (*metis backtrack-lvl-tl-trail reduce-trail-to.simps*)

lemma *init-clss-update-trail*[*simp*]:
  init-clss (*reduce-trail-to F S*) = *init-clss S*
  apply (*induction F S rule*: *reduce-trail-to.induct*)
  by (*metis init-clss-tl-trail reduce-trail-to.simps*)

lemma *learned-clss-update-trail*[*simp*]:
  learned-clss (*reduce-trail-to F S*) = *learned-clss S*
  apply (*induction F S rule*: *reduce-trail-to.induct*)
  by (*metis learned-clss-tl-trail reduce-trail-to.simps*)

lemma *conflicting-reduce-trail-to*[*simp*]:
  conflicting (*reduce-trail-to F S*) = *None* ⟷ *conflicting S* = *None*
  apply (*induction F S rule*: *reduce-trail-to.induct*)
  by (*metis conflicting-update-trail map-option-is-None*)

lemma *trail-eq-reduce-trail-to-eq*:
  trail S = *trail T* ⟹ *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
  apply (*induction F S arbitrary*: *T rule*: *reduce-trail-to.induct*)
  by (*metis trail-tl-trail reduce-trail-to.simps*)

lemma *reduce-trail-to-state-eq*$_{NOT}$*-compatible*:
  assumes *ST*: *S* ∼ *T*
  shows *reduce-trail-to F S* ∼ *reduce-trail-to F T*
proof −
  have *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
    using *trail-eq-reduce-trail-to-eq*[*of S T F*] *ST* **by** *auto*

then show *?thesis* **using** *ST* **by** (*auto simp del*: *state-simp simp*: *state-eq-def*)
qed

lemma *reduce-trail-to-trail-tl-trail-decomp*[*simp*]:
trail S = *F'* @ *Decided K* # *F* ⟹ (*trail* (*reduce-trail-to F S*)) = *F*
apply (*rule reduce-trail-to-skip-beginning*[*of - F'* @ *Decided K* # []])
by (*cases F'*) (*auto simp add*:*tl-append reduce-trail-to-skip-beginning*)

lemma *reduce-trail-to-add-learned-cls*[*simp*]:
trail (*reduce-trail-to F* (*add-learned-cls C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-remove-learned-cls*[*simp*]:
trail (*reduce-trail-to F* (*remove-cls C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-update-conflicting*[*simp*]:
trail (*reduce-trail-to F* (*update-conflicting C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-update-backtrack-lvl*[*simp*]:
trail (*reduce-trail-to F* (*update-backtrack-lvl k S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-length*:
length M = *length M'* ⟹ *reduce-trail-to M S* = *reduce-trail-to M' S*
apply (*induction M S rule*: *reduce-trail-to.induct*)
by (*simp add*: *reduce-trail-to.simps*)

lemma *trail-reduce-trail-to-drop*:
trail (*reduce-trail-to F S*) =
(*if length* (*trail S*) ≥ *length F*
then drop (*length* (*trail S*) − *length F*) (*trail S*)
else [])
apply (*induction F S rule*: *reduce-trail-to.induct*)
apply (*rename-tac F S*, *case-tac trail S*)
apply *auto*[]
apply (*rename-tac list*, *case-tac Suc* (*length list*) > *length F*)
prefer *2* **apply** (*metis diff-is-0-eq drop-Cons' length-Cons nat-le-linear nat-less-le reduce-trail-to-eq-length trail-reduce-trail-to-length-le*)
apply (*subgoal-tac Suc* (*length list*) − *length F* = *Suc* (*length list* − *length F*))
by (*auto simp add*: *reduce-trail-to-length-ne*)

lemma *in-get-all-ann-decomposition-trail-update-trail*[*simp*]:
assumes *H*: (*L* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*))
shows *trail* (*reduce-trail-to M1 S*) = *M1*
proof −
obtain *K* **where**
L: *L* = *Decided K*
using *H* **by** (*cases L*) (*auto dest*!: *in-get-all-ann-decomposition-decided-or-empty*)
obtain *c* **where**
tr-S: *trail S* = *c* @ *M2* @ *L* # *M1*
using *H* **by** *auto*
show *?thesis*
by (*rule reduce-trail-to-trail-tl-trail-decomp*[*of - c* @ *M2 K*])
(*auto simp*: *tr-S L*)

qed

lemma *conflicting-cons-trail-conflicting*[*simp*]:
  assumes *undefined-lit* (*trail S*) (*lit-of L*)
  shows
    conflicting (*cons-trail L S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
  using *assms conflicting-cons-trail*[*of L S*] *map-option-is-None* **by** *fastforce+*

lemma *conflicting-add-learned-cls-conflicting*[*simp*]:
  conflicting (*add-learned-cls C S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
  by *fastforce+*

lemma *conflicting-update-backtracl-lvl*[*simp*]:
  conflicting (*update-backtrack-lvl k S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
  using *map-option-is-None conflicting-update-backtrack-lvl*[*of k S*] **by** *fastforce+*

end — end of $state_W$ locale

6.1.2 CDCL Rules

Because of the strategy we will later use, we distinguish propagate, conflict from the other rules

locale $conflict\text{-}driven\text{-}clause\text{-}learning_W$ =
  $state_W$
    — functions for the state:
      — access functions:
    trail init-clss learned-clss backtrack-lvl conflicting
      — changing state:
    cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
    update-conflicting

      — get state:
    init-state
  for
    trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
    init-clss :: *'st* ⇒ *'v clauses* **and**
    learned-clss :: *'st* ⇒ *'v clauses* **and**
    backtrack-lvl :: *'st* ⇒ *nat* **and**
    conflicting :: *'st* ⇒ *'v clause option* **and**

    cons-trail :: (*'v*, *'v clause*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
    tl-trail :: *'st* ⇒ *'st* **and**
    add-learned-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
    remove-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
    update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
    update-conflicting :: *'v clause option* ⇒ *'st* ⇒ *'st* **and**

    init-state :: *'v clauses* ⇒ *'st*
begin

inductive *propagate* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate-rule: *conflicting S* = *None* ⟹
  E ∈# *clauses S* ⟹
  L ∈# *E* ⟹
  trail S ⊨*as CNot* (*E* − {#*L*#}) ⟹
  undefined-lit (*trail S*) *L* ⟹

$T \sim \textit{cons-trail} \ (\textit{Propagated } L \ E) \ S \Longrightarrow$
$\textit{propagate } S \ T$

inductive-cases *propagateE*: *propagate S T*

inductive *conflict* :: $'st \Rightarrow 'st \Rightarrow \textit{bool}$ **for** S :: $'st$ **where**
conflict-rule:
$\textit{conflicting } S = \textit{None} \Longrightarrow$
$D \in\# \textit{ clauses } S \Longrightarrow$
$\textit{trail } S \models\!\textit{as CNot } D \Longrightarrow$
$T \sim \textit{update-conflicting} \ (\textit{Some } D) \ S \Longrightarrow$
$\textit{conflict } S \ T$

inductive-cases *conflictE*: *conflict S T*

inductive *backtrack* :: $'st \Rightarrow 'st \Rightarrow \textit{bool}$ **for** S :: $'st$ **where**
backtrack-rule:
$\textit{conflicting } S = \textit{Some } D \Longrightarrow$
$L \in\# \ D \Longrightarrow$
$(\textit{Decided } K \ \# \ M1, \ M2) \in \textit{set} \ (\textit{get-all-ann-decomposition} \ (\textit{trail } S)) \Longrightarrow$
$\textit{get-level} \ (\textit{trail } S) \ L = \textit{backtrack-lvl } S \Longrightarrow$
$\textit{get-level} \ (\textit{trail } S) \ L = \textit{get-maximum-level} \ (\textit{trail } S) \ D \Longrightarrow$
$\textit{get-maximum-level} \ (\textit{trail } S) \ (D - \{\#L\#\}) \equiv i \Longrightarrow$
$\textit{get-level} \ (\textit{trail } S) \ K = i + 1 \Longrightarrow$
$T \sim \textit{cons-trail} \ (\textit{Propagated } L \ D)$
$(\textit{reduce-trail-to } M1$
$(\textit{add-learned-cls } D$
$(\textit{update-backtrack-lvl } i$
$(\textit{update-conflicting None } S)))) \Longrightarrow$
$\textit{backtrack } S \ T$

inductive-cases *backtrackE*: *backtrack S T*
thm *backtrackE*

inductive *decide* :: $'st \Rightarrow 'st \Rightarrow \textit{bool}$ **for** S :: $'st$ **where**
decide-rule:
$\textit{conflicting } S = \textit{None} \Longrightarrow$
$\textit{undefined-lit} \ (\textit{trail } S) \ L \Longrightarrow$
$\textit{atm-of } L \in \textit{atms-of-mm} \ (\textit{init-clss } S) \Longrightarrow$
$T \sim \textit{cons-trail} \ (\textit{Decided } L) \ (\textit{incr-lvl } S) \Longrightarrow$
$\textit{decide } S \ T$

inductive-cases *decideE*: *decide S T*

inductive *skip* :: $'st \Rightarrow 'st \Rightarrow \textit{bool}$ **for** S :: $'st$ **where**
skip-rule:
$\textit{trail } S = \textit{Propagated } L \ C' \ \# \ M \Longrightarrow$
$\textit{conflicting } S = \textit{Some } E \Longrightarrow$
$-L \notin\# \ E \Longrightarrow$
$E \neq \{\#\} \Longrightarrow$
$T \sim \textit{tl-trail } S \Longrightarrow$
$\textit{skip } S \ T$

inductive-cases *skipE*: *skip S T*

$\textit{get-maximum-level} \ (\textit{Propagated } L \ (C + \{\#L\#\}) \ \# \ M) \ D = k \lor k = 0$ (that was in a previous version of the book) is equivalent to *get-maximum-level* (*Propagated* L (C + {#L#}) # M) D = k, when the structural invariants holds.

inductive *resolve* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
resolve-rule: *trail* $S \neq []$ ⟹
  hd-trail S = *Propagated* L E ⟹
  L ∈# E ⟹
  conflicting S = *Some* D' ⟹
  $-L$ ∈# D' ⟹
  get-maximum-level (*trail* S) ((*remove1-mset* ($-L$) D')) = *backtrack-lvl* S ⟹
  T ∼ *update-conflicting* (*Some* (*resolve-cls* L D' E))
    (*tl-trail* S) ⟹
  resolve S T

inductive-cases *resolveE*: *resolve* S T

inductive *restart* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
restart: *state* S = (M, N, U, k, *None*) ⟹
  ¬M ⊨*asm clauses* S ⟹
  U' ⊆# U ⟹
  state T = ([], N, U', *0*, *None*) ⟹
  restart S T

inductive-cases *restartE*: *restart* S T

We add the condition C ∉# *init-clss* S, to maintain consistency even without the strategy.

inductive *forget* :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget-rule:
  conflicting S = *None* ⟹
  C ∈# *learned-clss* S ⟹
  ¬(*trail* S) ⊨*asm clauses* S ⟹
  C ∉ *set* (*get-all-mark-of-propagated* (*trail* S)) ⟹
  C ∉# *init-clss* S ⟹
  T ∼ *remove-cls* C S ⟹
  forget S T

inductive-cases *forgetE*: *forget* S T

inductive $cdcl_W$*-rf* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
restart: *restart* S T ⟹ $cdcl_W$*-rf* S T |
forget: *forget* S T ⟹ $cdcl_W$*-rf* S T

inductive $cdcl_W$*-bj* :: *'st* ⇒ *'st* ⇒ *bool* **where**
skip: *skip* S S' ⟹ $cdcl_W$*-bj* S S' |
resolve: *resolve* S S' ⟹ $cdcl_W$*-bj* S S' |
backtrack: *backtrack* S S' ⟹ $cdcl_W$*-bj* S S'

inductive-cases $cdcl_W$*-bjE*: $cdcl_W$*-bj* S T

inductive $cdcl_W$*-o* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
decide: *decide* S S' ⟹ $cdcl_W$*-o* S S' |
bj: $cdcl_W$*-bj* S S' ⟹ $cdcl_W$*-o* S S'

inductive $cdcl_W$ :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
propagate: *propagate* S S' ⟹ $cdcl_W$ S S' |
conflict: *conflict* S S' ⟹ $cdcl_W$ S S' |

other: $cdcl_W$-*o* S S' $\Longrightarrow$ $cdcl_W$ S S'|
rf: $cdcl_W$-*rf* S S' $\Longrightarrow$ $cdcl_W$ S S'

lemma *rtranclp-propagate-is-rtranclp-*$cdcl_W$:
  propagate** S S' $\Longrightarrow$ $cdcl_W{}^{**}$ S S'
  apply (*induction rule*: *rtranclp-induct*)
    apply *simp*
  apply (*frule propagate*)
  using *rtranclp-trans*[*of* $cdcl_W$] **by** *blast*

lemma $cdcl_W$-*all-rules-induct*[*consumes 1*, *case-names propagate conflict forget restart decide skip resolve backtrack*]:
  fixes S :: *'st*
  assumes
    $cdcl_W$: $cdcl_W$ S S' **and**
    propagate: $\bigwedge T$. *propagate* S T $\Longrightarrow$ P S T **and**
    conflict: $\bigwedge T$. *conflict* S T $\Longrightarrow$ P S T **and**
    forget: $\bigwedge T$. *forget* S T $\Longrightarrow$ P S T **and**
    restart: $\bigwedge T$. *restart* S T $\Longrightarrow$ P S T **and**
    decide: $\bigwedge T$. *decide* S T $\Longrightarrow$ P S T **and**
    skip: $\bigwedge T$. *skip* S T $\Longrightarrow$ P S T **and**
    resolve: $\bigwedge T$. *resolve* S T $\Longrightarrow$ P S T **and**
    backtrack: $\bigwedge T$. *backtrack* S T $\Longrightarrow$ P S T
  shows P S S'
  using *assms*(*1*)
proof (*induct* S' *rule*: $cdcl_W$.*induct*)
  case (*propagate* S') **note** *propagate* = *this*(*1*)
  then show *?case* **using** *assms*(*2*) **by** *auto*
next
  case (*conflict* S')
  then show *?case* **using** *assms*(*3*) **by** *auto*
next
  case (*other* S')
  then show *?case*
    proof (*induct rule*: $cdcl_W$-*o.induct*)
      case (*decide* U)
      then show *?case* **using** *assms*(*6*) **by** *auto*
    next
      case (*bj* S')
      then show *?case* **using** *assms*(*7*−*9*) **by** (*induction rule*: $cdcl_W$-*bj.induct*) *auto*
    qed
next
  case (*rf* S')
  then show *?case*
    by (*induct rule*: $cdcl_W$-*rf.induct*) (*fast dest*: *forget restart*)+
qed

lemma $cdcl_W$-*all-induct*[*consumes 1*, *case-names propagate conflict forget restart decide skip resolve backtrack*]:
  fixes S :: *'st*
  assumes
    $cdcl_W$: $cdcl_W$ S S' **and**
    propagateH: $\bigwedge C$ L T. *conflicting* S = *None* $\Longrightarrow$
      C $\in\#$ *clauses* S $\Longrightarrow$
      L $\in\#$ C $\Longrightarrow$
      trail S $\models$*as CNot* (*remove1-mset* L C) $\Longrightarrow$

undefined-lit (*trail* S) $L \Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* L C) $S \Longrightarrow$
$P\ S\ T$ **and**
conflictH: $\bigwedge D\ T$. *conflicting* S = *None* $\Longrightarrow$
$D \in\#$ *clauses* $S \Longrightarrow$
trail $S \models$*as CNot* $D \Longrightarrow$
$T \sim$ *update-conflicting* (*Some* D) $S \Longrightarrow$
$P\ S\ T$ **and**
forgetH: $\bigwedge C\ T$. *conflicting* S = *None* $\Longrightarrow$
$C \in\#$ *learned-clss* $S \Longrightarrow$
$\neg$(*trail* S) $\models$*asm clauses* $S \Longrightarrow$
$C \notin$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\Longrightarrow$
$C \notin\#$ *init-clss* $S \Longrightarrow$
$T \sim$ *remove-cls* C $S \Longrightarrow$
$P\ S\ T$ **and**
restartH: $\bigwedge T\ U$. $\neg$*trail* $S \models$*asm clauses* $S \Longrightarrow$
conflicting S = *None* $\Longrightarrow$
state T = ([], *init-clss* S, U, *0*, *None*) $\Longrightarrow$
$U \subseteq\#$ *learned-clss* $S \Longrightarrow$
$P\ S\ T$ **and**
decideH: $\bigwedge L\ T$. *conflicting* S = *None* $\Longrightarrow$
undefined-lit (*trail* S) $L \Longrightarrow$
atm-of $L \in$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
$T \sim$ *cons-trail* (*Decided* L) (*incr-lvl* S) $\Longrightarrow$
$P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T$.
trail S = *Propagated* L C' # $M \Longrightarrow$
conflicting S = *Some* $E \Longrightarrow$
$-L \notin\# E \Longrightarrow E \neq \{\#\} \Longrightarrow$
$T \sim$ *tl-trail* $S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T$.
trail S = *Propagated* L E # $M \Longrightarrow$
$L \in\# E \Longrightarrow$
hd-trail S = *Propagated* L $E \Longrightarrow$
conflicting S = *Some* $D \Longrightarrow$
$-L \in\# D \Longrightarrow$
get-maximum-level (*trail* S) ((*remove1-mset* ($-L$) D)) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*resolve-cls* L D E)) (*tl-trail* S) $\Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge L\ D\ K\ i\ M1\ M2\ T$.
conflicting S = *Some* $D \Longrightarrow$
$L \in\# D \Longrightarrow$
(*Decided* K # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) $D \Longrightarrow$
get-maximum-level (*trail* S) (*remove1-mset* L D) $\equiv i \Longrightarrow$
get-level (*trail* S) K = i+*1* $\Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* L D)
(*reduce-trail-to* *M1*
(*add-learned-cls* D
(*update-backtrack-lvl* i
(*update-conflicting* *None* S)))) $\Longrightarrow$
$P\ S\ T$
shows $P\ S\ S'$

using $cdcl_W$
proof (*induct S S' rule*: $cdcl_W$*-all-rules-induct*)
case (*propagate S'*)
then show *?case*
by (*auto elim*!: *propagateE intro*!: *propagateH*)
next
case (*conflict S'*)
then show *?case*
by (*auto elim*!: *conflictE intro*!: *conflictH*)
next
case (*restart S'*)
then show *?case*
by (*auto elim*!: *restartE intro*!: *restartH*)
next
case (*decide T*)
then show *?case*
by (*auto elim*!: *decideE intro*!: *decideH*)
next
case (*backtrack S'*)
then show *?case* **by** (*auto elim*!: *backtrackE intro*!: *backtrackH*
simp del: *state-simp simp add*: *state-eq-def*)
next
case (*forget S'*)
then show *?case* **by** (*auto elim*!: *forgetE intro*!: *forgetH*)
next
case (*skip S'*)
then show *?case* **by** (*auto elim*!: *skipE intro*!: *skipH*)
next
case (*resolve S'*)
then show *?case*
by (*cases trail S*) (*auto elim*!: *resolveE intro*!: *resolveH*)
qed

lemma $cdcl_W$*-o-induct*[*consumes 1*, *case-names decide skip resolve backtrack*]:
fixes S :: *'st*
assumes $cdcl_W$: $cdcl_W$*-o* S T **and**
decideH: $\bigwedge L\ T.$ *conflicting* S = *None* $\Longrightarrow$ *undefined-lit* (*trail* S) L
$\Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* (*init-clss* S)
$\Longrightarrow$ $T \sim$ *cons-trail* (*Decided* L) (*incr-lvl* S)
$\Longrightarrow$ $P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
trail S = *Propagated* $L\ C'$ # M $\Longrightarrow$
conflicting S = *Some* E $\Longrightarrow$
$-L \notin\# E \Longrightarrow E \neq \{\#\} \Longrightarrow$
$T \sim$ *tl-trail* S $\Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
trail S = *Propagated* $L\ E$ # M $\Longrightarrow$
$L \in\# E \Longrightarrow$
hd-trail S = *Propagated* $L\ E$ $\Longrightarrow$
conflicting S = *Some* D $\Longrightarrow$
$-L \in\# D \Longrightarrow$
get-maximum-level (*trail* S) ((*remove1-mset* $(-L)$ D)) = *backtrack-lvl* S $\Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*resolve-cls* $L\ D\ E$)) (*tl-trail* S) $\Longrightarrow$
$P\ S\ T$ **and**

backtrackH: $\bigwedge L\ D\ K\ i\ M1\ M2\ T$.
conflicting S = *Some* $D \Longrightarrow$
$L \in\#\ D \Longrightarrow$
(*Decided* K # $M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) $D \Longrightarrow$
get-maximum-level (*trail* S) (*remove1-mset* L D) $\equiv i \Longrightarrow$
get-level (*trail* S) $K = i + 1 \Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* L D)
(*reduce-trail-to* $M1$
(*add-learned-cls* D
(*update-backtrack-lvl* i
(*update-conflicting* *None* S)))) $\Longrightarrow$
$P\ S\ T$
shows $P\ S\ T$
using $cdcl_W$ **apply** (*induct* T *rule*: $cdcl_W$*-o.induct*)
using *assms*(*2*) **apply** (*auto elim*: *decideE*)[*1*]
apply (*elim* $cdcl_W$*-bjE skipE resolveE backtrackE*)
apply (*frule skipH*; *simp*)
apply (*cases trail* S; *auto elim*!: *resolveE intro*!: *resolveH*)
apply (*frule backtrackH*; *simp*)
done

thm $cdcl_W$*-o.induct*
lemma $cdcl_W$*-o-all-rules-induct*[*consumes 1*, *case-names decide backtrack skip resolve*]:
fixes $S\ T$ :: *'st*
assumes
$cdcl_W$*-o* $S\ T$ **and**
$\bigwedge T$. *decide* $S\ T \Longrightarrow P\ S\ T$ **and**
$\bigwedge T$. *backtrack* $S\ T \Longrightarrow P\ S\ T$ **and**
$\bigwedge T$. *skip* $S\ T \Longrightarrow P\ S\ T$ **and**
$\bigwedge T$. *resolve* $S\ T \Longrightarrow P\ S\ T$
shows $P\ S\ T$
using *assms* **by** (*induct* T *rule*: $cdcl_W$*-o.induct*) (*auto simp*: $cdcl_W$*-bj.simps*)

lemma $cdcl_W$*-o-rule-cases*[*consumes 1*, *case-names decide backtrack skip resolve*]:
fixes $S\ T$ :: *'st*
assumes
$cdcl_W$*-o* $S\ T$ **and**
decide $S\ T \Longrightarrow P$ **and**
backtrack $S\ T \Longrightarrow P$ **and**
skip $S\ T \Longrightarrow P$ **and**
resolve $S\ T \Longrightarrow P$
shows P
using *assms* **by** (*auto simp*: $cdcl_W$*-o.simps* $cdcl_W$*-bj.simps*)

6.1.3 Structural Invariants

Properties of the trail

We here establish that:

- the consistency of the trail;
- the fact that there is no duplicate in the trail.

lemma *backtrack-lit-skiped*:
  assumes
    L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
    M1: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
    no-dup: *no-dup* (*trail S*) **and**
    bt-l: *backtrack-lvl S* = *length* (*filter is-decided* (*trail S*)) **and**
    lev-K: *get-level* (*trail S*) *K* = *i* + *1*
  shows *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
proof (*rule ccontr*)
  let *?M* = *trail S*
  assume *L-in-M1*: ¬*atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
  obtain *c* **where**
    Mc: *trail S* = *c* @ *M2* @ *Decided K* # *M1*
    using *M1* **by** *blast*
  have *atm-of L* ∉ *atm-of* ‘ *lits-of-l c* **and** *atm-of L* ∉ *atm-of* ‘ *lits-of-l M2* **and**
    atm-of L ≠ *atm-of K* **and** *Kc*: *atm-of K* ∉ *atm-of* ‘ *lits-of-l c* **and**
    KM2: *atm-of K* ∉ *atm-of* ‘ *lits-of-l M2*
    using *L-in-M1 no-dup* **unfolding** *Mc lits-of-def* **by** *force+*
  then have *g-M-eq-g-M1*: *get-level ?M L* = *get-level M1 L*
    using *L-in-M1* **unfolding** *Mc* **by** *auto*
  then have *get-level M1 L* < *Suc i*
    using *count-decided-ge-get-level*[*of L M1*] *KM2 lev-K Kc* **unfolding** *Mc*
    by (*auto simp del*: *count-decided-ge-get-level*)
  moreover have *Suc i* ≤ *backtrack-lvl S* **using** *bt-l KM2 lev-K Kc* **unfolding** *Mc* **by** (*simp add*: *Mc*)
  ultimately show *False* **using** *L g-M-eq-g-M1* **by** *auto*
qed

lemma $cdcl_W$*-distinctinv-1*:
  assumes
    $cdcl_W$ *S S′* **and**
    no-dup (*trail S*) **and**
    bt-lev: *backtrack-lvl S* = *count-decided* (*trail S*)
  shows *no-dup* (*trail S′*)
  using *assms*
proof (*induct rule*: $cdcl_W$*-all-induct*)
  case (*backtrack L D K i M1 M2 T*) **note** *decomp* = *this*(*3*) **and** *L* = *this*(*4*) **and** *lev-K* = *this*(*7*)
and
    T = *this*(*8*) **and** *n-d* = *this*(*9*)
  obtain *c* **where** *Mc*: *trail S* = *c* @ *M2* @ *Decided K* # *M1*
    using *decomp* **by** *auto*
  have *no-dup* (*M2* @ *Decided K* # *M1*)
    using *Mc n-d* **by** *fastforce*
  moreover have *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
    using *backtrack-lit-skiped*[*of L S K M1 M2 i*] *L decomp lev-K n-d bt-lev* **by** *fast*
  moreover then have *undefined-lit M1 L*
     by (*simp add*: *defined-lit-map lits-of-def image-image*)
  ultimately show *?case* **using** *decomp T n-d* **by** (*simp add*: *lits-of-def image-image*)
qed (*auto simp*: *defined-lit-map*)

Item 1 page 81 of Weidenbach's book

lemma $cdcl_W$*-consistent-inv-2*:
  assumes
    $cdcl_W$ *S S′* **and**
    no-dup (*trail S*) **and**
    backtrack-lvl S = *count-decided* (*trail S*)
  shows *consistent-interp* (*lits-of-l* (*trail S′*))

using $cdcl_W$*-distinctinv-1*[*OF assms*] *distinct-consistent-interp* **by** *fast*

lemma $cdcl_W$*-o-bt*:
assumes
$cdcl_W$*-o S S'* **and**
backtrack-lvl S = *count-decided* (*trail S*) **and**
n-d[*simp*]: *no-dup* (*trail S*)
shows *backtrack-lvl S'* = *count-decided* (*trail S'*)
using *assms*
proof (*induct rule*: $cdcl_W$*-o-induct*)
case (*backtrack L D K i M1 M2 T*) **note** *decomp* = *this*(*3*) **and** *levK* = *this*(*7*) **and** *T* = *this*(*8*) **and**
level = *this*(*9*)
have [*simp*]: *trail* (*reduce-trail-to M1 S*) = *M1*
using *decomp* **by** *auto*
obtain *c* **where** *M*: *trail S* = *c* @ *M2* @ *Decided K* # *M1* **using** *decomp* **by** *auto*
moreover have *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
using *backtrack-lit-skiped*[*of L S K M1 M2 i*] *backtrack*(*4*,*8*,*9*) *levK decomp*
by (*fastforce simp add*: *lits-of-def*)
moreover then have *undefined-lit M1 L*
by (*simp add*: *defined-lit-map lits-of-def image-image*)
moreover
have *atm-of K* ∉ *atm-of* ‘ *lits-of-l M1* **and** *atm-of K* ∉ *atm-of* ‘ *lits-of-l c*
and *atm-of K* ∉ *atm-of* ‘ *lits-of-l M2*
using *T n-d levK* **unfolding** *M* **by** (*auto simp*: *lits-of-def*)
ultimately show *?case*
using *T levK* **unfolding** *M* **by** (*auto dest*!: *append-cons-eq-upt-length*)
qed *auto*

lemma $cdcl_W$*-rf-bt*:
assumes
$cdcl_W$*-rf S S'* **and**
backtrack-lvl S = *count-decided* (*trail S*)
shows *backtrack-lvl S'* = *count-decided* (*trail S'*)
using *assms* **by** (*induct rule*: $cdcl_W$*-rf.induct*) (*auto elim*: *restartE forgetE*)

Item 7 page 81 of Weidenbach's book

lemma $cdcl_W$*-bt*:
assumes
$cdcl_W$ *S S'* **and**
backtrack-lvl S = *count-decided* (*trail S*) **and**
no-dup (*trail S*)
shows *backtrack-lvl S'* = *count-decided* (*trail S'*)
using *assms* **by** (*induct rule*: $cdcl_W$*.induct*) (*auto simp*: $cdcl_W$*-o-bt* $cdcl_W$*-rf-bt*
elim: *conflictE propagateE*)

We write *1* + *count-decided* (*trail S*) instead of *backtrack-lvl S* to avoid non termination of rewriting.

definition $cdcl_W$*-M-level-inv* :: *'st* ⇒ *bool* **where**
$cdcl_W$*-M-level-inv S* ⟷
consistent-interp (*lits-of-l* (*trail S*))
∧ *no-dup* (*trail S*)
∧ *backtrack-lvl S* = *count-decided* (*trail S*)

lemma $cdcl_W$*-M-level-inv-decomp*:

assumes *cdcl*$_W$*-M-level-inv S*
shows
consistent-interp (*lits-of-l* (*trail S*)) **and**
no-dup (*trail S*)
using *assms* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *fastforce+*

lemma *cdcl*$_W$*-consistent-inv*:
fixes *S S'* :: *'st*
assumes
cdcl$_W$ *S S'* **and**
cdcl$_W$*-M-level-inv S*
shows *cdcl*$_W$*-M-level-inv S'*
using *assms cdcl*$_W$*-consistent-inv-2 cdcl*$_W$*-distinctinv-1 cdcl*$_W$*-bt*
unfolding *cdcl*$_W$*-M-level-inv-def* **by** *meson+*

lemma *rtranclp-cdcl*$_W$*-consistent-inv*:
assumes
cdcl$_W$** *S S'* **and**
cdcl$_W$*-M-level-inv S*
shows *cdcl*$_W$*-M-level-inv S'*
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*auto intro*: *cdcl*$_W$*-consistent-inv*)

lemma *tranclp-cdcl*$_W$*-consistent-inv*:
assumes
cdcl$_W$$^{++}$ *S S'* **and**
cdcl$_W$*-M-level-inv S*
shows *cdcl*$_W$*-M-level-inv S'*
using *assms* **by** (*induct rule*: *tranclp-induct*) (*auto intro*: *cdcl*$_W$*-consistent-inv*)

lemma *cdcl*$_W$*-M-level-inv-S0-cdcl*$_W$[*simp*]:
cdcl$_W$*-M-level-inv* (*init-state N*)
unfolding *cdcl*$_W$*-M-level-inv-def* **by** *auto*

lemma *cdcl*$_W$*-M-level-inv-get-level-le-backtrack-lvl*:
assumes *inv*: *cdcl*$_W$*-M-level-inv S*
shows *get-level* (*trail S*) $L \leq$ *backtrack-lvl S*
using *inv* **unfolding** *cdcl*$_W$*-M-level-inv-def*
by *simp*

lemma *backtrack-ex-decomp*:
assumes
M-l: *cdcl*$_W$*-M-level-inv S* **and**
i-S: $i <$ *backtrack-lvl S*
shows $\exists$ *K M1 M2*. (*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail S*)) $\wedge$
get-level (*trail S*) *K* = *Suc i*
proof −
let *?M* = *trail S*
have $i <$ *count-decided* (*trail S*)
using *i-S M-l* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-def*)
then obtain *c K c'* **where** *tr-S*: *trail S* = *c* @ *Decided K* # *c'* **and**
lev-K: *get-level* (*trail S*) *K* = *Suc i*
using *le-count-decided-decomp*[*of trail S i*] *M-l* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-def*)
obtain *M1 M2* **where** (*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail S*))
using *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons* **unfolding** *tr-S* **by** *fast*
then show *?thesis* **using** *lev-K* **by** *blast*
qed

lemma *backtrack-lvl-backtrack-decrease*:
  assumes *inv*: $cdcl_W$*-M-level-inv S* **and** *bt*: *backtrack S T*
  shows *backtrack-lvl T* < *backtrack-lvl S*
  using *inv bt le-count-decided-decomp*[*of trail S backtrack-lvl T*]
  unfolding $cdcl_W$*-M-level-inv-def*
  by (*fastforce elim*!: *backtrackE dest*!: *get-all-ann-decomposition-exists-prepend*
    simp: *append-assoc*[*of - - -#* -, *symmetric*] *simp del*: *append-assoc*)

Compatibility with *op* ∼

lemma *propagate-state-eq-compatible*:
  assumes
    propa: *propagate S T* **and**
    SS': *S* ∼ *S'* **and**
    TT': *T* ∼ *T'*
  shows *propagate S' T'*
proof −
  obtain *C L* **where**
    conf: *conflicting S* = *None* **and**
    C: *C* ∈# *clauses S* **and**
    L: *L* ∈# *C* **and**
    tr: *trail S* |=*as CNot* (*remove1-mset L C*) **and**
    undef: *undefined-lit* (*trail S*) *L* **and**
    T: *T* ∼ *cons-trail* (*Propagated L C*) *S*
  using *propa* **by** (*elim propagateE*) *auto*

  have *C'*: *C* ∈# *clauses S'*
    using *SS' C*
    by (*auto simp*: *state-eq-def clauses-def simp del*: *state-simp*)

  show *?thesis*
    apply (*rule propagate-rule*[*of - C*])
    using *state-eq-sym*[*of S S'*] *SS' conf C' L tr undef TT' T*
    by (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *conflict-state-eq-compatible*:
  assumes
    confl: *conflict S T* **and**
    TT': *T* ∼ *T'* **and**
    SS': *S* ∼ *S'*
  shows *conflict S' T'*
proof −
  obtain *D* **where**
    conf: *conflicting S* = *None* **and**
    D: *D* ∈# *clauses S* **and**
    tr: *trail S* |=*as CNot D* **and**
    T: *T* ∼ *update-conflicting* (*Some D*) *S*
  using *confl* **by** (*elim conflictE*) *auto*

  have *D'*: *D* ∈# *clauses S'*
    using *D SS'* **by** *fastforce*

  show *?thesis*
    apply (*rule conflict-rule*[*of - D*])

using *state-eq-sym*[*of S S'*] *SS' conf D' tr TT' T*
by (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *backtrack-state-eq-compatible*:
assumes
bt: *backtrack S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$ **and**
inv: $cdcl_W$*-M-level-inv S*
shows *backtrack S' T'*
proof −
obtain *D L K i M1 M2* **where**
conf: *conflicting S = Some D* **and**
L: $L \in\# D$ **and**
decomp: (*Decided K # M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
lev: *get-level* (*trail S*) *L = backtrack-lvl S* **and**
max: *get-level* (*trail S*) *L = get-maximum-level* (*trail S*) *D* **and**
max-D: *get-maximum-level* (*trail S*) (*remove1-mset L D*) ≡ *i* **and**
lev-K: *get-level* (*trail S*) *K = Suc i* **and**
T: $T \sim$ *cons-trail* (*Propagated L D*)
(*reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None S*))))
using *bt inv* **by** (*elim backtrackE*) *metis*
have *D'*: *conflicting S' = Some D*
using *SS' conf* **by** (*cases conflicting S'*) *auto*

have *T'*: $T' \sim$ *cons-trail* (*Propagated L D*)
(*reduce-trail-to M1* (*add-learned-cls D*
(*update-backtrack-lvl i* (*update-conflicting None S'*))))
using *TT'* **unfolding** *state-eq-def*
using *decomp D' inv SS' T* **by** (*auto simp add*: $cdcl_W$*-M-level-inv-def*)

show *?thesis*
apply (*rule backtrack-rule*[*of - D*])
apply (*rule D'*)
using *state-eq-sym*[*of S S'*] *TT' SS' D' conf L decomp lev max max-D T*
apply (*auto simp*: *state-eq-def simp del*: *state-simp*)[]
using *decomp SS' lev SS' max-D max T' lev-K* **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *decide-state-eq-compatible*:
assumes
decide S T **and**
$S \sim S'$ **and**
$T \sim T'$
shows *decide S' T'*
using *assms* **apply** (*elim decideE*)
by (*rule decide-rule*) (*auto simp*: *state-eq-def clauses-def simp del*: *state-simp*)

lemma *skip-state-eq-compatible*:
assumes
skip: *skip S T* **and**
SS': $S \sim S'$ **and**

TT': $T \sim T'$
shows *skip* S' T'
proof −
obtain *L C' M E* **where**
tr: *trail* S = *Propagated L C'* # M **and**
raw: *conflicting S* = *Some E* **and**
L: $-L \notin\# E$ **and**
E: $E \neq \{\#\}$ **and**
T: $T \sim$ *tl-trail S*
using *skip* **by** (*elim skipE*) *simp*
obtain E' **where** E': *conflicting S'* = *Some E'*
using *SS' raw* **by** (*cases conflicting S'*) (*auto simp*: *state-eq-def simp del*: *state-simp*)
show *?thesis*
apply (*rule skip-rule*)
using *tr raw L E T SS'* **apply** (*auto simp*: *simp del*:)[]
using *E'* **apply** *simp*
using *E' SS' L raw E* **apply** (*auto simp*: *state-eq-def simp del*: *state-simp*)[2]
using *T TT' SS'* **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *resolve-state-eq-compatible*:
assumes
res: *resolve S T* **and**
TT': $T \sim T'$ **and**
SS': $S \sim S'$
shows *resolve S' T'*
proof −
obtain *E D L* **where**
tr: *trail* $S \neq []$ **and**
hd: *hd-trail S* = *Propagated L E* **and**
L: $L \in\# E$ **and**
raw: *conflicting S* = *Some D* **and**
LD: $-L \in\# D$ **and**
i: *get-maximum-level* (*trail S*) ((*remove1-mset* $(-L)$ *D*)) = *backtrack-lvl S* **and**
T: $T \sim$ *update-conflicting* (*Some* (*resolve-cls L D E*)) (*tl-trail S*)
using *assms* **by** (*elim resolveE*) *simp*

obtain D' **where**
D': *conflicting S'* = *Some D'*
using *SS' raw* **by** *fastforce*
have [*simp*]: $D = D'$
using *D' SS' raw state-simp*(5) **by** *fastforce*
have *T'T*: $T' \sim T$
using *TT' state-eq-sym* **by** *auto*
show *?thesis*
apply (*rule resolve-rule*)
using *tr SS'* **apply** *simp*
using *hd SS'* **apply** *simp*
using *L* **apply** *simp*
using *D'* **apply** *simp*
using *D' SS' raw LD* **apply** (*auto simp add*: *state-eq-def simp del*: *state-simp*)[]
using *D' SS' raw LD* **apply** (*auto simp add*: *state-eq-def simp del*: *state-simp*)[]
using *raw SS' i* **apply** (*auto simp add*: *state-eq-def simp del*: *state-simp*)[]
using *T T'T SS'* **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *forget-state-eq-compatible*:
  assumes
    forget: *forget S T* **and**
    SS': $S \sim S'$ **and**
    TT': $T \sim T'$
  shows *forget* S' T'
proof –
  obtain C **where**
    conf: *conflicting S = None* **and**
    C: $C \in\#$ *learned-clss S* **and**
    tr: $\neg$(*trail S*) $\models$*asm clauses S* **and**
    C1: $C \notin$ *set* (*get-all-mark-of-propagated* (*trail S*)) **and**
    C2: $C \notin\#$ *init-clss S* **and**
    T: $T \sim$ *remove-cls C S*
    using *forget* **by** (*elim forgetE*) *simp*

  show *?thesis*
    apply (*rule forget-rule*)
         using *SS' conf* **apply** *simp*
        using *C SS'* **apply** *simp*
       using *SS' tr* **apply** *simp*
      using *SS' C1* **apply** *simp*
     using *SS' C2* **apply** *simp*
    using *T TT' SS'* **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma $cdcl_W$*-state-eq-compatible*:
  assumes
    $cdcl_W$ *S T* **and** $\neg$*restart S T* **and**
    $S \sim S'$
    $T \sim T'$ **and**
    $cdcl_W$*-M-level-inv S*
  shows $cdcl_W$ S' T'
  using *assms* **by** (*meson backtrack backtrack-state-eq-compatible bj* $cdcl_W$*.simps* $cdcl_W$*-o-rule-cases*
    $cdcl_W$*-rf.cases conflict-state-eq-compatible decide decide-state-eq-compatible forget*
    forget-state-eq-compatible propagate-state-eq-compatible resolve resolve-state-eq-compatible
    skip skip-state-eq-compatible state-eq-ref)

lemma $cdcl_W$*-bj-state-eq-compatible*:
  assumes
    $cdcl_W$*-bj S T* **and** $cdcl_W$*-M-level-inv S*
    $T \sim T'$
  shows $cdcl_W$*-bj S* T'
  using *assms* **by** (*meson backtrack backtrack-state-eq-compatible* $cdcl_W$*-bjE resolve*
    resolve-state-eq-compatible skip skip-state-eq-compatible state-eq-ref)

lemma *tranclp-*$cdcl_W$*-bj-state-eq-compatible*:
  assumes
    $cdcl_W$*-bj*$^{++}$ *S T* **and** *inv*: $cdcl_W$*-M-level-inv S* **and**
    $S \sim S'$ **and**
    $T \sim T'$
  shows $cdcl_W$*-bj*$^{++}$ S' T'
  using *assms*
proof (*induction arbitrary*: S' T')
  case *base*
  then show *?case*

unfolding *tranclp-unfold-end* **by** (*meson backtrack-state-eq-compatible* $cdcl_W$*-bj.simps resolve-state-eq-compatible rtranclp-unfold skip-state-eq-compatible*)
next
case (*step T U*) **note** *IH* = *this*(3)[*OF this*(4−5)]
have ${cdcl_W}^{++}$ *S T*
using *tranclp-mono*[*of* $cdcl_W$*-bj* $cdcl_W$] *step.hyps*(1) $cdcl_W$*.other* $cdcl_W$*-o.bj* **by** *blast*
then have $cdcl_W$*-M-level-inv T*
using *inv tranclp-*$cdcl_W$*-consistent-inv* **by** *blast*
then have $cdcl_W$*-bj*$^{++}$ *T T′*
using ‹*U* ∼ *T′*› $cdcl_W$*-bj-state-eq-compatible*[*of T U*] ‹$cdcl_W$*-bj T U*› **by** *auto*
then show *?case*
using *IH*[*of T*] **by** *auto*
qed

Conservation of some Properties

lemma $cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o S S′* **and**
inv: $cdcl_W$*-M-level-inv S*
shows *init-clss S* = *init-clss S′*
using *assms* **by** (*induct rule*: $cdcl_W$*-o-induct*) (*auto simp*: *inv* $cdcl_W$*-M-level-inv-decomp*)

lemma *tranclp-*$cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o*$^{++}$ *S S′* **and**
inv: $cdcl_W$*-M-level-inv S*
shows *init-clss S* = *init-clss S′*
using *assms* **apply** (*induct rule*: *tranclp.induct*)
by (*auto dest*: $cdcl_W$*-o-no-more-init-clss*
dest!: *tranclp-*$cdcl_W$*-consistent-inv dest*: *tranclp-mono-explicit*[*of* $cdcl_W$*-o - -* $cdcl_W$]
simp: *other*)

lemma *rtranclp-*$cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o*** *S S′* **and**
inv: $cdcl_W$*-M-level-inv S*
shows *init-clss S* = *init-clss S′*
using *assms* **unfolding** *rtranclp-unfold* **by** (*auto intro*: *tranclp-*$cdcl_W$*-o-no-more-init-clss*)

lemma $cdcl_W$*-init-clss*:
assumes
$cdcl_W$ *S T* **and**
inv: $cdcl_W$*-M-level-inv S*
shows *init-clss S* = *init-clss T*
using *assms* **by** (*induction rule*: $cdcl_W$*-all-induct*)
(*auto simp*: *inv* $cdcl_W$*-M-level-inv-decomp not-in-iff*)

lemma *rtranclp-*$cdcl_W$*-init-clss*:
${cdcl_W}^{**}$ *S T* ⟹ $cdcl_W$*-M-level-inv S* ⟹ *init-clss S* = *init-clss T*
by (*induct rule*: *rtranclp-induct*) (*auto dest*: $cdcl_W$*-init-clss rtranclp-*$cdcl_W$*-consistent-inv*)

lemma *tranclp-*$cdcl_W$*-init-clss*:
${cdcl_W}^{++}$ *S T* ⟹ $cdcl_W$*-M-level-inv S* ⟹ *init-clss S* = *init-clss T*
using *rtranclp-*$cdcl_W$*-init-clss*[*of S T*] **unfolding** *rtranclp-unfold* **by** *auto*

Learned Clause

This invariant shows that:

- the learned clauses are entailed by the initial set of clauses.
- the conflicting clause is entailed by the initial set of clauses.
- the marks are entailed by the clauses.

definition cdcl_W*-learned-clause* $(S :: 'st) \longleftrightarrow$
$(\mathit{init\text{-}clss}\ S \models psm\ \mathit{learned\text{-}clss}\ S$
$\wedge (\forall T.\ \mathit{conflicting}\ S = \mathit{Some}\ T \longrightarrow \mathit{init\text{-}clss}\ S \models pm\ T)$
$\wedge\ \mathit{set}\ (\mathit{get\text{-}all\text{-}mark\text{-}of\text{-}propagated}\ (\mathit{trail}\ S)) \subseteq \mathit{set\text{-}mset}\ (\mathit{clauses}\ S))$

of Weidenbach's book for the inital state and some additional structural properties about the trail.

lemma cdcl_W*-learned-clause-S0-*cdcl_W[*simp*]:
cdcl_W*-learned-clause* (*init-state N*)
unfolding cdcl_W*-learned-clause-def* **by** *auto*

Item 4 page 81 of Weidenbach's book

lemma cdcl_W*-learned-clss*:
assumes
$\mathit{cdcl}_W\ S\ S'$ **and**
learned: cdcl_W*-learned-clause S* **and**
lev-inv: cdcl_W*-M-level-inv S*
shows cdcl_W*-learned-clause* S'
using *assms*(*1*) *lev-inv learned*
proof (*induct rule*: cdcl_W*-all-induct*)
case (*backtrack K i M1 M2 L D T*) **note** *decomp* = *this*(*3*) **and** *confl* = *this*(*1*) **and** *lev-K* = *this* (*7*) **and**
undef = *this*(*8*) **and** *T* = *this*(*9*)
show *?case*
using *decomp confl learned undef T lev-K* **unfolding** cdcl_W*-learned-clause-def*
by (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *clauses-def lev-inv* cdcl_W*-M-level-inv-decomp dest*: *true-clss-clss-left-right*)
next
case (*resolve L C M D*) **note** *trail* = *this*(*1*) **and** *CL* = *this*(*2*) **and** *confl* = *this*(*4*) **and** *DL* = *this*(*5*)
and *lvl* = *this*(*6*) **and** *T* = *this*(*7*)
moreover
have *init-clss S* $\models$*psm learned-clss S*
using *learned trail* **unfolding** cdcl_W*-learned-clause-def clauses-def* **by** *auto*
then have *init-clss S* $\models$*pm C* + {#*L*#}
using *trail learned* **unfolding** cdcl_W*-learned-clause-def clauses-def*
by (*auto dest*: *true-clss-clss-in-imp-true-clss-cls*)
moreover have *remove1-mset* (− *L*) *D* + {#− *L*#} = *D*
using *DL* **by** (*auto simp*: *multiset-eq-iff*)
moreover have *remove1-mset L C* + {#*L*#} = *C*
using *CL* **by** (*auto simp*: *multiset-eq-iff*)
ultimately show *?case*
using *learned T*
by (*auto dest*: *mk-disjoint-insert*
simp add: cdcl_W*-learned-clause-def clauses-def*
intro!: *true-clss-cls-union-mset-true-clss-cls-or-not-true-clss-cls-or*[*of - - L*])

next
  case (*restart T*)
  then show *?case*
    using *learned*
    by (*auto*
      simp: *clauses-def state-eq-def* $cdcl_W$*-learned-clause-def*
      simp del: *state-simp*
      dest: *true-clss-clssm-subsetE*)
next
  case *propagate*
  then show *?case* **using** *learned* **by** (*auto simp*: $cdcl_W$*-learned-clause-def*)
next
  case *conflict*
  then show *?case* **using** *learned*
    by (*fastforce simp*: $cdcl_W$*-learned-clause-def clauses-def*
      true-clss-clss-in-imp-true-clss-cls)
next
  case (*forget U*)
  then show *?case* **using** *learned*
    by (*auto simp*: $cdcl_W$*-learned-clause-def clauses-def split*: *if-split-asm*)
qed (*auto simp*: $cdcl_W$*-learned-clause-def clauses-def*)

lemma *rtranclp-*$cdcl_W$*-learned-clss*:
  assumes
    $cdcl_W^{**}$ S S' **and**
    $cdcl_W$*-M-level-inv S*
    $cdcl_W$*-learned-clause S*
  shows $cdcl_W$*-learned-clause S′*
  using *assms* **by** *induction* (*auto dest*: $cdcl_W$*-learned-clss intro*: *rtranclp-*$cdcl_W$*-consistent-inv*)

No alien atom in the state

This invariant means that all the literals are in the set of clauses. These properties are implicit in Weidenbach's book.

definition *no-strange-atm S′* ⟷ (
    (∀ *T*. *conflicting S′* = *Some T* ⟶ *atms-of T* ⊆ *atms-of-mm* (*init-clss S′*))
  ∧ (∀ *L mark*. *Propagated L mark* ∈ *set* (*trail S′*)
      ⟶ *atms-of mark* ⊆ *atms-of-mm* (*init-clss S′*))
  ∧ *atms-of-mm* (*learned-clss S′*) ⊆ *atms-of-mm* (*init-clss S′*)
  ∧ *atm-of* ‘ (*lits-of-l* (*trail S′*)) ⊆ *atms-of-mm* (*init-clss S′*))

lemma *no-strange-atm-decomp*:
  assumes *no-strange-atm S*
  shows *conflicting S* = *Some T* ⟹ *atms-of T* ⊆ *atms-of-mm* (*init-clss S*)
  and (∀ *L mark*. *Propagated L mark* ∈ *set* (*trail S*)
    ⟶ *atms-of mark* ⊆ *atms-of-mm* (*init-clss S*))
  and *atms-of-mm* (*learned-clss S*) ⊆ *atms-of-mm* (*init-clss S*)
  and *atm-of* ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*init-clss S*)
  using *assms* **unfolding** *no-strange-atm-def* **by** *blast+*

lemma *no-strange-atm-S0* [*simp*]: *no-strange-atm* (*init-state N*)
  unfolding *no-strange-atm-def* **by** *auto*

lemma *in-atms-of-implies-atm-of-on-atms-of-ms*:
  C + {#*L*#} ∈# *A* ⟹ *x* ∈ *atms-of C* ⟹ *x* ∈ *atms-of-mm A*

using *multi-member-split* **by** *fastforce*

lemma *propagate-no-strange-atm-inv*:
assumes
propagate S T **and**
alien: *no-strange-atm S*
shows *no-strange-atm T*
using *assms(1)*
proof (*induction*)
case (*propagate-rule C L T*) **note** *confl* = *this*(*1*) **and** *C* = *this*(*2*) **and** *C-L* = *this*(*3*) **and** *tr* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *T* = *this*(*6*)
have *atm-CL*: *atms-of C* ⊆ *atms-of-mm* (*init-clss S*)
using *C alien* **unfolding** *no-strange-atm-def*
by (*auto simp*: *clauses-def atms-of-ms-def*)
show *?case*
unfolding *no-strange-atm-def*
proof (*intro conjI allI impI*, *goal-cases*)
case *1*
then show *?case*
using *confl T undef* **by** *auto*
next
case (*2 L′ mark′*)
then show *?case*
using *C-L T alien undef atm-CL* **unfolding** *no-strange-atm-def clauses-def* **by** (*auto 5 5*)
next
case (*3*)
show *?case* **using** *T alien undef* **unfolding** *no-strange-atm-def* **by** *auto*
next
case (*4*)
show *?case*
using *T alien undef C-L atm-CL* **unfolding** *no-strange-atm-def* **by** (*auto simp*: *atms-of-def*)
qed
qed

lemma *in-atms-of-remove1-mset-in-atms-of*:
x ∈ *atms-of* (*remove1-mset L C*) ⟹ *x* ∈ *atms-of C*
using *in-diffD* **unfolding** *atms-of-def* **by** *fastforce*

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*:
atms-of-mm (*learned-clss S*) ⊆ *atms-of-mm* (*init-clss S*) ⟹
x ∈ *atms-of-mm* (*learned-clss T*) ⟹
learned-clss T ⊆# *learned-clss S* ⟹
x ∈ *atms-of-mm* (*init-clss S*)
by (*meson atms-of-ms-mono contra-subsetD set-mset-mono*)

lemma *$cdcl_W$-no-strange-atm-explicit*:
assumes
$cdcl_W$ S S′ **and**
lev: *$cdcl_W$-M-level-inv S* **and**
conf: ∀ *T*. *conflicting S* = *Some T* ⟶ *atms-of T* ⊆ *atms-of-mm* (*init-clss S*) **and**
decided: ∀ *L mark*. *Propagated L mark* ∈ *set* (*trail S*)
⟶ *atms-of mark* ⊆ *atms-of-mm* (*init-clss S*) **and**
learned: *atms-of-mm* (*learned-clss S*) ⊆ *atms-of-mm* (*init-clss S*) **and**
trail: *atm-of* ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*init-clss S*)
shows
(∀ *T*. *conflicting S′* = *Some T* ⟶ *atms-of T* ⊆ *atms-of-mm* (*init-clss S′*)) ∧

($\forall L$ *mark. Propagated L mark* $\in$ *set* (*trail* S')
  $\longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss* S')) $\wedge$
atms-of-mm (*learned-clss* S') $\subseteq$ *atms-of-mm* (*init-clss* S') $\wedge$
atm-of ' (*lits-of-l* (*trail* S')) $\subseteq$ *atms-of-mm* (*init-clss* S')
(**is** *?C* S' $\wedge$ *?M* S' $\wedge$ *?U* S' $\wedge$ *?V* S')
using *assms*(*1*,*2*)
proof (*induct rule*: $cdcl_W$*-all-induct*)
case (*propagate C L T*) **note** *confl* = *this*(*1*) **and** *C-L* = *this*(*2*) **and** *tr* = *this*(*3*) **and** *undef* = *this*(*4*)
and *T* = *this*(*5*)
show *?case*
using *propagate-rule*[*OF propagate.hyps*(*1*−*3*) - *propagate.hyps*(*5*,*6*), *simplified*]
propagate.hyps(*4*) *propagate-no-strange-atm-inv*[*of S T*]
conf decided learned trail **unfolding** *no-strange-atm-def* **by** *presburger*
next
case (*decide L*)
then show *?case* **using** *learned decided conf trail* **unfolding** *clauses-def* **by** *auto*
next
case (*skip L C M D*)
then show *?case* **using** *learned decided conf trail* **by** *auto*
next
case (*conflict D T*) **note** *D-S* = *this*(*2*) **and** *T* = *this*(*4*)
have *D*: *atm-of* ' *set-mset D* $\subseteq \bigcup$(*atms-of* ' (*set-mset* (*clauses S*)))
using *D-S* **by** (*auto simp add*: *atms-of-def atms-of-ms-def*)
moreover {
fix *xa* :: *'v literal*
assume *a1*: *atm-of* ' *set-mset D* $\subseteq$ ($\bigcup$*x*$\in$*set-mset* (*init-clss S*). *atms-of x*)
$\cup$ ($\bigcup$*x*$\in$*set-mset* (*learned-clss S*). *atms-of x*)
assume *a2*:
($\bigcup$*x*$\in$*set-mset* (*learned-clss S*). *atms-of x*) $\subseteq$ ($\bigcup$*x*$\in$*set-mset* (*init-clss S*). *atms-of x*)
assume *xa* $\in\#$ *D*
then have *atm-of xa* $\in$ *UNION* (*set-mset* (*init-clss S*)) *atms-of*
using *a2 a1* **by** (*metis* (*no-types*) *Un-iff atm-of-lit-in-atms-of atms-of-def subset-Un-eq*)
then have $\exists$ *m*$\in$*set-mset* (*init-clss S*). *atm-of xa* $\in$ *atms-of m*
by *blast*
} **note** *H* = *this*
ultimately show *?case* **using** *conflict.prems T learned decided conf trail*
unfolding *atms-of-def atms-of-ms-def clauses-def*
by (*auto simp add*: *H*)
next
case (*restart T*)
then show *?case* **using** *learned decided conf trail*
by (*auto intro*: *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*)
next
case (*forget C T*) **note** *confl* = *this*(*1*) **and** *C* = *this*(*4*) **and** *C-le* = *this*(*5*) **and**
T = *this*(*6*)
have *H*: $\bigwedge$*L mark. Propagated L mark* $\in$ *set* (*trail S*) $\Longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss S*)
using *decided* **by** *simp*
show *?case* **unfolding** *clauses-def* **apply** (*intro conjI*)
using *conf confl T trail C* **unfolding** *clauses-def* **apply** (*auto dest*!: *H*)[]
using *T trail C C-le* **apply** (*auto dest*!: *H*)[]
using *T learned C-le atms-of-ms-remove-subset*[*of set-mset* (*learned-clss S*)] **apply** *auto*[]
using *T trail C-le* **apply** (*auto simp*: *clauses-def lits-of-def*)[]
done
next
case (*backtrack L D K i M1 M2 T*) **note** *confl* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *decomp* = *this*(*3*)

and
  lev-K = *this*(*7*) **and** *T* = *this*(*8*)
 have *?C T*
  using *conf T decomp lev lev-K* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
 moreover have *set M1* ⊆ *set* (*trail S*)
  using *decomp* **by** *auto*
 then have *M*: *?M T*
  using *decided conf confl T decomp lev lev-K*
  by (*auto simp*: *image-subset-iff clauses-def* $cdcl_W$*-M-level-inv-decomp*)
 moreover have *?U T*
  using *learned decomp conf confl T lev lev-K* **unfolding** *clauses-def*
  by (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
 moreover have *?V T*
  using *M conf confl trail T decomp lev LD lev-K*
  by (*auto simp*: $cdcl_W$*-M-level-inv-decomp atms-of-def*
   dest!: *get-all-ann-decomposition-exists-prepend*)
 ultimately show *?case* **by** *blast*
next
 case (*resolve L C M D T*) **note** *trail-S* = *this*(*1*) **and** *confl* = *this*(*4*) **and** *T* = *this*(*7*)
 let *?T* = *update-conflicting* (*Some* (*resolve-cls L D C*)) (*tl-trail S*)
 have *?C ?T*
  using *confl trail-S conf decided* **by** (*auto dest*!: *in-atms-of-remove1-mset-in-atms-of*)
 moreover have *?M ?T*
  using *confl trail-S conf decided* **by** *auto*
 moreover have *?U ?T*
  using *trail learned* **by** *auto*
 moreover have *?V ?T*
  using *confl trail-S trail* **by** *auto*
 ultimately show *?case* **using** *T* **by** *simp*
qed

lemma $cdcl_W$*-no-strange-atm-inv*:
 assumes $cdcl_W$ *S S'* **and** *no-strange-atm S* **and** $cdcl_W$*-M-level-inv S*
 shows *no-strange-atm S'*
 using $cdcl_W$*-no-strange-atm-explicit*[*OF assms*(*1*)] *assms*(*2*,*3*) **unfolding** *no-strange-atm-def* **by** *fast*

lemma *rtranclp-*$cdcl_W$*-no-strange-atm-inv*:
 assumes $cdcl_W^{**}$ *S S'* **and** *no-strange-atm S* **and** $cdcl_W$*-M-level-inv S*
 shows *no-strange-atm S'*
 using *assms* **by** *induction* (*auto intro*: $cdcl_W$*-no-strange-atm-inv rtranclp-*$cdcl_W$*-consistent-inv*)

No Duplicates all Around

This invariant shows that there is no duplicate (no literal appearing twice in the formula). The last part could be proven using the previous invariant also. Remark that we will show later that there cannot be duplicate *clause*.

definition *distinct-*$cdcl_W$*-state* (*S* ::'*st*)
 ⟷ ((∀ *T*. *conflicting S* = *Some T* ⟶ *distinct-mset T*)
  ∧ *distinct-mset-mset* (*learned-clss S*)
  ∧ *distinct-mset-mset* (*init-clss S*)
  ∧ (∀ *L mark*. (*Propagated L mark* ∈ *set* (*trail S*) ⟶ *distinct-mset mark*)))

lemma *distinct-*$cdcl_W$*-state-decomp*:
 assumes *distinct-*$cdcl_W$*-state* (*S* ::'*st*)
 shows

$\forall T$. *conflicting* S = *Some* $T \longrightarrow$ *distinct-mset* T **and**
distinct-mset-mset (*learned-clss* S) **and**
distinct-mset-mset (*init-clss* S) **and**
$\forall L$ *mark*. (*Propagated* L *mark* $\in$ *set* (*trail* S) $\longrightarrow$ *distinct-mset mark*)
using *assms* **unfolding** *distinct-cdcl$_W$-state-def* **by** *blast+*

lemma *distinct-cdcl$_W$-state-decomp-2*:
assumes *distinct-cdcl$_W$-state* (S ::'*st*) **and** *conflicting* S = *Some* T
shows *distinct-mset* T
using *assms* **unfolding** *distinct-cdcl$_W$-state-def* **by** *auto*

lemma *distinct-cdcl$_W$-state-S0-cdcl$_W$*[*simp*]:
distinct-mset-mset $N \Longrightarrow$ *distinct-cdcl$_W$-state* (*init-state* N)
unfolding *distinct-cdcl$_W$-state-def* **by** *auto*

lemma *distinct-cdcl$_W$-state-inv*:
assumes
cdcl$_W$ S S' **and**
lev-inv: *cdcl$_W$-M-level-inv* S **and**
distinct-cdcl$_W$-state S
shows *distinct-cdcl$_W$-state* S'
using *assms*(*1*,*2*,*2*,*3*)
proof (*induct rule*: *cdcl$_W$-all-induct*)
case (*backtrack L D K i M1 M2*)
then show *?case*
using *lev-inv* **unfolding** *distinct-cdcl$_W$-state-def*
by (*auto dest*: *get-all-ann-decomposition-incl simp*: *cdcl$_W$-M-level-inv-decomp*)
next
case *restart*
then show *?case*
unfolding *distinct-cdcl$_W$-state-def distinct-mset-set-def clauses-def* **by** *auto*
next
case *resolve*
then show *?case*
by (*auto simp add*: *distinct-cdcl$_W$-state-def distinct-mset-set-def clauses-def*
distinct-mset-single-add
intro!: *distinct-mset-union-mset*)
qed (*auto simp*: *distinct-cdcl$_W$-state-def distinct-mset-set-def clauses-def*
dest!: *in-diffD*)

lemma *rtanclp-distinct-cdcl$_W$-state-inv*:
assumes
cdcl$_W$** S S' **and**
cdcl$_W$-M-level-inv S **and**
distinct-cdcl$_W$-state S
shows *distinct-cdcl$_W$-state* S'
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
using *distinct-cdcl$_W$-state-inv rtranclp-cdcl$_W$-consistent-inv* **by** *blast+*

Conflicts and Annotations

This invariant shows that each mark contains a contradiction only related to the previously defined variable.

abbreviation *every-mark-is-a-conflict* :: '*st* $\Rightarrow$ *bool* **where**
every-mark-is-a-conflict S $\equiv$

$\forall L$ *mark a b. a* @ *Propagated L mark* # *b* = (*trail S*)
  $\longrightarrow$ (*b* $\models$*as CNot* (*mark* $-$ {#*L*#}) $\wedge$ *L* $\in$# *mark*)

definition *cdcl*$_W$*-conflicting S* $\longleftrightarrow$
  ($\forall$ *T. conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as CNot T*)
  $\wedge$ *every-mark-is-a-conflict S*

lemma *backtrack-atms-of-D-in-M1*:
  fixes *M1* :: ('*v*, '*v clause*) *ann-lits*
  assumes
    inv: *cdcl*$_W$*-M-level-inv S* **and**
    i: *get-maximum-level* (*trail S*) ((*remove1-mset L D*)) $\equiv$ *i* **and**
    decomp: (*Decided K* # *M1*, *M2*)
      $\in$ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
    S-lvl: *backtrack-lvl S* = *get-maximum-level* (*trail S*) *D* **and**
    S-confl: *conflicting S* = *Some D* **and**
    lev-K: *get-level* (*trail S*) *K* = *Suc i* **and**
    T: *T* $\sim$ *cons-trail* (*Propagated L D*)
            (*reduce-trail-to M1*
              (*add-learned-cls D*
                (*update-backtrack-lvl i*
                  (*update-conflicting None S*)))) **and**
    confl: $\forall$ *T. conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as CNot T*
  shows *atms-of* ((*remove1-mset L D*)) $\subseteq$ *atm-of* ' *lits-of-l* (*tl* (*trail T*))
proof (*rule ccontr*)
  let *?k* = *get-maximum-level* (*trail S*) *D*
  let *?D'* = *remove1-mset L D*
  have *trail S* $\models$*as CNot D* **using** *confl S-confl* **by** *auto*
  then have *vars-of-D*: *atms-of D* $\subseteq$ *atm-of* ' *lits-of-l* (*trail S*) **unfolding** *atms-of-def*
    by (*meson image-subsetI true-annots-CNot-all-atms-defined*)

  obtain *M0* **where** *M*: *trail S* = *M0* @ *M2* @ *Decided K* # *M1*
    using *decomp* **by** *auto*

  have *max*: *?k* = *count-decided* (*M0* @ *M2* @ *Decided K* # *M1*)
    using *inv* **unfolding** *cdcl*$_W$*-M-level-inv-def S-lvl M* **by** *simp*
  assume *a*: $\neg$ *?thesis*
  then obtain *L'* **where**
    L': *L'* $\in$ *atms-of ?D'* **and**
    L'-notin-M1: *L'* $\notin$ *atm-of* ' *lits-of-l M1*
    using *T decomp inv* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
  then have *L'-in*: *L'* $\in$ *atm-of* ' *lits-of-l* (*M0* @ *M2* @ *Decided K* # [])
    using *vars-of-D* **unfolding** *M* **by** (*auto dest*: *in-atms-of-remove1-mset-in-atms-of*)
  then obtain *L''* **where**
    L'' $\in$# *?D'* **and**
    L'': *L'* = *atm-of L''*
    using *L' L'-notin-M1* **unfolding** *atms-of-def* **by** *auto*
  have *atm-of K* $\notin$ *atm-of* ' *lits-of-l* (*M0* @ *M2*)
    using *inv* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-def M lits-of-def*)
  then have *count-decided M1* = *i*
    using *lev-K* **unfolding** *M* **by** (*auto simp*: *image-Un*)
  then have *lev-L''*:
    get-level (*trail S*) *L''* = *get-level* (*M0* @ *M2* @ *Decided K* # []) *L''* + *i*
    using *L'-notin-M1 L'' get-rev-level-skip-end*[*OF L'-in*[*unfolded L''*], *of M1*] *M* **by** *auto*
  moreover
    consider

($M0$) $L' \in$ *atm-of* ‘ *lits-of-l M0* |
($M2$) $L' \in$ *atm-of* ‘ *lits-of-l M2* |
(K) $L' =$ *atm-of K*
using *inv L'-in* **unfolding** L'' **by** (*auto simp*: *cdcl$_W$-M-level-inv-def*)
then have *get-level* (*M0* @ *M2* @ *Decided K* # []) $L'' \geq$ *Suc 0*
proof *cases*
case *M0*
then have $L' \neq$ *atm-of K*
using *inv* ‹*atm-of K* ∉ *atm-of* ‘ *lits-of-l* (*M0* @ *M2*)› **unfolding** L'' **by** *auto*
then show *?thesis* **using** *M0* **unfolding** L'' **by** *auto*
next
case *M2*
then have $L' \notin$ *atm-of* ‘ *lits-of-l* (*M0* @ *Decided K* # [])
using *inv* ‹*atm-of K* ∉ *atm-of* ‘ *lits-of-l* (*M0* @ *M2*)› **unfolding** L''
by (*auto simp*: *M cdcl$_W$-M-level-inv-def atm-lit-of-set-lits-of-l*)
then show *?thesis* **using** *M2* **unfolding** L'' **by** (*auto simp*: *image-Un*)
next
case *K*
then have $L' \notin$ *atm-of* ‘ *lits-of-l* (*M0* @ *M2*)
using *inv* **unfolding** L'' **by** (*auto simp*: *cdcl$_W$-M-level-inv-def atm-lit-of-set-lits-of-l M*)
then show *?thesis* **using** *K* **unfolding** L'' **by** (*auto simp*: *image-Un*)
qed
ultimately have *get-level* (*trail S*) $L'' \geq i + 1$
using *lev-*L'' **unfolding** *M* **by** *simp*
then have *get-maximum-level* (*trail S*) *?D'* $\geq i + 1$
using *get-maximum-level-ge-get-level*[*OF* ‹L'' ∈# *?D'*›, *of trail S*] **by** *auto*
then show *False* **using** *i* **by** *auto*
qed

lemma *distinct-atms-of-incl-not-in-other*:
assumes
a1: *no-dup* (*M* @ *M'*) **and**
a2: *atms-of D* ⊆ *atm-of* ‘ *lits-of-l M'* **and**
a3: *x* ∈ *atms-of D*
shows *x* ∉ *atm-of* ‘ *lits-of-l M*
proof −
have *ff1*: ⋀*l ms. undefined-lit ms l* ∨ *atm-of l*
∈ *set* (*map* (λ*m. atm-of* (*lit-of* (*m* ::('*a*, '*b*) *ann-lit*))) *ms*)
by (*simp add*: *defined-lit-map*)
have *ff2*: ⋀*a. a* ∉ *atms-of D* ∨ *a* ∈ *atm-of* ‘ *lits-of-l M'*
using *a2* **by** (*meson subsetCE*)
have *ff3*: ⋀*a. a* ∉ *set* (*map* (λ*m. atm-of* (*lit-of m*)) *M'*)
∨ *a* ∉ *set* (*map* (λ*m. atm-of* (*lit-of m*)) *M*)
using *a1* **by** (*metis* (*lifting*) *IntI distinct-append empty-iff map-append*)
have ∀ *L a f*. ∃ *l*. ((*a*::'*a*) ∉ *f* ‘ *L* ∨ (*l* ::'*a literal*) ∈ *L*) ∧ (*a* ∉ *f* ‘ *L* ∨ *f l* = *a*)
by *blast*
then show *x* ∉ *atm-of* ‘ *lits-of-l M*
using *ff3 ff2 ff1 a3* **by** (*metis* (*no-types*) *Decided-Propagated-in-iff-in-lits-of-l*)
qed

Item 5 page 81 of Weidenbach's book

lemma *cdcl$_W$-propagate-is-conclusion*:
assumes
cdcl$_W$ S S' **and**
inv: *cdcl$_W$-M-level-inv S* **and**
decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**

learned: *cdcl*$_W$*-learned-clause S* **and**
confl: $\forall$ *T*. *conflicting S = Some T* $\longrightarrow$ *trail S* $\models$*as CNot T* **and**
alien: *no-strange-atm S*
shows *all-decomposition-implies-m* (*init-clss S'*) (*get-all-ann-decomposition* (*trail S'*))
using *assms*(*1*,*2*)
proof (*induct rule*: *cdcl*$_W$*-all-induct*)
case *restart*
then show *?case* **by** *auto*
next
case *forget*
then show *?case* **using** *decomp* **by** *auto*
next
case *conflict*
then show *?case* **using** *decomp* **by** *auto*
next
case (*resolve L C M D*) **note** *tr* = *this*(*1*) **and** *T* = *this*(*7*)
let *?decomp* = *get-all-ann-decomposition M*
have *M*: *set ?decomp* = *insert* (*hd ?decomp*) (*set* (*tl ?decomp*))
by (*cases ?decomp*) *auto*
show *?case*
using *decomp tr T* **unfolding** *all-decomposition-implies-def*
by (*cases hd* (*get-all-ann-decomposition M*))
(*auto simp*: *M*)
next
case (*skip L C' M D*) **note** *tr* = *this*(*1*) **and** *T* = *this*(*5*)
have *M*: *set* (*get-all-ann-decomposition M*)
= *insert* (*hd* (*get-all-ann-decomposition M*)) (*set* (*tl* (*get-all-ann-decomposition M*)))
by (*cases get-all-ann-decomposition M*) *auto*
show *?case*
using *decomp tr T* **unfolding** *all-decomposition-implies-def*
by (*cases hd* (*get-all-ann-decomposition M*))
(*auto simp add*: *M*)
next
case *decide* **note** *S* = *this*(*1*) **and** *undef* = *this*(*2*) **and** *T* = *this*(*4*)
show *?case* **using** *decomp T undef* **unfolding** *S all-decomposition-implies-def* **by** *auto*
next
case (*propagate C L T*) **note** *propa* = *this*(*2*) **and** *L* = *this*(*3*) **and** *undef* = *this*(*5*) **and** *T* = *this*(*6*)
obtain *a y* **where** *ay*: *hd* (*get-all-ann-decomposition* (*trail S*)) = (*a*, *y*)
by (*cases hd* (*get-all-ann-decomposition* (*trail S*)))
then have *M*: *trail S* = *y* @ *a* **using** *get-all-ann-decomposition-decomp* **by** *blast*
have *M'*: *set* (*get-all-ann-decomposition* (*trail S*))
= *insert* (*a*, *y*) (*set* (*tl* (*get-all-ann-decomposition* (*trail S*))))
using *ay* **by** (*cases get-all-ann-decomposition* (*trail S*)) *auto*
have *unmark-l a* $\cup$ *set-mset* (*init-clss S*) $\models$*ps unmark-l y*
using *decomp ay* **unfolding** *all-decomposition-implies-def*
by (*cases get-all-ann-decomposition* (*trail S*)) *fastforce+*
then have *a-Un-N-M*: *unmark-l a* $\cup$ *set-mset* (*init-clss S*)
$\models$*ps unmark-l* (*trail S*)
unfolding *M* **by** (*auto simp add*: *all-in-true-clss-clss image-Un*)

have *unmark-l a* $\cup$ *set-mset* (*init-clss S*) $\models$*p* {#*L*#} (**is** *?I* $\models$*p* -)
proof (*rule true-clss-cls-plus-CNot*)
show *?I* $\models$*p remove1-mset L C* + {#*L*#}
apply (*rule true-clss-clss-in-imp-true-clss-cls*[*of* -
set-mset (*init-clss S*) $\cup$ *set-mset* (*learned-clss S*)])
using *learned propa L* **by** (*auto simp*: *clauses-def cdcl*$_W$*-learned-clause-def*

true-annot-CNot-diff)
next
have *unmark-l* (*trail S*) $\models$*ps CNot* (*remove1-mset L C*)
using ‹(*trail S*) $\models$*as CNot* (*remove1-mset L C*)› *true-annots-true-clss-clss*
by *blast*
then show *?I* $\models$*ps CNot* (*remove1-mset L C*)
using *a-Un-N-M true-clss-clss-left-right true-clss-clss-union-l-r* **by** *blast*
qed
moreover have $\bigwedge$*aa b*.
$\forall$ (*Ls*, *seen*)$\in$*set* (*get-all-ann-decomposition* (*y* @ *a*)).
unmark-l Ls $\cup$ *set-mset* (*init-clss S*) $\models$*ps unmark-l seen* $\Longrightarrow$
(*aa*, *b*) $\in$ *set* (*tl* (*get-all-ann-decomposition* (*y* @ *a*))) $\Longrightarrow$
unmark-l aa $\cup$ *set-mset* (*init-clss S*) $\models$*ps unmark-l b*
by (*metis* (*no-types*, *lifting*) *case-prod-conv get-all-ann-decomposition-never-empty-sym*
list.collapse list.set-intros(*2*))

ultimately show *?case*
using *decomp T undef* **unfolding** *ay all-decomposition-implies-def*
using *M* ‹*unmark-l a* $\cup$ *set-mset* (*init-clss S*) $\models$*ps unmark-l y*›
ay **by** *auto*
next
case (*backtrack L D K i M1 M2 T*) **note** *conf* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *decomp'* = *this*(*3*)
and
lev-L = *this*(*4*) **and** *lev-K* = *this*(*7*) **and** *undef* = *this*(*8*) **and** *T* = *this*(*9*)
let *?D'* = *remove1-mset L D*
have $\forall$ *l* $\in$ *set M2*. $\neg$*is-decided l*
using *get-all-ann-decomposition-snd-not-decided decomp'* **by** *blast*
obtain *M0* **where** *M*: *trail S* = *M0* @ *M2* @ *Decided K* # *M1*
using *decomp'* **by** *auto*
show *?case* **unfolding** *all-decomposition-implies-def*
proof
fix *x*
assume *x* $\in$ *set* (*get-all-ann-decomposition* (*trail T*))
then have *x*: *x* $\in$ *set* (*get-all-ann-decomposition* (*Propagated L D* # *M1*))
using *T decomp' undef inv* **by** (*simp add*: $cdcl_W$*-M-level-inv-decomp*)
let *?m* = *get-all-ann-decomposition* (*Propagated L D* # *M1*)
let *?hd* = *hd ?m*
let *?tl* = *tl ?m*
consider
(*hd*) *x* = *?hd*
| (*tl*) *x* $\in$ *set ?tl*
using *x* **by** (*cases ?m*) *auto*
then show *case x of* (*Ls*, *seen*) $\Rightarrow$ *unmark-l Ls* $\cup$ *set-mset* (*init-clss T*) $\models$*ps unmark-l seen*
proof *cases*
case *tl*
then have *x* $\in$ *set* (*get-all-ann-decomposition* (*trail S*))
using *tl-get-all-ann-decomposition-skip-some*[*of x*] **by** (*simp add*: *list.set-sel*(*2*) *M*)
then show *?thesis*
using *decomp learned decomp confl alien inv T undef M*
unfolding *all-decomposition-implies-def* $cdcl_W$*-M-level-inv-def*
by *auto*
next
case *hd*
obtain *M1' M1''* **where** *M1*: *hd* (*get-all-ann-decomposition M1*) = (*M1'*, *M1''*)
by (*cases hd* (*get-all-ann-decomposition M1*))
then have *x'*: *x* = (*M1'*, *Propagated L D* # *M1''*)

    using ⟨x = *?hd*⟩ **by** *auto*
  have (*M1′*, *M1″*) ∈ *set* (*get-all-ann-decomposition* (*trail S*))
    using *M1*[*symmetric*] *hd-get-all-ann-decomposition-skip-some*[*OF M1*[*symmetric*],
      of M0 @ *M2*] **unfolding** *M* **by** *fastforce*
  then have *1*: *unmark-l M1′* ∪ *set-mset* (*init-clss S*) ⊨*ps unmark-l M1″*
    using *decomp* **unfolding** *all-decomposition-implies-def* **by** *auto*

  moreover
    have *vars-of-D*: *atms-of ?D′* ⊆ *atm-of* ‘ *lits-of-l M1*
      using *backtrack-atms-of-D-in-M1*[*of S D L i K M1 M2 T*] *backtrack.hyps inv conf confl*
      by (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
    have *no-dup* (*trail S*) **using** *inv* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
    then have *vars-in-M1*:
      ∀ *x* ∈ *atms-of ?D′*. *x* ∉ *atm-of* ‘ *lits-of-l* (*M0* @ *M2* @ *Decided K* # [])
      using *vars-of-D distinct-atms-of-incl-not-in-other*[*of*
        M0 @*M2* @ *Decided K* # [] *M1*] **unfolding** *M* **by** *auto*
    have *trail S* ⊨*as CNot* (*remove1-mset L D*)
      using *conf confl LD* **unfolding** *M true-annots-true-cls-def-iff-negation-in-model*
      by (*auto dest*!: *Multiset.in-diffD*)
    then have *M1* ⊨*as CNot ?D′*
      using *vars-in-M1 true-annots-remove-if-notin-vars*[*of M0* @ *M2* @ *Decided K* # []
        M1 CNot ?D′] *conf confl* **unfolding** *M lits-of-def* **by** *simp*
    have *M1* = *M1″* @ *M1′* **by** (*simp add*: *M1 get-all-ann-decomposition-decomp*)
    have *TT*: *unmark-l M1′* ∪ *set-mset* (*init-clss S*) ⊨*ps CNot ?D′*
      using *true-annots-true-clss-cls*[*OF* ⟨*M1* ⊨*as CNot ?D′*⟩] *true-clss-clss-left-right*[*OF 1*]
      unfolding ⟨*M1* = *M1″* @ *M1′*⟩ **by** (*auto simp add*: *inf-sup-aci*(*5*,*7*))
    have *init-clss S* ⊨*pm ?D′* + {#*L*#}
      using *conf learned confl LD* **unfolding** $cdcl_W$*-learned-clause-def* **by** *auto*
    then have *T′*: *unmark-l M1′* ∪ *set-mset* (*init-clss S*) ⊨*p ?D′* + {#*L*#} **by** *auto*
    have *atms-of* (*?D′* + {#*L*#}) ⊆ *atms-of-mm* (*clauses S*)
      using *alien conf LD* **unfolding** *no-strange-atm-def clauses-def* **by** *auto*
    then have *unmark-l M1′* ∪ *set-mset* (*init-clss S*) ⊨*p* {#*L*#}
      using *true-clss-cls-plus-CNot*[*OF T′ TT*] **by** *auto*

  ultimately show *?thesis*
      using *T′ T decomp′ undef inv* **unfolding** *x′* **by** (*simp add*: $cdcl_W$*-M-level-inv-decomp*)
  qed
 qed
qed

lemma $cdcl_W$*-propagate-is-false*:
 assumes
  $cdcl_W$ *S S′* **and**
  lev: $cdcl_W$*-M-level-inv S* **and**
  learned: $cdcl_W$*-learned-clause S* **and**
  decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**
  confl: ∀ *T*. *conflicting S* = *Some T* ⟶ *trail S* ⊨*as CNot T* **and**
  alien: *no-strange-atm S* **and**
  mark-confl: *every-mark-is-a-conflict S*
 shows *every-mark-is-a-conflict S′*
 using *assms*(*1*,*2*)
proof (*induct rule*: $cdcl_W$*-all-induct*)
 case (*propagate C L T*) **note** *LC* = *this*(*3*) **and** *confl* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *T* = *this*(*6*)
 show *?case*
  proof (*intro allI impI*)

```
      fix L′ mark a b
      assume a @ Propagated L′ mark # b = trail T
      then consider
          (hd) a = [] and L = L′ and mark = C and b = trail S
        | (tl) tl a @ Propagated L′ mark # b = trail S
        using T undef by (cases a) fastforce+
      then show b ⊨as CNot (mark − {#L′#}) ∧ L′ ∈# mark
        using mark-confl confl LC by cases auto
    qed
next
  case (decide L) note undef[simp] = this(2) and T = this(4)
  have ⋀a La mark b. a @ Propagated La mark # b = Decided L # trail S
    ⟹ tl a @ Propagated La mark # b = trail S by (case-tac a) auto
  then show ?case using mark-confl T unfolding decide.hyps(1) by fastforce
next
  case (skip L C′ M D T) note tr = this(1) and T = this(5)
  show ?case
    proof (intro allI impI)
      fix L′ mark a b
      assume a @ Propagated L′ mark # b = trail T
      then have a @ Propagated L′ mark # b = M using tr T by simp
      then have (Propagated L C′ # a) @ Propagated L′ mark # b = Propagated L C′ # M by auto
      moreover have ∀ La mark a b. a @ Propagated La mark # b = Propagated L C′ # M
        ⟶ b ⊨as CNot (mark − {#La#}) ∧ La ∈# mark
        using mark-confl unfolding skip.hyps(1) by simp
      ultimately show b ⊨as CNot (mark − {#L′#}) ∧ L′ ∈# mark by blast
    qed
next
  case (conflict D)
  then show ?case using mark-confl by simp
next
  case (resolve L C M D T) note tr-S = this(1) and T = this(7)
  show ?case unfolding resolve.hyps(1)
    proof (intro allI impI)
      fix L′ mark a b
      assume a @ Propagated L′ mark # b = trail T
      then have (Propagated L (C + {#L#}) # a) @ Propagated L′ mark # b
        = Propagated L (C + {#L#}) # M
        using T tr-S by auto
      then show b ⊨as CNot (mark − {#L′#}) ∧ L′ ∈# mark
        using mark-confl unfolding tr-S by (metis Cons-eq-appendI list.sel(3))
    qed
next
  case restart
  then show ?case by auto
next
  case forget
  then show ?case using mark-confl by auto
next
  case (backtrack L D K i M1 M2 T) note conf = this(1) and LD = this(2) and decomp = this(3)
and
    lev-K = this(7) and T = this(8)
  have ∀ l ∈ set M2. ¬is-decided l
    using get-all-ann-decomposition-snd-not-decided decomp by blast
  obtain M0 where M: trail S = M0 @ M2 @ Decided K # M1
    using decomp by auto
```

have [*simp*]: *trail* (*reduce-trail-to M1* (*add-learned-cls D*
(*update-backtrack-lvl i* (*update-conflicting None S*)))) = *M1*
using *decomp lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
let *?D′* = *remove1-mset L D*
show *?case*
proof (*intro allI impI*)
fix *La* :: *′v literal* **and** *mark* :: *′v clause* **and**
a b :: (*′v*, *′v clause*) *ann-lits*
assume *a* @ *Propagated La mark* # *b* = *trail T*
then consider
(*hd-tr*) *a* = [] **and**
(*Propagated La mark* :: (*′v*, *′v clause*) *ann-lit*) = *Propagated L D* **and**
b = *M1*
| (*tl-tr*) *tl a* @ *Propagated La mark* # *b* = *M1*
using *M T decomp lev* **by** (*cases a*) (*auto simp*: $cdcl_W$*-M-level-inv-def*)
then show *b* ⊨*as CNot* (*mark* − {#*La*#}) ∧ *La* ∈# *mark*
proof *cases*
case *hd-tr* **note** *A* = *this*(*1*) **and** *P* = *this*(*2*) **and** *b* = *this*(*3*)
have *trail S* ⊨*as CNot D* **using** *conf confl* **by** *auto*
then have *vars-of-D*: *atms-of D* ⊆ *atm-of* ‘ *lits-of-l* (*trail S*)
unfolding *atms-of-def*
by (*meson image-subsetI true-annots-CNot-all-atms-defined*)
have *vars-of-D*: *atms-of ?D′* ⊆ *atm-of* ‘ *lits-of-l M1*
using *backtrack-atms-of-D-in-M1*[*of S D L i K M1 M2 T*] *T backtrack lev confl*
by (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
have *no-dup* (*trail S*) **using** *lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
then have ∀ *x* ∈ *atms-of ?D′*. *x* ∉ *atm-of* ‘ *lits-of-l* (*M0* @ *M2* @ *Decided K* # [])
using *vars-of-D distinct-atms-of-incl-not-in-other*[*of*
M0 @ *M2* @ *Decided K* # [] *M1*] **unfolding** *M* **by** *auto*
then have *M1* ⊨*as CNot ?D′*
using *true-annots-remove-if-notin-vars*[*of M0* @ *M2* @ *Decided K* # []
M1 CNot ?D′] ‹*trail S* ⊨*as CNot D*› **unfolding** *M lits-of-def*
by (*simp add*: *true-annot-CNot-diff*)
then show *b* ⊨*as CNot* (*mark* − {#*La*#}) ∧ *La* ∈# *mark*
using *P LD b* **by** *auto*
next
case *tl-tr*
then obtain *c′* **where** *c′* @ *Propagated La mark* # *b* = *trail S*
unfolding *M* **by** *auto*
then show *b* ⊨*as CNot* (*mark* − {#*La*#}) ∧ *La* ∈# *mark*
using *mark-confl* **by** *auto*
qed
qed
qed

lemma $cdcl_W$*-conflicting-is-false*:
assumes
$cdcl_W$ *S S′* **and**
M-lev: $cdcl_W$*-M-level-inv S* **and**
confl-inv: ∀ *T*. *conflicting S* = *Some T* ⟶ *trail S* ⊨*as CNot T* **and**
decided-confl: ∀ *L mark a b*. *a* @ *Propagated L mark* # *b* = (*trail S*)
⟶ (*b* ⊨*as CNot* (*mark* − {#*L*#}) ∧ *L* ∈# *mark*) **and**
dist: *distinct-*$cdcl_W$*-state S*
shows ∀ *T*. *conflicting S′* = *Some T* ⟶ *trail S′* ⊨*as CNot T*
using *assms*(*1*,*2*)
proof (*induct rule*: $cdcl_W$*-all-induct*)

case (*skip L C' M D T*) **note** *tr-S* = *this*(*1*) **and** *confl* = *this*(*2*) **and** *L-D* = *this*(*3*) **and** *T* = *this*(*5*)
have *D*: *Propagated L C' # M* $\models$*as CNot D* **using** *assms skip* **by** *auto*
moreover
have *L* $\notin$# *D*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then have $-$ *L* $\in$ *lits-of-l M*
using *in-CNot-implies-uminus*(*2*)[*of L D Propagated L C' # M*]
⟨*Propagated L C' # M* $\models$*as CNot D*⟩ **by** *simp*
then show *False*
by (*metis* (*no-types*, *hide-lams*) *M-lev* $cdcl_W$*-M-level-inv-decomp*(*1*) *consistent-interp-def image-insert insert-iff list.set*(*2*) *lits-of-def ann-lit.sel*(*2*) *tr-S*)
qed
ultimately show *?case*
using *tr-S confl L-D T* **unfolding** $cdcl_W$*-M-level-inv-def*
by (*auto intro*: *true-annots-CNot-lit-of-notin-skip*)
next
case (*resolve L C M D T*) **note** *tr* = *this*(*1*) **and** *LC* = *this*(*2*) **and** *confl* = *this*(*4*) **and** *LD* = *this*(*5*)
and *T* = *this*(*7*)
let *?C* = *remove1-mset L C*
let *?D* = *remove1-mset* ($-$*L*) *D*
show *?case*
proof (*intro allI impI*)
fix *T'*
have *tl* (*trail S*) $\models$*as CNot ?C* **using** *tr decided-confl* **by** *fastforce*
moreover
have *distinct-mset* (*?D* + {#$-$ *L*#}) **using** *confl dist LD*
unfolding *distinct-*$cdcl_W$*-state-def* **by** *auto*
then have $-$*L* $\notin$# *?D* **unfolding** *distinct-mset-def*
by (*meson* ⟨*distinct-mset* (*?D* + {#$-$ *L*#})⟩ *distinct-mset-single-add*)
have *M* $\models$*as CNot ?D*
proof $-$
have *Propagated L* (*?C* + {#*L*#}) # *M* $\models$*as CNot ?D* $\cup$ *CNot* {#$-$ *L*#}
using *confl tr confl-inv LC* **by** (*metis CNot-plus LD insert-DiffM2*)
then show *?thesis*
using *M-lev* ⟨$-$ *L* $\notin$# *?D*⟩ *tr true-annots-lit-of-notin-skip*
unfolding $cdcl_W$*-M-level-inv-def* **by** *force*
qed
moreover assume *conflicting T* = *Some T'*
ultimately
show *trail T* $\models$*as CNot T'*
using *tr T* **by** *auto*
qed
qed (*auto simp*: *M-lev* $cdcl_W$*-M-level-inv-decomp*)

lemma $cdcl_W$*-conflicting-decomp*:
assumes $cdcl_W$*-conflicting S*
shows $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as CNot T*
and $\forall$ *L mark a b*. *a* @ *Propagated L mark* # *b* = (*trail S*)
$\longrightarrow$ (*b* $\models$*as CNot* (*mark* $-$ {#*L*#}) $\wedge$ *L* $\in$# *mark*)
using *assms* **unfolding** $cdcl_W$*-conflicting-def* **by** *blast+*

lemma $cdcl_W$*-conflicting-decomp2*:
assumes $cdcl_W$*-conflicting S* **and** *conflicting S* = *Some T*

shows *trail* $S \models as$ *CNot* T
using *assms* **unfolding** $cdcl_W$*-conflicting-def* **by** *blast+*

lemma $cdcl_W$*-conflicting-S0-*$cdcl_W$[*simp*]:
$cdcl_W$*-conflicting* (*init-state* N)
unfolding $cdcl_W$*-conflicting-def* **by** *auto*

Putting all the invariants together

lemma $cdcl_W$*-all-inv*:
assumes
$cdcl_W$: $cdcl_W$ S S' **and**
1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
2: $cdcl_W$*-learned-clause* S **and**
4: $cdcl_W$*-M-level-inv* S **and**
5: *no-strange-atm* S **and**
7: *distinct-*$cdcl_W$*-state* S **and**
8: $cdcl_W$*-conflicting* S
shows
all-decomposition-implies-m (*init-clss* S') (*get-all-ann-decomposition* (*trail* S')) **and**
$cdcl_W$*-learned-clause* S' **and**
$cdcl_W$*-M-level-inv* S' **and**
no-strange-atm S' **and**
distinct-$cdcl_W$*-state* S' **and**
$cdcl_W$*-conflicting* S'
proof −
show *S1*: *all-decomposition-implies-m* (*init-clss* S') (*get-all-ann-decomposition* (*trail* S'))
using $cdcl_W$*-propagate-is-conclusion*[*OF* $cdcl_W$ *4 1 2 - 5*] *8* **unfolding** $cdcl_W$*-conflicting-def*
by *blast*
show *S2*: $cdcl_W$*-learned-clause* S' **using** $cdcl_W$*-learned-clss*[*OF* $cdcl_W$ *2 4*] **.**
show *S4*: $cdcl_W$*-M-level-inv* S' **using** $cdcl_W$*-consistent-inv*[*OF* $cdcl_W$ *4*] **.**
show *S5*: *no-strange-atm* S' **using** $cdcl_W$*-no-strange-atm-inv*[*OF* $cdcl_W$ *5 4*] **.**
show *S7*: *distinct-*$cdcl_W$*-state* S' **using** *distinct-*$cdcl_W$*-state-inv*[*OF* $cdcl_W$ *4 7*] **.**
show *S8*: $cdcl_W$*-conflicting* S'
using $cdcl_W$*-conflicting-is-false*[*OF* $cdcl_W$ *4 - - 7*] *8* $cdcl_W$*-propagate-is-false*[*OF* $cdcl_W$ *4 2 1 - 5*]
unfolding $cdcl_W$*-conflicting-def* **by** *fast*
qed

lemma *rtranclp-*$cdcl_W$*-all-inv*:
assumes
$cdcl_W$: *rtranclp* $cdcl_W$ S S' **and**
1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
2: $cdcl_W$*-learned-clause* S **and**
4: $cdcl_W$*-M-level-inv* S **and**
5: *no-strange-atm* S **and**
7: *distinct-*$cdcl_W$*-state* S **and**
8: $cdcl_W$*-conflicting* S
shows
all-decomposition-implies-m (*init-clss* S') (*get-all-ann-decomposition* (*trail* S')) **and**
$cdcl_W$*-learned-clause* S' **and**
$cdcl_W$*-M-level-inv* S' **and**
no-strange-atm S' **and**
distinct-$cdcl_W$*-state* S' **and**
$cdcl_W$*-conflicting* S'
using *assms*

proof (*induct rule*: *rtranclp-induct*)
  case *base*
    case *1* **then show** *?case* **by** *blast*
    case *2* **then show** *?case* **by** *blast*
    case *3* **then show** *?case* **by** *blast*
    case *4* **then show** *?case* **by** *blast*
    case *5* **then show** *?case* **by** *blast*
    case *6* **then show** *?case* **by** *blast*
next
  case (*step* S' S'') **note** $H = this$
    case *1* **with** $H(3-7)$[*OF this*$(1-6)$] **show** *?case* **using** $cdcl_W$*-all-inv*[*OF* $H(2)$]
      H **by** *presburger*
    case *2* **with** $H(3-7)$[*OF this*$(1-6)$] **show** *?case* **using** $cdcl_W$*-all-inv*[*OF* $H(2)$]
      H **by** *presburger*
    case *3* **with** $H(3-7)$[*OF this*$(1-6)$] **show** *?case* **using** $cdcl_W$*-all-inv*[*OF* $H(2)$]
      H **by** *presburger*
    case *4* **with** $H(3-7)$[*OF this*$(1-6)$] **show** *?case* **using** $cdcl_W$*-all-inv*[*OF* $H(2)$]
      H **by** *presburger*
    case *5* **with** $H(3-7)$[*OF this*$(1-6)$] **show** *?case* **using** $cdcl_W$*-all-inv*[*OF* $H(2)$]
      H **by** *presburger*
    case *6* **with** $H(3-7)$[*OF this*$(1-6)$] **show** *?case* **using** $cdcl_W$*-all-inv*[*OF* $H(2)$]
      H **by** *presburger*
qed

lemma *all-invariant-S0-*$cdcl_W$:
  assumes *distinct-mset-mset N*
  shows
    all-decomposition-implies-m (*init-clss* (*init-state N*))
                    (*get-all-ann-decomposition* (*trail* (*init-state N*))) **and**
    $cdcl_W$*-learned-clause* (*init-state N*) **and**
    $\forall$ *T*. *conflicting* (*init-state N*) = *Some T* $\longrightarrow$ (*trail* (*init-state N*))$\models$*as CNot T* **and**
    no-strange-atm (*init-state N*) **and**
    consistent-interp (*lits-of-l* (*trail* (*init-state N*))) **and**
    $\forall$ *L mark a b*. *a* @ *Propagated L mark* # *b* = *trail* (*init-state N*) $\longrightarrow$
     (*b* $\models$*as CNot* (*mark* − {#*L*#}) $\wedge$ *L* $\in$# *mark*) **and**
     distinct-$cdcl_W$*-state* (*init-state N*)
  using *assms* **by** *auto*

Item 6 page 81 of Weidenbach's book

lemma $cdcl_W$*-only-propagated-vars-unsat*:
  assumes
    decided: $\forall$ *x* $\in$ *set M*. $\neg$ *is-decided x* **and**
    DN: *D* $\in$# *clauses S* **and**
    D: *M* $\models$*as CNot D* **and**
    inv: *all-decomposition-implies-m N* (*get-all-ann-decomposition M*) **and**
    state: *state S* = (*M*, *N*, *U*, *k*, *C*) **and**
    learned-cl: $cdcl_W$*-learned-clause S* **and**
    atm-incl: *no-strange-atm S*
  shows *unsatisfiable* (*set-mset N*)
proof (*rule ccontr*)
  assume $\neg$ *unsatisfiable* (*set-mset N*)
  then obtain *I* **where**
    I: *I* $\models$*s set-mset N* **and**
    cons: *consistent-interp I* **and**
    tot: *total-over-m I* (*set-mset N*)
    unfolding *satisfiable-def* **by** *auto*

have *atms-of-mm N* ∪ *atms-of-mm U* = *atms-of-mm N*
using *atm-incl state* **unfolding** *total-over-m-def no-strange-atm-def*
by (*auto simp add*: *clauses-def*)
then have *total-over-m I* (*set-mset N*) **using** *tot* **unfolding** *total-over-m-def* **by** *auto*
moreover then have *total-over-m I* (*set-mset* (*learned-clss S*))
using *atm-incl state* **unfolding** *no-strange-atm-def total-over-m-def total-over-set-def*
by *auto*
moreover have *N* ⊨*psm U* **using** *learned-cl state* **unfolding** $cdcl_W$*-learned-clause-def* **by** *auto*
ultimately have *I-D*: *I* ⊨ *D*
using *I DN cons state* **unfolding** *true-clss-clss-def true-clss-def Ball-def*
by (*auto simp add*: *clauses-def*)

have *l0*: {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set M*} = {} **using** *decided* **by** *auto*
have *atms-of-ms* (*set-mset N* ∪ *unmark-l M*) = *atms-of-mm N*
using *atm-incl state* **unfolding** *no-strange-atm-def* **by** *auto*
then have *total-over-m I* (*set-mset N* ∪ *unmark-l M*)
using *tot* **unfolding** *total-over-m-def* **by** *auto*
then have *I* ⊨*s unmark-l M*
using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons I*
unfolding *true-clss-clss-def l0* **by** *auto*
then have *IM*: *I* ⊨*s unmark-l M* **by** *auto*
{
fix *K*
assume *K* ∈# *D*
then have −*K* ∈ *lits-of-l M*
using *D* **unfolding** *true-annots-def Ball-def CNot-def true-annot-def true-cls-def true-lit-def Bex-def* **by** *force*
then have −*K* ∈ *I* **using** *IM true-clss-singleton-lit-of-implies-incl lits-of-def* **by** *fastforce* **}**
then have ¬ *I* ⊨ *D* **using** *cons* **unfolding** *true-cls-def true-lit-def consistent-interp-def* **by** *auto*
then show *False* **using** *I-D* **by** *blast*
qed

Item 5 page 81 of Weidenbach's book

We have actually a much stronger theorem, namely *all-decomposition-implies-propagated-lits-are-implied*, that show that the only choices we made are decided in the formula

lemma
assumes *all-decomposition-implies-m N* (*get-all-ann-decomposition M*)
and ∀ *m* ∈ *set M*. ¬*is-decided m*
shows *set-mset N* ⊨*ps unmark-l M*
proof −
have *T*: {*unmark L* |*L*. *is-decided L* ∧ *L* ∈ *set M*} = {} **using** *assms*(*2*) **by** *auto*
then show *?thesis*
using *all-decomposition-implies-propagated-lits-are-implied*[*OF assms*(*1*)] **unfolding** *T* **by** *simp*
qed

Item 7 page 81 of Weidenbach's book (part 1)

lemma *conflict-with-false-implies-unsat*:
assumes
$cdcl_W$: $cdcl_W$ *S S'* **and**
lev: $cdcl_W$*-M-level-inv S* **and**
[*simp*]: *conflicting S'* = *Some* {#} **and**
learned: $cdcl_W$*-learned-clause S*
shows *unsatisfiable* (*set-mset* (*init-clss S*))
using *assms*
proof −

have *cdcl*$_W$*-learned-clause* S' **using** *cdcl*$_W$*-learned-clss cdcl*$_W$ *learned lev* **by** *auto*
then have *init-clss* $S' \models pm$ $\{\#\}$ **using** *assms*(*3*) **unfolding** *cdcl*$_W$*-learned-clause-def* **by** *auto*
then have *init-clss* $S \models pm$ $\{\#\}$
using *cdcl*$_W$*-init-clss*[*OF assms*(*1*) *lev*] **by** *auto*
then show *?thesis* **unfolding** *satisfiable-def true-clss-cls-def* **by** *auto*
qed

Item 7 page 81 of Weidenbach's book (part 2)

lemma *conflict-with-false-implies-terminated*:
assumes *cdcl*$_W$ S S'
and *conflicting* S = *Some* $\{\#\}$
shows *False*
using *assms* **by** (*induct rule*: *cdcl*$_W$*-all-induct*) *auto*

No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

lemma *learned-clss-are-not-tautologies*:
assumes
cdcl$_W$ S S' **and**
lev: *cdcl*$_W$*-M-level-inv* S **and**
conflicting: *cdcl*$_W$*-conflicting* S **and**
no-tauto: $\forall$ *s* $\in\#$ *learned-clss* S. $\neg$*tautology s*
shows $\forall$ *s* $\in\#$ *learned-clss* S'. $\neg$*tautology s*
using *assms*
proof (*induct rule*: *cdcl*$_W$*-all-induct*)
case (*backtrack L D K i M1 M2 T*) **note** *confl* = *this*(*1*)
have *consistent-interp* (*lits-of-l* (*trail S*)) **using** *lev* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
moreover
have *trail* $S \models as$ *CNot D*
using *conflicting confl* **unfolding** *cdcl*$_W$*-conflicting-def* **by** *auto*
then have *lits-of-l* (*trail S*) $\models s$ *CNot D*
using *true-annots-true-cls* **by** *blast*
ultimately have $\neg$*tautology D* **using** *consistent-CNot-not-tautology* **by** *blast*
then show *?case* **using** *backtrack no-tauto lev*
by (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp split*: *if-split-asm*)
next
case *restart*
then show *?case* **using** *state-eq-learned-clss no-tauto*
by (*auto intro*: *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*)
qed (*auto dest*!: *in-diffD*)

definition *final-cdcl*$_W$*-state* (S :: *'st*)
$\longleftrightarrow$ (*trail* $S \models asm$ *init-clss* S
$\vee$ (($\forall$ $L \in$ *set* (*trail S*). $\neg$*is-decided L*) $\wedge$
($\exists$ $C \in\#$ *init-clss* S. *trail* $S \models as$ *CNot C*)))

definition *termination-cdcl*$_W$*-state* (S :: *'st*)
$\longleftrightarrow$ (*trail* $S \models asm$ *init-clss* S
$\vee$ (($\forall$ $L \in$ *atms-of-mm* (*init-clss S*). $L \in$ *atm-of* ‘ *lits-of-l* (*trail S*))
$\wedge$ ($\exists$ $C \in\#$ *init-clss* S. *trail* $S \models as$ *CNot C*)))

6.1.4 CDCL Strong Completeness

lemma *cdcl*$_W$*-can-do-step*:
  assumes
    consistent-interp (*set M*) **and**
    distinct M **and**
    atm-of ' (*set M*) $\subseteq$ *atms-of-mm N*
  shows $\exists$ *S*. *rtranclp cdcl*$_W$ (*init-state N*) *S*
    $\wedge$ *state S* = (*map* (λ*L*. *Decided L*) *M*, *N*, {#}, *length M*, *None*)
  using *assms*
proof (*induct M*)
  case *Nil*
  then show *?case* **apply** − **by** (*rule exI*[*of* - *init-state N*]) *auto*
next
  case (*Cons L M*) **note** *IH* = *this*(*1*)
  have *consistent-interp* (*set M*) **and** *distinct M* **and** *atm-of* ' *set M* $\subseteq$ *atms-of-mm N*
    using *Cons.prems*(*1*−*3*) **unfolding** *consistent-interp-def* **by** *auto*
  then obtain *S* **where**
    st: *cdcl*$_W$** (*init-state N*) *S* **and**
    S: *state S* = (*map* (λ*L*. *Decided L*) *M*, *N*, {#}, *length M*, *None*)
    using *IH* **by** *blast*
  let *?S*$_0$ = *incr-lvl* (*cons-trail* (*Decided L*) *S*)
  have *undefined-lit* (*map* (λ*L*. *Decided L*) *M*) *L*
    using *Cons.prems*(*1*,*2*) **unfolding** *defined-lit-def consistent-interp-def* **by** *fastforce*
  moreover have *init-clss S* = *N*
    using *S* **by** *blast*
  moreover have *atm-of L* $\in$ *atms-of-mm N* **using** *Cons.prems*(*3*) **by** *auto*
  moreover have *undef*: *undefined-lit* (*trail S*) *L*
    using *S* ‹*distinct* (*L*#*M*)› *calculation*(*1*) **by** (*auto simp*: *defined-lit-map*)
  ultimately have *cdcl*$_W$ *S ?S*$_0$
    using *cdcl*$_W$*.other*[*OF cdcl*$_W$*-o.decide*[*OF decide-rule*[*of S L ?S*$_0$]]] *S*
    by (*auto simp*: *state-eq-def simp del*: *state-simp*)
  then have *cdcl*$_W$** (*init-state N*) *?S*$_0$
    using *st* **by** *auto*
  then show *?case*
    using *S undef* **by** (*auto intro*!: *exI*[*of* - *?S*$_0$] *del*: *simp del*:)
qed

theorem 2.9.11 page 84 of Weidenbach's book

lemma *cdcl*$_W$*-strong-completeness*:
  assumes
    MN: *set M* $\models$*sm N* **and**
    cons: *consistent-interp* (*set M*) **and**
    dist: *distinct M* **and**
    atm: *atm-of* ' (*set M*) $\subseteq$ *atms-of-mm N*
  obtains *S* **where**
    state S = (*map* (λ*L*. *Decided L*) *M*, *N*, {#}, *length M*, *None*) **and**
    rtranclp cdcl$_W$ (*init-state N*) *S* **and**
    final-cdcl$_W$*-state S*
proof −
  obtain *S* **where**
    st: *rtranclp cdcl*$_W$ (*init-state N*) *S* **and**
    S: *state S* = (*map* (λ*L*. *Decided L*) *M*, *N*, {#}, *length M*, *None*)
    using *cdcl*$_W$*-can-do-step*[*OF cons dist atm*] **by** *auto*
  have *lits-of-l* (*map* (λ*L*. *Decided L*) *M*) = *set M*
    by (*induct M*, *auto*)

then have *map* ($\lambda L.$ *Decided L*) *M* $\models$*asm N* **using** *MN true-annots-true-cls* **by** *metis*
then have *final-cdcl$_W$-state S*
using *S* **unfolding** *final-cdcl$_W$-state-def* **by** *auto*
then show *?thesis* **using** *that st S* **by** *blast*
qed

6.1.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

Definition

lemma *tranclp-conflict*:
tranclp conflict S S' $\Longrightarrow$ *conflict S S'*
apply (*induct rule*: *tranclp.induct*)
apply *simp*
by (*metis conflictE conflicting-update-conflicting option.distinct*(*1*) *state-eq-conflicting*)

lemma *tranclp-conflict-iff*[*iff*]:
full1 conflict S S' $\longleftrightarrow$ *conflict S S'*
proof –
have *tranclp conflict S S'* $\Longrightarrow$ *conflict S S'* **by** (*meson tranclp-conflict rtranclpD*)
then show *?thesis* **unfolding** *full1-def*
by (*metis conflict.simps conflicting-update-conflicting option.distinct*(*1*)
state-eq-conflicting tranclp.intros(*1*))
qed

inductive *cdcl$_W$-cp* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **where**
conflict'[*intro*]: *conflict S S'* $\Longrightarrow$ *cdcl$_W$-cp S S'* |
propagate': *propagate S S'* $\Longrightarrow$ *cdcl$_W$-cp S S'*

lemma *rtranclp-cdcl$_W$-cp-rtranclp-cdcl$_W$*:
cdcl$_W$-cp** S T $\Longrightarrow$ *cdcl$_W$** S T*
by (*induction rule*: *rtranclp-induct*) (*auto simp*: *cdcl$_W$-cp.simps dest*: *cdcl$_W$.intros*)

lemma *cdcl$_W$-cp-state-eq-compatible*:
assumes
cdcl$_W$-cp S T **and**
S $\sim$ *S'* **and**
T $\sim$ *T'*
shows *cdcl$_W$-cp S' T'*
using *assms*
apply (*induction*)
using *conflict-state-eq-compatible* **apply** *auto*[*1*]
using *propagate' propagate-state-eq-compatible* **by** *auto*

lemma *tranclp-cdcl$_W$-cp-state-eq-compatible*:
assumes
cdcl$_W$-cp^{++} S T **and**
S $\sim$ *S'* **and**
T $\sim$ *T'*
shows *cdcl$_W$-cp^{++} S' T'*
using *assms*
proof *induction*
case *base*

then show *?case*
using $cdcl_W$*-cp-state-eq-compatible* **by** *blast*
next
case (*step U V*)
obtain *ss* :: *'st* **where**
$cdcl_W$*-cp S ss* **and** $cdcl_W$*-cp*** *ss U*
by (*metis* (*no-types*) *step*(*1*) *tranclpD*)
then show *?case*
by (*meson* $cdcl_W$*-cp-state-eq-compatible rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp2 state-eq-ref step*(*2*) *step*(*4*) *step*(*5*))
qed

lemma *option-full-*$cdcl_W$*-cp*:
conflicting $S \neq$ *None* $\Longrightarrow$ *full* $cdcl_W$*-cp S S*
unfolding *full-def rtranclp-unfold tranclp-unfold*
by (*auto simp add*: $cdcl_W$*-cp.simps elim*: *conflictE propagateE*)

lemma *skip-unique*:
skip S T $\Longrightarrow$ *skip S T'* $\Longrightarrow$ $T \sim T'$
by (*fastforce simp*: *state-eq-def simp del*: *state-simp elim*: *skipE*)

lemma *resolve-unique*:
resolve S T $\Longrightarrow$ *resolve S T'* $\Longrightarrow$ $T \sim T'$
by (*fastforce simp*: *state-eq-def simp del*: *state-simp elim*: *resolveE*)

lemma $cdcl_W$*-cp-no-more-clauses*:
assumes $cdcl_W$*-cp S S'*
shows *clauses S* = *clauses S'*
using *assms* **by** (*induct rule*: $cdcl_W$*-cp.induct*) (*auto elim!*: *conflictE propagateE*)

lemma *tranclp-*$cdcl_W$*-cp-no-more-clauses*:
assumes $cdcl_W$*-cp*$^{++}$ *S S'*
shows *clauses S* = *clauses S'*
using *assms* **by** (*induct rule*: *tranclp.induct*) (*auto dest*: $cdcl_W$*-cp-no-more-clauses*)

lemma *rtranclp-*$cdcl_W$*-cp-no-more-clauses*:
assumes $cdcl_W$*-cp*** *S S'*
shows *clauses S* = *clauses S'*
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*fastforce dest*: $cdcl_W$*-cp-no-more-clauses*)+

lemma *no-conflict-after-conflict*:
conflict S T $\Longrightarrow$ $\neg$*conflict T U*
by (*metis conflictE conflicting-update-conflicting option.distinct*(*1*) *state-simp*(*5*))

lemma *no-propagate-after-conflict*:
conflict S T $\Longrightarrow$ $\neg$*propagate T U*
by (*metis conflictE conflicting-update-conflicting option.distinct*(*1*) *propagate.cases state-eq-conflicting*)

lemma *tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not*:
assumes $cdcl_W$*-cp*$^{++}$ *S U*
shows (*propagate*$^{++}$ *S U* $\wedge$ *conflicting U* = *None*)
$\vee$ ($\exists$ *T D*. *propagate*** *S T* $\wedge$ *conflict T U* $\wedge$ *conflicting U* = *Some D*)
proof −
have *propagate*$^{++}$ *S U* $\vee$ ($\exists$ *T*. *propagate*** *S T* $\wedge$ *conflict T U*)
using *assms* **by** *induction*

(*force simp*: $cdcl_W$*-cp.simps tranclp-into-rtranclp dest*: *no-conflict-after-conflict no-propagate-after-conflict*)+
moreover
have $propagate^{++}$ *S U* $\Longrightarrow$ *conflicting U = None*
unfolding *tranclp-unfold-end* **by** (*auto elim*!: *propagateE*)
moreover
have $\bigwedge T$. *conflict T U* $\Longrightarrow$ $\exists D$. *conflicting U = Some D*
by (*auto elim*!: *conflictE simp*: *state-eq-def simp del*: *state-simp*)
ultimately show *?thesis* **by** *meson*
qed

lemma $cdcl_W$*-cp-conflicting-not-empty*[*simp*]: *conflicting S = Some D* $\Longrightarrow$ $\neg cdcl_W$*-cp S S'*
proof
assume $cdcl_W$*-cp S S'* **and** *conflicting S = Some D*
then show *False* **by** (*induct rule*: $cdcl_W$*-cp.induct*)
(*auto elim*: *conflictE propagateE simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *no-step-*$cdcl_W$*-cp-no-conflict-no-propagate*:
assumes *no-step* $cdcl_W$*-cp S*
shows *no-step conflict S* **and** *no-step propagate S*
using *assms conflict'* **apply** *blast*
by (*meson assms conflict' propagate'*)

CDCL with the reasonable strategy: we fully propagate the conflict and propagate, then we apply any other possible rule $cdcl_W$*-o S S'* and re-apply conflict and propagate $cdcl_W$-$cp^{\downarrow}$ *S' S''*

inductive $cdcl_W$*-stgy* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
conflict': *full1* $cdcl_W$*-cp S S'* $\Longrightarrow$ $cdcl_W$*-stgy S S'* |
other': $cdcl_W$*-o S S'* $\Longrightarrow$ *no-step* $cdcl_W$*-cp S* $\Longrightarrow$ *full* $cdcl_W$*-cp S' S''* $\Longrightarrow$ $cdcl_W$*-stgy S S''*

Invariants

These are the same invariants as before, but lifted

lemma $cdcl_W$*-cp-learned-clause-inv*:
assumes $cdcl_W$*-cp S S'*
shows *learned-clss S = learned-clss S'*
using *assms* **by** (*induct rule*: $cdcl_W$*-cp.induct*) (*fastforce elim*: *conflictE propagateE*)+

lemma *rtranclp-*$cdcl_W$*-cp-learned-clause-inv*:
assumes $cdcl_W$-cp^{**} *S S'*
shows *learned-clss S = learned-clss S'*
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*fastforce dest*: $cdcl_W$*-cp-learned-clause-inv*)+

lemma *tranclp-*$cdcl_W$*-cp-learned-clause-inv*:
assumes $cdcl_W$-cp^{++} *S S'*
shows *learned-clss S = learned-clss S'*
using *assms* **by** (*simp add*: *rtranclp-*$cdcl_W$*-cp-learned-clause-inv tranclp-into-rtranclp*)

lemma $cdcl_W$*-cp-backtrack-lvl*:
assumes $cdcl_W$*-cp S S'*
shows *backtrack-lvl S = backtrack-lvl S'*
using *assms* **by** (*induct rule*: $cdcl_W$*-cp.induct*) (*fastforce elim*: *conflictE propagateE*)+

lemma *rtranclp-*$cdcl_W$*-cp-backtrack-lvl*:

assumes $cdcl_W$-*cp*** *S S'*
shows *backtrack-lvl S* = *backtrack-lvl S'*
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*fastforce dest*: $cdcl_W$-*cp-backtrack-lvl*)+

lemma $cdcl_W$-*cp-consistent-inv*:
assumes $cdcl_W$-*cp S S'* **and** $cdcl_W$-*M-level-inv S*
shows $cdcl_W$-*M-level-inv S'*
using *assms*
proof (*induct rule*: $cdcl_W$-*cp.induct*)
case (*conflict'*)
then show *?case* **using** $cdcl_W$-*consistent-inv* $cdcl_W$.*conflict* **by** *blast*
next
case (*propagate' S S'*)
have $cdcl_W$ *S S'*
using *propagate'.hyps*(*1*) *propagate* **by** *blast*
then show $cdcl_W$-*M-level-inv S'*
using *propagate'.prems*(*1*) $cdcl_W$-*consistent-inv propagate* **by** *blast*
qed

lemma *full1*-$cdcl_W$-*cp-consistent-inv*:
assumes *full1* $cdcl_W$-*cp S S'* **and** $cdcl_W$-*M-level-inv S*
shows $cdcl_W$-*M-level-inv S'*
using *assms* **unfolding** *full1-def*
by (*metis rtranclp*-$cdcl_W$-*cp-rtranclp*-$cdcl_W$ *rtranclp-unfold tranclp*-$cdcl_W$-*consistent-inv*)

lemma *rtranclp*-$cdcl_W$-*cp-consistent-inv*:
assumes *rtranclp* $cdcl_W$-*cp S S'* **and** $cdcl_W$-*M-level-inv S*
shows $cdcl_W$-*M-level-inv S'*
using *assms* **unfolding** *full1-def*
by (*induction rule*: *rtranclp-induct*) (*blast intro*: $cdcl_W$-*cp-consistent-inv*)+

lemma $cdcl_W$-*stgy-consistent-inv*:
assumes $cdcl_W$-*stgy S S'* **and** $cdcl_W$-*M-level-inv S*
shows $cdcl_W$-*M-level-inv S'*
using *assms* **apply** (*induct rule*: $cdcl_W$-*stgy.induct*)
unfolding *full-unfold* **by** (*blast intro*: $cdcl_W$-*consistent-inv full1*-$cdcl_W$-*cp-consistent-inv*
$cdcl_W$.*other*)+

lemma *rtranclp*-$cdcl_W$-*stgy-consistent-inv*:
assumes $cdcl_W$-*stgy*** *S S'* **and** $cdcl_W$-*M-level-inv S*
shows $cdcl_W$-*M-level-inv S'*
using *assms* **by** *induction* (*auto dest*!: $cdcl_W$-*stgy-consistent-inv*)

lemma $cdcl_W$-*cp-no-more-init-clss*:
assumes $cdcl_W$-*cp S S'*
shows *init-clss S* = *init-clss S'*
using *assms* **by** (*induct rule*: $cdcl_W$-*cp.induct*) (*auto elim*: *conflictE propagateE*)

lemma *tranclp*-$cdcl_W$-*cp-no-more-init-clss*:
assumes $cdcl_W$-*cp*$^{++}$ *S S'*
shows *init-clss S* = *init-clss S'*
using *assms* **by** (*induct rule*: *tranclp.induct*) (*auto dest*: $cdcl_W$-*cp-no-more-init-clss*)

lemma $cdcl_W$-*stgy-no-more-init-clss*:
assumes $cdcl_W$-*stgy S S'* **and** $cdcl_W$-*M-level-inv S*
shows *init-clss S* = *init-clss S'*

using *assms*
apply (*induct rule*: *cdcl$_W$-stgy.induct*)
unfolding *full1-def full-def* **apply** (*blast dest*: *tranclp-cdcl$_W$-cp-no-more-init-clss tranclp-cdcl$_W$-o-no-more-init-clss*)
by (*metis cdcl$_W$-o-no-more-init-clss rtranclp-unfold tranclp-cdcl$_W$-cp-no-more-init-clss*)

lemma *rtranclp-cdcl$_W$-stgy-no-more-init-clss*:
assumes *cdcl$_W$-stgy** S S′* **and** *cdcl$_W$-M-level-inv S*
shows *init-clss S = init-clss S′*
using *assms*
apply (*induct rule*: *rtranclp-induct*, *simp*)
using *cdcl$_W$-stgy-no-more-init-clss* **by** (*simp add*: *rtranclp-cdcl$_W$-stgy-consistent-inv*)

lemma *cdcl$_W$-cp-dropWhile-trail′*:
assumes *cdcl$_W$-cp S S′*
obtains *M* **where** *trail S′ = M @ trail S* **and** (∀ *l* ∈ *set M*. ¬*is-decided l*)
using *assms* **by** *induction* (*fastforce elim*: *conflictE propagateE*)+

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail′*:
assumes *cdcl$_W$-cp** S S′*
obtains *M* :: (′*v*, ′*v clause*) *ann-lits* **where**
trail S′ = M @ trail S **and** ∀ *l* ∈ *set M*. ¬*is-decided l*
using *assms* **by** *induction* (*fastforce dest*!: *cdcl$_W$-cp-dropWhile-trail′*)+

lemma *cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp S S′*
shows ∃ *M*. *trail S′ = M @ trail S* ∧ (∀ *l* ∈ *set M*. ¬*is-decided l*)
using *assms* **by** *induction* (*fastforce elim*: *conflictE propagateE*)+

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp** S S′*
shows ∃ *M*. *trail S′ = M @ trail S* ∧ (∀ *l* ∈ *set M*. ¬*is-decided l*)
using *assms* **by** *induction* (*fastforce dest*: *cdcl$_W$-cp-dropWhile-trail*)+

This theorem can be seen a a termination theorem for *cdcl$_W$-cp*.

lemma *length-model-le-vars*:
assumes
no-strange-atm S **and**
no-d: *no-dup* (*trail S*) **and**
finite (*atms-of-mm* (*init-clss S*))
shows *length* (*trail S*) ≤ *card* (*atms-of-mm* (*init-clss S*))
proof −
obtain *M N U k D* **where** *S*: *state S* = (*M*, *N*, *U*, *k*, *D*) **by** (*cases state S*, *auto*)
have *finite* (*atm-of* ‘ *lits-of-l* (*trail S*))
using *assms*(*1*,*3*) **unfolding** *S* **by** (*auto simp add*: *finite-subset*)
have *length* (*trail S*) = *card* (*atm-of* ‘ *lits-of-l* (*trail S*))
using *no-dup-length-eq-card-atm-of-lits-of-l no-d* **by** *blast*
then show *?thesis* **using** *assms*(*1*) **unfolding** *no-strange-atm-def*
by (*auto simp add*: *assms*(*3*) *card-mono*)
qed

lemma *cdcl$_W$-cp-decreasing-measure*:
assumes
cdcl$_W$: *cdcl$_W$-cp S T* **and**
M-lev: *cdcl$_W$-M-level-inv S* **and**
alien: *no-strange-atm S*

```
  shows (λS. card (atms-of-mm (init-clss S)) − length (trail S)
      + (if conflicting S = None then 1 else 0)) S
    > (λS. card (atms-of-mm (init-clss S)) − length (trail S)
      + (if conflicting S = None then 1 else 0)) T
  using assms
proof −
  have length (trail T) ≤ card (atms-of-mm (init-clss T))
    apply (rule length-model-le-vars)
       using cdcl_W-no-strange-atm-inv alien M-lev apply (meson cdcl_W cdcl_W.simps cdcl_W-cp.cases)
      using M-lev cdcl_W cdcl_W-cp-consistent-inv cdcl_W-M-level-inv-def apply blast
      using cdcl_W by (auto simp: cdcl_W-cp.simps)
  with assms
  show ?thesis by induction (auto elim!: conflictE propagateE
    simp del: state-simp simp: state-eq-def)+
qed

lemma cdcl_W-cp-wf: wf {(b, a). (cdcl_W-M-level-inv a ∧ no-strange-atm a) ∧ cdcl_W-cp a b}
  apply (rule wf-wf-if-measure'[of less-than - -
      (λS. card (atms-of-mm (init-clss S)) − length (trail S)
        + (if conflicting S = None then 1 else 0))])
    apply simp
  using cdcl_W-cp-decreasing-measure unfolding less-than-iff by blast

lemma rtranclp-cdcl_W-all-struct-inv-cdcl_W-cp-iff-rtranclp-cdcl_W-cp:
  assumes
    lev: cdcl_W-M-level-inv S and
    alien: no-strange-atm S
  shows (λa b. (cdcl_W-M-level-inv a ∧ no-strange-atm a) ∧ cdcl_W-cp a b)** S T
    ⟷ cdcl_W-cp** S T
  (is ?I S T ⟷ ?C S T)
proof
  assume
    ?I S T
  then show ?C S T by induction auto
next
  assume
    ?C S T
  then show ?I S T
    proof induction
      case base
      then show ?case by simp
    next
      case (step T U) note st = this(1) and cp = this(2) and IH = this(3)
      have cdcl_W** S T
        by (metis rtranclp-unfold cdcl_W-cp-conflicting-not-empty cp st
          rtranclp-propagate-is-rtranclp-cdcl_W tranclp-cdcl_W-cp-propagate-with-conflict-or-not)
      then have
        cdcl_W-M-level-inv T and
        no-strange-atm T
         using ‹cdcl_W** S T› apply (simp add: assms(1) rtranclp-cdcl_W-consistent-inv)
        using ‹cdcl_W** S T› alien rtranclp-cdcl_W-no-strange-atm-inv lev by blast
      then have (λa b. (cdcl_W-M-level-inv a ∧ no-strange-atm a) ∧ cdcl_W-cp a b)** T U
        using cp by auto
      then show ?case using IH by auto
    qed
qed
```

lemma $cdcl_W$-*cp-normalized-element*:
  assumes
    lev: $cdcl_W$-*M-level-inv S* **and**
    no-strange-atm S
  obtains *T* **where** *full* $cdcl_W$-*cp S T*
proof –
  let *?inv* = λ*a*. ($cdcl_W$-*M-level-inv a* ∧ *no-strange-atm a*)
  obtain *T* **where** *T*: *full* (λ*a b*. *?inv a* ∧ $cdcl_W$-*cp a b*) *S T*
    using $cdcl_W$-*cp-wf wf-exists-normal-form*[*of* λ*a b*. *?inv a* ∧ $cdcl_W$-*cp a b*]
    unfolding *full-def* **by** *blast*
    then have $cdcl_W$-cp^{**} *S T*
      using *rtranclp-*$cdcl_W$-*all-struct-inv-*$cdcl_W$-*cp-iff-rtranclp-*$cdcl_W$-*cp assms* **unfolding** *full-def*
      by *blast*
    moreover
      then have $cdcl_W{}^{**}$ *S T*
        using *rtranclp-*$cdcl_W$-*cp-rtranclp-*$cdcl_W$ **by** *blast*
      then have
        $cdcl_W$-*M-level-inv T* **and**
        no-strange-atm T
         using ⟨$cdcl_W{}^{**}$ *S T*⟩ **apply** (*simp add*: *assms*(*1*) *rtranclp-*$cdcl_W$-*consistent-inv*)
        using ⟨$cdcl_W{}^{**}$ *S T*⟩ *assms*(*2*) *rtranclp-*$cdcl_W$-*no-strange-atm-inv lev* **by** *blast*
      then have *no-step* $cdcl_W$-*cp T*
        using *T* **unfolding** *full-def* **by** *auto*
    ultimately show *thesis* **using** *that* **unfolding** *full-def* **by** *blast*
qed

lemma *always-exists-full-*$cdcl_W$-*cp-step*:
  assumes *no-strange-atm S*
  shows ∃ *S''*. *full* $cdcl_W$-*cp S S''*
  using *assms*
proof (*induct card* (*atms-of-mm* (*init-clss S*) – *atm-of* '*lits-of-l* (*trail S*)) *arbitrary*: *S*)
  case *0* **note** *card* = *this*(*1*) **and** *alien* = *this*(*2*)
  then have *atm*: *atms-of-mm* (*init-clss S*) = *atm-of* ' *lits-of-l* (*trail S*)
    unfolding *no-strange-atm-def* **by** *auto*
  { assume *a*: ∃ *S'*. *conflict S S'*
    then obtain *S'* **where** *S'*: *conflict S S'* **by** *metis*
    then have ∀ *S''*. ¬$cdcl_W$-*cp S' S''*
      by (*auto simp*: $cdcl_W$-*cp.simps elim*!: *conflictE propagateE*
        simp del: *state-simp simp*: *state-eq-def*)
    then have *?case* **using** *a S'* $cdcl_W$-*cp.conflict'* **unfolding** *full-def* **by** *blast*
  }
  moreover {
    assume *a*: ∃ *S'*. *propagate S S'*
    then obtain *S'* **where** *propagate S S'* **by** *blast*
    then obtain *E L* **where**
      S: *conflicting S* = *None* **and**
      E: *E* ∈# *clauses S* **and**
      LE: *L* ∈# *E* **and**
      tr: *trail S* ⊨*as CNot* (*E* – {#*L*#}) **and**
      undef: *undefined-lit* (*trail S*) *L* **and**
      S': *S'* ∼ *cons-trail* (*Propagated L E*) *S*
      by (*elim propagateE*) *simp*
    have *atms-of-mm* (*learned-clss S*) ⊆ *atms-of-mm* (*init-clss S*)
      using *alien S* **unfolding** *no-strange-atm-def* **by** *auto*
    then have *atm-of L* ∈ *atms-of-mm* (*init-clss S*)

using *E LE S undef* **unfolding** *clauses-def* **by** (*force simp*: *in-implies-atm-of-on-atms-of-ms*)
then have *False* **using** *undef S* **unfolding** *atm* **unfolding** *lits-of-def*
by (*auto simp add*: *defined-lit-map*)
}
ultimately show *?case* **unfolding** *full-def* **by** (*metis* $cdcl_W$*-cp.cases rtranclp.rtrancl-refl*)
next
case (*Suc n*) **note** *IH* = *this*(*1*) **and** *card* = *this*(*2*) **and** *alien* = *this*(*3*)
{ assume *a*: $\exists S'$. *conflict S S'*
then obtain S' **where** S': *conflict S S'* **by** *metis*
then have $\forall S''$. $\neg cdcl_W$*-cp S' S''*
by (*auto simp*: $cdcl_W$*-cp.simps elim*!: *conflictE propagateE*
simp del: *state-simp simp*: *state-eq-def*)
then have *?case* **unfolding** *full-def Ex-def* **using** S' $cdcl_W$*-cp.conflict'* **by** *blast*
}
moreover {
assume *a*: $\exists S'$. *propagate S S'*
then obtain S' **where** *propagate*: *propagate S S'* **by** *blast*
then obtain *E L* **where**
S: *conflicting S* = *None* **and**
E: *E* ∈# *clauses S* **and**
LE: *L* ∈# *E* **and**
tr: *trail S* ⊨*as CNot* (*E* − {#*L*#}) **and**
undef: *undefined-lit* (*trail S*) *L* **and**
S': S' ∼ *cons-trail* (*Propagated L E*) *S*
by (*elim propagateE*) *simp*
then have *atm-of L* ∉ *atm-of* ‘ *lits-of-l* (*trail S*)
unfolding *lits-of-def* **by** (*auto simp add*: *defined-lit-map*)
moreover
have *no-strange-atm S'* **using** *alien propagate propagate-no-strange-atm-inv* **by** *blast*
then have *atm-of L* ∈ *atms-of-mm* (*init-clss S*)
using S' *LE E undef* **unfolding** *no-strange-atm-def*
by (*auto simp*: *clauses-def in-implies-atm-of-on-atms-of-ms*)
then have $\bigwedge A$. {*atm-of L*} ⊆ *atms-of-mm* (*init-clss S*) − *A* ∨ *atm-of L* ∈ *A* **by** *force*
moreover have *Suc n* − *card* {*atm-of L*} = *n* **by** *simp*
moreover have *card* (*atms-of-mm* (*init-clss S*) − *atm-of* ‘ *lits-of-l* (*trail S*)) = *Suc n*
using *card S S'* **by** *simp*
ultimately
have *card* (*atms-of-mm* (*init-clss S*) − *atm-of* ‘ *insert L* (*lits-of-l* (*trail S*))) = *n*
by (*metis* (*no-types*) *Diff-insert card-Diff-subset finite.emptyI finite.insertI image-insert*)
then have *n* = *card* (*atms-of-mm* (*init-clss S'*) − *atm-of* ‘ *lits-of-l* (*trail S'*))
using *card S S' undef* **by** *simp*
then have *a1*: *Ex* (*full* $cdcl_W$*-cp S'*) **using** *IH* ‹*no-strange-atm S'*› **by** *blast*
have *?case*
proof −
obtain S'' :: *'st* **where**
ff1: $cdcl_W$*-cp*** *S' S''* ∧ *no-step* $cdcl_W$*-cp S''*
using *a1* **unfolding** *full-def* **by** *blast*
have $cdcl_W$*-cp*** *S S''*
using *ff1* $cdcl_W$*-cp.intros*(*2*)[*OF propagate*]
by (*metis* (*no-types*) *converse-rtranclp-into-rtranclp*)
then have $\exists S''$. $cdcl_W$*-cp*** *S S''* ∧ ($\forall S'''$. ¬ $cdcl_W$*-cp S'' S'''*)
using *ff1* **by** *blast*
then show *?thesis* **unfolding** *full-def*
by *meson*
qed
}

ultimately show *?case* **unfolding** *full-def* **by** (*metis* $cdcl_W$*-cp.cases rtranclp.rtrancl-refl*)
qed

Literal of highest level in conflicting clauses

One important property of the $cdcl_W$ with strategy is that, whenever a conflict takes place, there is at least a literal of level k involved (except if we have derived the false clause). The reason is that we apply conflicts before a decision is taken.

abbreviation *no-clause-is-false* :: *'st* ⇒ *bool* **where**
no-clause-is-false ≡
  λ*S*. (*conflicting S* = *None* ⟶ (∀ *D* ∈# *clauses S*. ¬*trail S* ⊨*as CNot D*))

abbreviation *conflict-is-false-with-level* :: *'st* ⇒ *bool* **where**
conflict-is-false-with-level S ≡ ∀ *D*. *conflicting S* = *Some D* ⟶ *D* ≠ {#}
  ⟶ (∃ *L* ∈# *D*. *get-level* (*trail S*) *L* = *backtrack-lvl S*)

lemma *not-conflict-not-any-negated-init-clss*:
  assumes ∀ *S'*. ¬*conflict S S'*
  shows *no-clause-is-false S*
proof (*clarify*)
  fix *D*
  assume *D* ∈# *local.clauses S* **and** *conflicting S* = *None* **and** *trail S* ⊨*as CNot D*
  then show *False*
    using *conflict-rule*[*of S D update-conflicting* (*Some D*) *S*] *assms*
    by *auto*
qed

lemma *full-*$cdcl_W$*-cp-not-any-negated-init-clss*:
  assumes *full* $cdcl_W$*-cp S S'*
  shows *no-clause-is-false S'*
  using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full-def* **by** *auto*

lemma *full1-*$cdcl_W$*-cp-not-any-negated-init-clss*:
  assumes *full1* $cdcl_W$*-cp S S'*
  shows *no-clause-is-false S'*
  using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full1-def* **by** *auto*

lemma $cdcl_W$*-stgy-not-non-negated-init-clss*:
  assumes $cdcl_W$*-stgy S S'*
  shows *no-clause-is-false S'*
  using *assms* **apply** (*induct rule*: $cdcl_W$*-stgy.induct*)
  using *full1-*$cdcl_W$*-cp-not-any-negated-init-clss full-*$cdcl_W$*-cp-not-any-negated-init-clss* **by** *metis+*

lemma *rtranclp-*$cdcl_W$*-stgy-not-non-negated-init-clss*:
  assumes $cdcl_W$*-stgy*** *S S'* **and** *no-clause-is-false S*
  shows *no-clause-is-false S'*
  using *assms* **by** (*induct rule*: *rtranclp-induct*) (*auto simp*: $cdcl_W$*-stgy-not-non-negated-init-clss*)

lemma $cdcl_W$*-stgy-conflict-ex-lit-of-max-level*:
  assumes
    $cdcl_W$*-cp S S'* **and**
    no-clause-is-false S **and**
    $cdcl_W$*-M-level-inv S*
  shows *conflict-is-false-with-level S'*
  using *assms*

proof (*induct rule*: *cdcl*$_W$*-cp.induct*)
  case *conflict'*
  then show *?case* **by** (*auto elim*: *conflictE*)
next
  case *propagate'*
  then show *?case* **by** (*auto elim*: *propagateE*)
qed

lemma *no-chained-conflict*:
  assumes *conflict S S'* **and** *conflict S' S''*
  shows *False*
  using *assms* **unfolding** *conflict.simps*
  by (*metis conflicting-update-conflicting option.distinct*(*1*) *state-eq-conflicting*)

lemma *rtranclp-cdcl*$_W$*-cp-propa-or-propa-confl*:
  assumes *cdcl*$_W$*-cp*** *S U*
  shows *propagate*** *S U* ∨ (∃ *T*. *propagate*** *S T* ∧ *conflict T U*)
  using *assms*
proof *induction*
  case *base*
  then show *?case* **by** *auto*
next
  case (*step U V*) **note** *SU* = *this*(*1*) **and** *UV* = *this*(*2*) **and** *IH* = *this*(*3*)
  consider (*confl*) *T* **where** *propagate*** *S T* **and** *conflict T U*
    | (*propa*) *propagate*** *S U* **using** *IH* **by** *auto*
  then show *?case*
    proof *cases*
      case *confl*
      then have *False* **using** *UV* **by** (*auto elim*: *conflictE*)
      then show *?thesis* **by** *fast*
    next
      case *propa*
      also have *conflict U V* ∨ *propagate U V* **using** *UV* **by** (*auto simp add*: *cdcl*$_W$*-cp.simps*)
      ultimately show *?thesis* **by** *force*
    qed
qed

lemma *rtranclp-cdcl*$_W$*-co-conflict-ex-lit-of-max-level*:
  assumes *full*: *full cdcl*$_W$*-cp S U*
  and *cls-f*: *no-clause-is-false S*
  and *conflict-is-false-with-level S*
  and *lev*: *cdcl*$_W$*-M-level-inv S*
  shows *conflict-is-false-with-level U*
proof (*intro allI impI*)
  fix *D*
  assume
    confl: *conflicting U* = *Some D* **and**
    D: *D* ≠ {#}
  consider (*CT*) *conflicting S* = *None* | (*SD*) *D'* **where** *conflicting S* = *Some D'*
    by (*cases conflicting S*) *auto*
  then show ∃*L*∈#*D*. *get-level* (*trail U*) *L* = *backtrack-lvl U*
    proof *cases*
      case *SD*
      then have *S* = *U*
        by (*metis* (*no-types*) *assms*(*1*) *cdcl*$_W$*-cp-conflicting-not-empty full-def rtranclpD tranclpD*)
      then show *?thesis* **using** *assms*(*3*) *confl D* **by** *blast*−

next
  case *CT*
  have *init-clss U* = *init-clss S* **and** *learned-clss U* = *learned-clss S*
    using *full* **unfolding** *full-def*
      apply (*metis* (*no-types*) *rtranclpD tranclp-cdcl$_W$-cp-no-more-init-clss*)
    by (*metis* (*mono-tags*, *lifting*) *full full-def rtranclp-cdcl$_W$-cp-learned-clause-inv*)
  obtain *T* **where** *propagate*** *S T* **and** *TU*: *conflict T U*
    proof −
      have *f5*: $U \neq S$
        using *confl CT* **by** *force*
      then have *cdcl$_W$-cp*$^{++}$ *S U*
        by (*metis full full-def rtranclpD*)
      have $\bigwedge$*p pa*. ¬ *propagate p pa* ∨ *conflicting pa* =
        (*None* :: '*v clause option*)
        by (*auto elim*: *propagateE*)
      then show *?thesis*
        using *f5 that tranclp-cdcl$_W$-cp-propagate-with-conflict-or-not*[*OF* ‹*cdcl$_W$-cp*$^{++}$ *S U*›]
        full confl CT **unfolding** *full-def* **by** *auto*
    qed
  obtain *D'* **where**
    conflicting T = *None* **and**
    D': *D'* ∈# *clauses T* **and**
    tr: *trail T* ⊨*as CNot* (*D'*) **and**
    U: *U* ∼ *update-conflicting* (*Some* (*D'*)) *T*
    using *TU* **by** (*auto elim*!: *conflictE*)
  have *init-clss T* = *init-clss S* **and** *learned-clss T* = *learned-clss S*
    using *U* ‹*init-clss U* = *init-clss S*› ‹*learned-clss U* = *learned-clss S*› **by** *auto*
  then have *D* ∈# *clauses S*
    using *confl U D'* **by** (*auto simp*: *clauses-def*)
  then have ¬ *trail S* ⊨*as CNot D*
    using *cls-f CT* **by** *simp*

  moreover
    obtain *M* **where** *tr-U*: *trail U* = *M* @ *trail S* **and** *nm*: ∀ *m*∈*set M*. ¬*is-decided m*
      by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *full-def rtranclp-cdcl$_W$-cp-dropWhile-trail*)
    have *trail U* ⊨*as CNot D*
      using *tr confl U* **by** (*auto elim*!: *conflictE*)
  ultimately obtain *L* **where** *L* ∈# *D* **and** −*L* ∈ *lits-of-l M*
    unfolding *tr-U CNot-def true-annots-def Ball-def true-annot-def true-cls-def* **by** *force*

  moreover have *inv-U*: *cdcl$_W$-M-level-inv U*
    by (*metis cdcl$_W$-stgy.conflict' cdcl$_W$-stgy-consistent-inv full full-unfold lev*)
  moreover
    have *backtrack-lvl U* = *backtrack-lvl S*
      using *full* **unfolding** *full-def* **by** (*auto dest*: *rtranclp-cdcl$_W$-cp-backtrack-lvl*)

  moreover
    have *no-dup* (*trail U*)
      using *inv-U* **unfolding** *cdcl$_W$-M-level-inv-def* **by** *auto*
    { fix *x* :: ('*v*, '*v clause*) *ann-lit* **and**
        xb :: ('*v*, '*v clause*) *ann-lit*
      assume *a1*: *atm-of L* = *atm-of* (*lit-of xb*)
      moreover assume *a2*: − *L* = *lit-of x*
      moreover assume *a3*: (λ*l*. *atm-of* (*lit-of l*)) ‘ *set M*
        ∩ (λ*l*. *atm-of* (*lit-of l*)) ‘ *set* (*trail S*) = {}
      moreover assume *a4*: *x* ∈ *set M*

moreover assume *a5*: *xb* ∈ *set* (*trail S*)
moreover have *atm-of* (− *L*) = *atm-of L*
by *auto*
ultimately have *False*
by *auto*
}
then have *LS*: *atm-of L* ∉ *atm-of* ‘ *lits-of-l* (*trail S*)
using ‹−*L* ∈ *lits-of-l M*› ‹*no-dup* (*trail U*)› **unfolding** *tr-U lits-of-def* **by** *auto*
ultimately have *get-level* (*trail U*) *L* = *backtrack-lvl U*
proof (*cases count-decided* (*trail S*) ≠ *0*, *goal-cases*)
case *2* **note** *LD* = *this*(*1*) **and** *LM* = *this*(*2*) **and** *inv-U* = *this*(*3*) **and** *US* = *this*(*4*) **and** *LS* = *this*(*5*) **and** *ne* = *this*(*6*)
have *backtrack-lvl S* = *0*
using *lev ne* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
moreover have *get-level M L* = *0*
using *nm* **by** *auto*
ultimately show *?thesis* **using** *LS ne US* **unfolding** *tr-U*
by (*simp add*: *lits-of-def filter-empty-conv*)
next
case *1* **note** *LD* = *this*(*1*) **and** *LM* = *this*(*2*) **and** *inv-U* = *this*(*3*) **and** *US* = *this*(*4*) **and** *LS* = *this*(*5*) **and** *ne* = *this*(*6*)

have *count-decided* (*trail S*) = *backtrack-lvl S*
using *ne lev* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
moreover have *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*
using ‹−*L* ∈ *lits-of-l M*› **by** (*simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def*)
ultimately show *?thesis*
using *nm ne get-level-skip-in-all-not-decided*[*of M L*] **unfolding** *lits-of-def US tr-U*
by *auto*
qed
then show ∃ *L*∈#*D*. *get-level* (*trail U*) *L* = *backtrack-lvl U*
using ‹*L* ∈# *D*› **by** *blast*
qed
qed

Literal of highest level in decided literals

definition *mark-is-false-with-level* :: *'st* ⇒ *bool* **where**
mark-is-false-with-level S' ≡
∀ *D M1 M2 L*. *M1* @ *Propagated L D* # *M2* = *trail S'* ⟶ *D* − {#*L*#} ≠ {#}
⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S'*) *L* = *count-decided M1*)

definition *no-more-propagation-to-do* :: *'st* ⇒ *bool* **where**
no-more-propagation-to-do S ≡
∀ *D M M' L*. *D* + {#*L*#} ∈# *clauses S* ⟶ *trail S* = *M'* @ *M* ⟶ *M* ⊨*as CNot D*
⟶ *undefined-lit M L* ⟶ *count-decided M* < *backtrack-lvl S*
⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S*) *L* = *count-decided M*)

lemma *propagate-no-more-propagation-to-do*:
assumes *propagate*: *propagate S S'*
and *H*: *no-more-propagation-to-do S*
and *lev-inv*: $cdcl_W$*-M-level-inv S*
shows *no-more-propagation-to-do S'*
using *assms*
proof −

obtain *E L* **where**
  S: *conflicting S = None* **and**
  E: *E* $\in\#$ *clauses S* **and**
  LE: *L* $\in\#$ *E* **and**
  tr: *trail S* $\models$*as CNot* (*E* $-$ {#*L*#}) **and**
  undefL: *undefined-lit* (*trail S*) *L* **and**
  S': *S'* $\sim$ *cons-trail* (*Propagated L E*) *S*
  using *propagate* **by** (*elim propagateE*) *simp*
let *?M'* = *Propagated L E* # *trail S*
show *?thesis* **unfolding** *no-more-propagation-to-do-def*
  proof (*intro allI impI*)
    fix *D M1 M2 L'*
    assume
      D-L: *D* + {#*L'*#} $\in\#$ *clauses S'* **and**
      trail S' = *M2* @ *M1* **and**
      get-max: *count-decided M1* < *backtrack-lvl S'* **and**
      M1 $\models$*as CNot D* **and**
      undef: *undefined-lit M1 L'*
    have *tl M2* @ *M1* = *trail S* $\vee$ (*M2* = [] $\wedge$ *M1* = *Propagated L E* # *trail S*)
      using ‹*trail S'* = *M2* @ *M1*› *S' S undefL lev-inv*
      by (*cases M2*) (*auto simp*:$cdcl_W$*-M-level-inv-decomp*)
    moreover {
      assume *tl M2* @ *M1* = *trail S*
      moreover have *D* + {#*L'*#} $\in\#$ *clauses S*
        using *D-L S S' undefL* **unfolding** *clauses-def* **by** *auto*
      moreover have *count-decided M1* < *backtrack-lvl S*
        using *get-max S S' undefL* **by** *auto*
      ultimately obtain *L'* **where** *L'* $\in\#$ *D* **and**
        get-level (*trail S*) *L'* = *count-decided M1*
        using *H* ‹*M1* $\models$*as CNot D*› *undef* **unfolding** *no-more-propagation-to-do-def* **by** *metis*
      moreover
        { **have** $cdcl_W$*-M-level-inv S'*
            using $cdcl_W$*-consistent-inv lev-inv* $cdcl_W$*.propagate*[*OF propagate*] **by** *blast*
          then have *no-dup ?M'* **using** *S' undefL* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
          moreover
            have *atm-of L'* $\in$ *atm-of* ‘ (*lits-of-l M1*)
              using ‹*L'* $\in\#$ *D*› ‹*M1* $\models$*as CNot D*› **by** (*metis atm-of-uminus image-eqI*
                in-CNot-implies-uminus(*2*))
            then have *atm-of L'* $\in$ *atm-of* ‘ (*lits-of-l* (*trail S*))
              using ‹*tl M2* @ *M1* = *trail S*›[*symmetric*] *S undefL* **by** *auto*
          ultimately have *atm-of L* $\neq$ *atm-of L'* **unfolding** *lits-of-def* **by** *auto*
      }
      ultimately have $\exists$ *L'* $\in\#$ *D*. *get-level* (*trail S'*) *L'* = *count-decided M1*
        using *S S' undefL* **by** *auto*
    }
    moreover {
      assume *M2* = [] **and** *M1*: *M1* = *Propagated L E* # *trail S*
      have $cdcl_W$*-M-level-inv S'*
        using $cdcl_W$*-consistent-inv*[*OF - lev-inv*] $cdcl_W$*.propagate*[*OF propagate*] **by** *blast*
      then have *count-decided M1* = *backtrack-lvl S'*
        using *S' M1 undefL* **unfolding** $cdcl_W$*-M-level-inv-def* **by** (*auto intro*: *Max-eqI*)
      then have *False* **using** *get-max* **by** *auto*
    }
    ultimately show $\exists$ *L*. *L* $\in\#$ *D* $\wedge$ *get-level* (*trail S'*) *L* = *count-decided M1*
      by *fast*
  qed

qed

lemma *conflict-no-more-propagation-to-do*:
  assumes
    conflict: *conflict S S'* **and**
    H: *no-more-propagation-to-do S* **and**
    M: $cdcl_W$*-M-level-inv S*
  shows *no-more-propagation-to-do S'*
  using *assms* **unfolding** *no-more-propagation-to-do-def* **by** (*force elim*!: *conflictE*)

lemma $cdcl_W$*-cp-no-more-propagation-to-do*:
  assumes
    conflict: $cdcl_W$*-cp S S'* **and**
    H: *no-more-propagation-to-do S* **and**
    M: $cdcl_W$*-M-level-inv S*
  shows *no-more-propagation-to-do S'*
  using *assms*
  proof (*induct rule*: $cdcl_W$*-cp.induct*)
  case (*conflict' S S'*)
  then show *?case* **using** *conflict-no-more-propagation-to-do*[*of S S'*] **by** *blast*
next
  case (*propagate' S S'*) **note** *S = this*
  show *1*: *no-more-propagation-to-do S'*
    using *propagate-no-more-propagation-to-do*[*of S S'*] *S* **by** *blast*
qed

lemma $cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step*:
  assumes
    o: $cdcl_W$*-o S S'* **and**
    alien: *no-strange-atm S* **and**
    lev: $cdcl_W$*-M-level-inv S*
  shows $\exists S'$. $cdcl_W$*-stgy S S'*
proof –
  obtain *S''* **where** *full* $cdcl_W$*-cp S' S''*
    using *always-exists-full-*$cdcl_W$*-cp-step alien* $cdcl_W$*-no-strange-atm-inv* $cdcl_W$*-o-no-more-init-clss*
     o other lev **by** (*meson* $cdcl_W$*-consistent-inv*)
  then show *?thesis*
    using *assms* **by** (*metis always-exists-full-*$cdcl_W$*-cp-step* $cdcl_W$*-stgy.conflict' full-unfold other'*)
qed

lemma *backtrack-no-decomp*:
  assumes
    S: *conflicting S = Some E* **and**
    LE: *L* ∈# *E* **and**
    L: *get-level* (*trail S*) *L = backtrack-lvl S* **and**
    D: *get-maximum-level* (*trail S*) (*remove1-mset L E*) < *backtrack-lvl S* **and**
    bt: *backtrack-lvl S = get-maximum-level* (*trail S*) *E* **and**
    M-L: $cdcl_W$*-M-level-inv S*
  shows $\exists S'$. $cdcl_W$*-o S S'*
proof –
  have *L-D*: *get-level* (*trail S*) *L = get-maximum-level* (*trail S*) *E*
    using *L D bt* **by** (*simp add*: *get-maximum-level-plus*)
  let *?i = get-maximum-level* (*trail S*) (*remove1-mset L E*)
  obtain *K M1 M2* **where**
    K: (*Decided K # M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
    lev-K: *get-level* (*trail S*) *K = Suc ?i*

    using *backtrack-ex-decomp*[*OF M-L*, *of ?i*] *D S* **by** *auto*
  show *?thesis* **using** *backtrack-rule*[*OF S LE K L*, *of ?i*] *bt L lev-K bj* **by** (*auto simp*: $cdcl_W$*-bj.simps*)
qed

lemma $cdcl_W$*-stgy-final-state-conclusive*:
  assumes
    termi: $\forall S'. \neg$ $cdcl_W$*-stgy* $S\ S'$ **and**
    decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**
    learned: $cdcl_W$*-learned-clause S* **and**
    level-inv: $cdcl_W$*-M-level-inv S* **and**
    alien: *no-strange-atm S* **and**
    no-dup: *distinct-*$cdcl_W$*-state S* **and**
    confl: $cdcl_W$*-conflicting S* **and**
    confl-k: *conflict-is-false-with-level S*
  shows (*conflicting S* = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss S*)))
      $\vee$ (*conflicting S* = *None* $\wedge$ *trail S* $\models$*as set-mset* (*init-clss S*))
proof −
  let *?M* = *trail S*
  let *?N* = *init-clss S*
  let *?k* = *backtrack-lvl S*
  let *?U* = *learned-clss S*
  consider
      (*None*) *conflicting S* = *None*
    | (*Some-Empty*) *E* **where** *conflicting S* = *Some E* **and** *E* = {#}
    | (*Some*) E' **where** *conflicting S* = *Some* E' **and**
      conflicting S = *Some* (E') **and** $E' \neq$ {#}
    by (*cases conflicting S*, *simp*) *auto*
  then show *?thesis*
    proof *cases*
      case (*Some-Empty E*)
      then have *conflicting S* = *Some* {#} **by** *auto*
      then have *unsatisfiable* (*set-mset* (*init-clss S*))
        using *assms*(*3*) **unfolding** $cdcl_W$*-learned-clause-def true-clss-cls-def*
        by (*metis* (*no-types*, *lifting*) *Un-insert-right atms-of-empty satisfiable-def*
          sup-bot.right-neutral total-over-m-insert total-over-set-empty true-cls-empty)
      then show *?thesis* **using** *Some-Empty* **by** *auto*
    next
      case *None*
      { **assume** ¬*?M* $\models$*asm ?N*
        have *atm-of* ‘ (*lits-of-l ?M*) = *atms-of-mm ?N* (**is** *?A* = *?B*)
          proof
            show *?A* $\subseteq$ *?B* **using** *alien* **unfolding** *no-strange-atm-def* **by** *auto*
            show *?B* $\subseteq$ *?A*
              proof (*rule ccontr*)
                assume ¬*?B* $\subseteq$ *?A*
                then obtain *l* **where** $l \in$ *?B* **and** $l \notin$ *?A* **by** *auto*
                then have *undefined-lit ?M* (*Pos l*)
                  using ‹$l \notin$ *?A*› **unfolding** *lits-of-def* **by** (*auto simp add*: *defined-lit-map*)
                moreover have *conflicting S* = *None*
                  using *None* **by** *auto*
                ultimately have $\exists S'.$ $cdcl_W$*-o* $S\ S'$
                  using $cdcl_W$*-o.decide decide-rule* ‹$l \in$ *?B*› *no-strange-atm-def*
                  by (*metis literal.sel*(*1*) *state-eq-def*)
                then show *False*
                  using *termi* $cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step*[*OF - alien*] *level-inv* **by** *blast*
              qed

qed
obtain D **where** $\neg$ *?M* $\models$*a* D **and** $D \in\#$ *?N*
using ‹$\neg$*?M* $\models$*asm ?N*› **unfolding** *lits-of-def true-annots-def Ball-def* **by** *auto*
have *atms-of* $D \subseteq$ *atm-of* ‘ (*lits-of-l ?M*)
using ‹$D \in\#$ *?N*› **unfolding** ‹*atm-of* ‘ (*lits-of-l ?M*) = *atms-of-mm ?N*› *atms-of-ms-def*
by (*auto simp add*: *atms-of-def*)
then have *a1*: *atm-of* ‘ *set-mset* $D \subseteq$ *atm-of* ‘ *lits-of-l* (*trail S*)
by (*auto simp add*: *atms-of-def lits-of-def*)
have *total-over-m* (*lits-of-l ?M*) {D}
using ‹*atms-of* $D \subseteq$ *atm-of* ‘ (*lits-of-l ?M*)›
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set **by** (*fastforce simp*: *total-over-set-def*)
then have *?M* $\models$*as CNot D*
using *total-not-true-cls-true-clss-CNot* ‹$\neg$ *trail S* $\models$*a D*› *true-annot-def*
true-annots-true-cls **by** *fastforce*
then have *False*
proof −
obtain S' **where**
f2: *full* $cdcl_W$*-cp* S S'
by (*meson alien always-exists-full-*$cdcl_W$*-cp-step level-inv*)
then have $S' = S$
using $cdcl_W$*-stgy.conflict'*[*of S*] **by** (*metis* (*no-types*) *full-unfold termi*)
then show *?thesis*
using *f2* ‹$D \in\#$ *init-clss S*› *None* ‹*trail S* $\models$*as CNot D*›
clauses-def full-$cdcl_W$*-cp-not-any-negated-init-clss* **by** *auto*
qed
}
then have *?M* $\models$*asm ?N* **by** *blast*
then show *?thesis*
using *None* **by** *auto*
next
case (*Some E'*) **note** *conf* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *nempty* = *this*(*3*)
then obtain L D **where**
E'[*simp*]: $E' = D + \{\#L\#\}$ **and**
lev-L: *get-level ?M L* = *?k*
by (*metis* (*mono-tags*) *confl-k insert-DiffM2*)
let *?D* = $D + \{\#L\#\}$
have *?D* $\neq \{\#\}$ **by** *auto*
have *?M* $\models$*as CNot ?D* **using** *confl LD* **unfolding** $cdcl_W$*-conflicting-def* **by** *auto*
then have *?M* $\neq$ [] **unfolding** *true-annots-def Ball-def true-annot-def true-cls-def* **by** *force*
have *M*: *?M* = *hd ?M* # *tl ?M* **using** ‹*?M* $\neq$ []› *list.collapse* **by** *fastforce*

have *g-k*: *get-maximum-level* (*trail S*) $D \leq$ *?k*
using *count-decided-ge-get-maximum-level*[*of ?M*] *level-inv*
unfolding $cdcl_W$*-M-level-inv-def*
by *auto*
{
assume *decided*: *is-decided* (*hd ?M*)
then obtain k' **where** k': $k' + 1$ = *?k*
using *level-inv M* **unfolding** $cdcl_W$*-M-level-inv-def*
by (*cases hd* (*trail S*); *cases trail S*) *auto*
obtain L' **where** L': *hd ?M* = *Decided* L' **using** *decided* **by** (*cases hd ?M*) *auto*
have $*$: $\bigwedge$*list. no-dup list* $\Longrightarrow$
$-L \in$ *lits-of-l list* $\Longrightarrow$ *atm-of* $L \in$ *atm-of* ‘ *lits-of-l list*
by (*metis atm-of-uminus imageI*)
have L'*-L*: $L' = -L$
proof (*rule ccontr*)

assume ¬ *?thesis*
moreover have $-L \in$ *lits-of-l ?M* **using** *confl LD* **unfolding** *cdcl*$_W$*-conflicting-def* **by** *auto*
ultimately have *get-level* (*hd* (*trail S*) # *tl* (*trail S*)) *L* = *get-level* (*tl ?M*) *L*
using *cdcl*$_W$*-M-level-inv-decomp*(*1*)[*OF level-inv*] **unfolding** *consistent-interp-def*
by (*subst* (*asm*) (*2*) *M*) (*auto simp add*: *atm-of-eq-atm-of* L')
moreover
have *count-decided* (*trail S*) = *?k*
using *level-inv* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have *count*: *count-decided* (*tl* (*trail S*)) = *?k* − *1*
using *level-inv* **unfolding** *cdcl*$_W$*-M-level-inv-def*
by (*subst* (*asm*) *M*) (*auto simp add*: L')
then have *get-level* (*tl ?M*) *L* < *?k*
using *count-decided-ge-get-level*[*of L tl ?M*] **unfolding** *count* k'[*symmetric*]
by *auto*
finally show *False* **using** *lev-L M* **by** *auto*
qed
have *L*: *hd ?M* = *Decided* ($-L$) **using** L'*-L* L' **by** *auto*

have *get-maximum-level* (*trail S*) *D* < *?k*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *get-maximum-level* (*trail S*) *D* = *?k* **using** *M g-k* **unfolding** *L* **by** *auto*
then obtain L'' **where** $L'' \in\#$ *D* **and** *L-k*: *get-level ?M* L'' = *?k*
using *get-maximum-level-exists-lit*[*of ?k ?M D*] **unfolding** k'[*symmetric*] **by** *auto*
have $L \neq L''$ **using** *no-dup* ‹$L'' \in\#$ *D*›
unfolding *distinct-cdcl*$_W$*-state-def LD*
by (*metis* E' *add.right-neutral add-diff-cancel-right*$'$
distinct-mem-diff-mset union-commute union-single-eq-member)
have $L'' = -L$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *get-level ?M* L'' = *get-level* (*tl ?M*) L''
using *M* ‹$L \neq L''$› *get-level-skip-beginning*[*of* L'' *hd ?M tl ?M*] **unfolding** *L*
by (*auto simp*: *atm-of-eq-atm-of*)
moreover
have *d*: *dropWhile* (λS. *atm-of* (*lit-of S*) ≠ *atm-of L*) (*tl* (*trail S*)) = []
using *level-inv* **unfolding** *cdcl*$_W$*-M-level-inv-def* **apply** (*subst* (*asm*)(*2*) *M*)
by (*auto simp*: *image-iff* L' L'*-L*)
have *get-level* (*tl* (*trail S*)) *L* = *0*
by (*auto simp*: *filter-empty-conv d*)
moreover
have *get-level* (*tl* (*trail S*)) L'' ≤ *count-decided* (*tl* (*trail S*))
by *auto*
then have *get-level* (*tl* (*trail S*)) L'' < *backtrack-lvl S*
using *level-inv* **unfolding** *cdcl*$_W$*-M-level-inv-def* **apply** (*subst* (*asm*)(*5*) *M*)
by (*auto simp*: *image-iff* L' L'*-L simp del*: *count-decided-ge-get-level*)
ultimately show *False*
apply −
apply (*subst* (*asm*) *M*, *subst* (*asm*)(*3*) *M*, *subst* (*asm*) L')
using *L-k*
apply (*auto simp*: L' L'*-L split*: *if-splits*)
apply (*subst* (*asm*)(*3*) *M*, *subst* (*asm*) L')
using ‹$L'' \neq -L$› **by** (*auto simp*: L' L'*-L split*: *if-splits*)
qed
then have *taut*: *tautology* (*D* + {#*L*#})
using ‹$L'' \in\#$ *D*› **by** (*metis add.commute mset-leD mset-le-add-left multi-member-this*

```
        tautology-minus)
      have consistent-interp (lits-of-l ?M)
        using level-inv unfolding cdcl_W-M-level-inv-def by auto
      then have ¬?M ⊨as CNot ?D
        using taut by (metis ‹L'' = − L› ‹L'' ∈# D› add.commute consistent-interp-def
          diff-union-cancelR in-CNot-implies-uminus(2) in-diffD multi-member-this)
      moreover have ?M ⊨as CNot ?D
        using confl no-dup LD unfolding cdcl_W-conflicting-def by auto
      ultimately show False by blast
    qed note H = this
  have get-maximum-level (trail S) D < get-maximum-level (trail S) (D + {#L#})
    using H by (auto simp: get-maximum-level-plus lev-L max-def)
  moreover have backtrack-lvl S = get-maximum-level (trail S) (D + {#L#})
    using H by (auto simp: get-maximum-level-plus lev-L max-def)
  ultimately have False
    using backtrack-no-decomp[OF conf - lev-L] level-inv termi
    cdcl_W-then-exists-cdcl_W-stgy-step[of S] alien unfolding E'
    by (auto simp add: lev-L max-def)
} note not-is-decided = this

moreover {
  let ?D = D + {#L#}
  have ?D ≠ {#} by auto
  have ?M ⊨as CNot ?D using confl LD unfolding cdcl_W-conflicting-def by auto
  then have ?M ≠ [] unfolding true-annots-def Ball-def true-annot-def true-cls-def by force
  assume nm: ¬is-decided (hd ?M)
  then obtain L' C where L'C: hd-trail S = Propagated L' C using ‹trail S ≠ []›
    by (cases hd-trail S) auto
  then have hd ?M = Propagated L' C
    using ‹trail S ≠ []› by fastforce
  then have M: ?M = Propagated L' C # tl ?M
    using ‹?M ≠ []› list.collapse by fastforce
  then obtain C' where C': C = C' + {#L'#}
    using confl unfolding cdcl_W-conflicting-def by (metis append-Nil diff-single-eq-union)
  { assume −L' ∉# ?D
    then have Ex (skip S)
      using skip-rule[OF M conf] unfolding E' by auto
    then have False
      using cdcl_W-then-exists-cdcl_W-stgy-step[of S] alien level-inv termi
      by (auto dest: cdcl_W-o.intros cdcl_W-bj.intros)
  }
  moreover {
    assume L'D: −L' ∈# ?D
    then obtain D' where D': ?D = D' + {#−L'#} by (metis insert-DiffM2)
    then have get-maximum-level (trail S) D' ≤ ?k
      using count-decided-ge-get-maximum-level[of Propagated L' C # tl ?M] M
      level-inv unfolding cdcl_W-M-level-inv-def by auto
    then have get-maximum-level (trail S) D' = ?k
      ∨ get-maximum-level (trail S) D' < ?k
      using le-neq-implies-less by blast
    moreover {
      assume g-D'-k: get-maximum-level (trail S) D' = ?k
      then have f1: get-maximum-level (trail S) D' = backtrack-lvl S
        using M by auto
      then have Ex (cdcl_W-o S)
        using f1 resolve-rule[of S L' C , OF ‹trail S ≠ []› - - conf] conf g-D'-k
```

L'C L'D **unfolding** *C' D' E'*
by (*fastforce simp add*: *D' intro*: $cdcl_W$*-o.intros* $cdcl_W$*-bj.intros*)
then have *False*
by (*meson alien* $cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step termi level-inv*)
}
moreover {
assume *a1*: *get-maximum-level* (*trail S*) *D'* < *?k*
then have *f3*: *get-maximum-level* (*trail S*) *D'* < *get-level* (*trail S*) (−*L'*)
using *a1 lev-L* **by** (*metis D' get-maximum-level-ge-get-level insert-noteq-member not-less*)
moreover have *backtrack-lvl S* = *get-level* (*trail S*) *L'*
apply (*subst M*)
using *level-inv* **unfolding** $cdcl_W$*-M-level-inv-def*
by (*subst* (*asm*)(*3*) *M*) (*auto simp add*: $cdcl_W$*-M-level-inv-decomp*)[]
moreover
then have *get-level* (*trail S*) *L'* = *get-maximum-level* (*trail S*) (*D'* + {#− *L'*#})
using *a1* **by** (*auto simp add*: *get-maximum-level-plus max-def*)
ultimately have *False*
using *M backtrack-no-decomp*[*of S - −L', OF conf*]
$cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step L'D level-inv termi alien*
unfolding *D' E'* **by** *auto*
}
ultimately have *False* **by** *blast*
}
ultimately have *False* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed
qed

lemma $cdcl_W$*-cp-tranclp-*$cdcl_W$:
$cdcl_W$*-cp S S'* ⟹ $cdcl_W^{++}$ *S S'*
apply (*induct rule*: $cdcl_W$*-cp.induct*)
by (*meson* $cdcl_W$*.conflict* $cdcl_W$*.propagate tranclp.r-into-trancl tranclp.trancl-into-trancl*)+

lemma *tranclp-*$cdcl_W$*-cp-tranclp-*$cdcl_W$:
$cdcl_W$*-cp*$^{++}$ *S S'* ⟹ $cdcl_W^{++}$ *S S'*
apply (*induct rule*: *tranclp.induct*)
apply (*simp add*: $cdcl_W$*-cp-tranclp-*$cdcl_W$)
by (*meson* $cdcl_W$*-cp-tranclp-*$cdcl_W$ *tranclp-trans*)

lemma $cdcl_W$*-stgy-tranclp-*$cdcl_W$:
$cdcl_W$*-stgy S S'* ⟹ $cdcl_W^{++}$ *S S'*
proof (*induct rule*: $cdcl_W$*-stgy.induct*)
case *conflict'*
then show *?case*
unfolding *full1-def* **by** (*simp add*: *tranclp-*$cdcl_W$*-cp-tranclp-*$cdcl_W$)
next
case (*other' S' S''*)
then have *S'* = *S''* ∨ $cdcl_W$*-cp*$^{++}$ *S' S''*
by (*simp add*: *rtranclp-unfold full-def*)
then show *?case*
using *other'* **by** (*meson* $cdcl_W$*.other tranclp.r-into-trancl*
tranclp-$cdcl_W$*-cp-tranclp-*$cdcl_W$ *tranclp-trans*)
qed

lemma *tranclp-cdcl$_W$-stgy-tranclp-cdcl$_W$*:
cdcl$_W$-stgy^{++} S S' $\Longrightarrow$ *cdcl$_W$$^{++}$ S S'*
apply (*induct rule*: *tranclp.induct*)
using *cdcl$_W$-stgy-tranclp-cdcl$_W$* **apply** *blast*
by (*meson cdcl$_W$-stgy-tranclp-cdcl$_W$ tranclp-trans*)

lemma *rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$*:
cdcl$_W$-stgy** S S' $\Longrightarrow$ *cdcl$_W$** S S'*
using *rtranclp-unfold*[*of cdcl$_W$-stgy S S'*] *tranclp-cdcl$_W$-stgy-tranclp-cdcl$_W$*[*of S S'*] **by** *auto*

lemma *not-empty-get-maximum-level-exists-lit*:
assumes *n*: $D \neq \{\#\}$
and *max*: *get-maximum-level M D* = *n*
shows $\exists$ *L* $\in\#$*D*. *get-level M L* = *n*
proof −
have *f*: *finite* (*insert 0* ((λ*L*. *get-level M L*) ' *set-mset D*)) **by** *auto*
then have *n* $\in$ ((λ*L*. *get-level M L*) ' *set-mset D*)
using *n max get-maximum-level-exists-lit-of-max-level image-iff*
unfolding *get-maximum-level-def* **by** *force*
then show $\exists$ *L* $\in\#$ *D*. *get-level M L* = *n* **by** *auto*
qed

lemma *cdcl$_W$-o-conflict-is-false-with-level-inv*:
assumes
cdcl$_W$-o S S' **and**
lev: *cdcl$_W$-M-level-inv S* **and**
confl-inv: *conflict-is-false-with-level S* **and**
n-d: *distinct-cdcl$_W$-state S* **and**
conflicting: *cdcl$_W$-conflicting S*
shows *conflict-is-false-with-level S'*
using *assms*(*1*,*2*)
proof (*induct rule*: *cdcl$_W$-o-induct*)
case (*resolve L C M D T*) **note** *tr-S* = *this*(*1*) **and** *confl* = *this*(*4*) **and** *LD* = *this*(*5*) **and** *T* = *this*(*7*)
have *uL-not-D*: −*L* $\notin\#$ *remove1-mset* (−*L*) *D*
using *n-d confl* **unfolding** *distinct-cdcl$_W$-state-def distinct-mset-def*
by (*metis distinct-cdcl$_W$-state-def distinct-mem-diff-mset multi-member-last n-d*)
moreover have *L-not-D*: *L* $\notin\#$ *remove1-mset* (−*L*) *D*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *L* $\in\#$ *D*
by (*auto simp*: *in-remove1-mset-neq*)
moreover have *Propagated L C # M* $\models$*as CNot D*
using *conflicting confl tr-S* **unfolding** *cdcl$_W$-conflicting-def* **by** *auto*
ultimately have −*L* $\in$ *lits-of-l* (*Propagated L C # M*)
using *in-CNot-implies-uminus*(*2*) **by** *blast*
moreover have *no-dup* (*Propagated L C # M*)
using *lev tr-S* **unfolding** *cdcl$_W$-M-level-inv-def* **by** *auto*
ultimately show *False* **unfolding** *lits-of-def* **by** (*metis consistent-interp-def image-eqI list.set-intros*(*1*) *lits-of-def ann-lit.sel*(*2*) *distinct-consistent-interp*)
qed

ultimately
have *g-D*: *get-maximum-level* (*Propagated L C # M*) (*remove1-mset* (−*L*) *D*)
= *get-maximum-level M* (*remove1-mset* (−*L*) *D*)
using *get-maximum-level-skip-first*[*of L remove1-mset* (−*L*) *D C M*]

    by (*simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def*)
  have *lev-L*[*simp*]: *get-level M L* = *0*
   apply (*rule atm-of-notin-get-rev-level-eq-0*)
   using *lev* **unfolding** $cdcl_W$*-M-level-inv-def tr-S* **by** (*auto simp*: *lits-of-def*)

  have *D*: *get-maximum-level M* (*remove1-mset* ($-L$) *D*) = *backtrack-lvl S*
   using *resolve.hyps*(*6*) *LD* **unfolding** *tr-S* **by** (*auto simp*: *get-maximum-level-plus max-def g-D*)
  have *get-maximum-level M* (*remove1-mset L C*) $\leq$ *backtrack-lvl S*
   using *count-decided-ge-get-maximum-level*[*of M*] *lev* **unfolding** *tr-S* $cdcl_W$*-M-level-inv-def* **by** *auto*
  then have
   get-maximum-level M (*remove1-mset* ($-$ *L*) *D* #∪ *remove1-mset L C*) =
    backtrack-lvl S
   by (*auto simp*: *get-maximum-level-union-mset get-maximum-level-plus max-def D*)
  then show *?case*
   using *tr-S not-empty-get-maximum-level-exists-lit*[*of*
    remove1-mset ($-$ *L*) *D* #∪ *remove1-mset L C M*] *T*
   by *auto*
next
  case (*skip L C′ M D T*) **note** *tr-S* = *this*(*1*) **and** *D* = *this*(*2*) **and** *T* = *this*(*5*)
  then obtain *La* **where**
   La ∈# *D* **and**
   get-level (*Propagated L C′* # *M*) *La* = *backtrack-lvl S*
   using *skip confl-inv* **by** *auto*
  moreover
   have *atm-of La* ≠ *atm-of L*
    proof (*rule ccontr*)
     assume ¬ *?thesis*
     then have *La*: *La* = *L* **using** ‹*La* ∈# *D*› ‹$-$ *L* ∉# *D*›
      by (*auto simp add*: *atm-of-eq-atm-of*)
     have *Propagated L C′* # *M* ⊨*as CNot D*
      using *conflicting tr-S D* **unfolding** $cdcl_W$*-conflicting-def* **by** *auto*
     then have $-L$ ∈ *lits-of-l M*
      using ‹*La* ∈# *D*› *in-CNot-implies-uminus*(*2*)[*of L D Propagated L C′* # *M*] **unfolding** *La*
      by *auto*
     then show *False* **using** *lev tr-S* **unfolding** $cdcl_W$*-M-level-inv-def consistent-interp-def* **by** *auto*
    qed
   then have *get-level* (*Propagated L C′* # *M*) *La* = *get-level M La* **by** *auto*
  ultimately show *?case* **using** *D tr-S T* **by** *auto*
next
  case *backtrack*
  then show *?case*
   by (*auto split*: *if-split-asm simp*: $cdcl_W$*-M-level-inv-decomp lev*)
qed *auto*

Strong completeness

lemma $cdcl_W$*-cp-propagate-confl*:
  assumes $cdcl_W$*-cp S T*
  shows $propagate^{**}$ *S T* ∨ (∃ *S′*. $propagate^{**}$ *S S′* ∧ *conflict S′ T*)
  using *assms* **by** *induction blast+*

lemma *rtranclp-*$cdcl_W$*-cp-propagate-confl*:
  assumes $cdcl_W$*-*cp^{**} *S T*
  shows $propagate^{**}$ *S T* ∨ (∃ *S′*. $propagate^{**}$ *S S′* ∧ *conflict S′ T*)
  by (*simp add*: *assms rtranclp-*$cdcl_W$*-cp-propa-or-propa-confl*)

lemma *propagate-high-levelE*:
  assumes *propagate S T*
  obtains M' N' *U k L C* **where**
    state S = (M', N', *U*, *k*, *None*) **and**
    state T = (*Propagated L* (*C* + {#*L*#}) # M', N', *U*, *k*, *None*) **and**
    C + {#*L*#} ∈# *local.clauses S* **and**
    M' ⊨*as CNot C* **and**
    undefined-lit (*trail S*) *L*
proof −
  obtain *E L* **where**
    conf: *conflicting S* = *None* **and**
    E: *E* ∈# *clauses S* **and**
    LE: *L* ∈# *E* **and**
    tr: *trail S* ⊨*as CNot* (*E* − {#*L*#}) **and**
    undef: *undefined-lit* (*trail S*) *L* **and**
    T: *T* ∼ *cons-trail* (*Propagated L E*) *S*
    using *assms* **by** (*elim propagateE*) *simp*
  obtain *M N U k* **where**
    S: *state S* = (*M*, *N*, *U*, *k*, *None*)
    using *conf* **by** *auto*
  show *thesis*
    using *that*[*of M N U k L remove1-mset L E*] *S T LE E tr undef*
    by *auto*
qed

lemma $cdcl_W$*-cp-propagate-completeness*:
  assumes *MN*: *set M* ⊨*s set-mset N* **and**
  cons: *consistent-interp* (*set M*) **and**
  tot: *total-over-m* (*set M*) (*set-mset N*) **and**
  lits-of-l (*trail S*) ⊆ *set M* **and**
  init-clss S = *N* **and**
  propagate** *S* S' **and**
  learned-clss S = {#}
  shows *length* (*trail S*) ≤ *length* (*trail* S') ∧ *lits-of-l* (*trail* S') ⊆ *set M*
  using *assms*(*6*,*4*,*5*,*7*)
proof (*induction rule*: *rtranclp-induct*)
  case *base*
  then show *?case* **by** *auto*
next
  case (*step Y Z*)
  note *st* = *this*(*1*) **and** *propa* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *lits'* = *this*(*4*) **and** *NS* = *this*(*5*) **and**
    learned = *this*(*6*)
  then have *len*: *length* (*trail S*) ≤ *length* (*trail Y*) **and** *LM*: *lits-of-l* (*trail Y*) ⊆ *set M*
     by *blast+*

  obtain M' N' *U k C L* **where**
    Y: *state Y* = (M', N', *U*, *k*, *None*) **and**
    Z: *state Z* = (*Propagated L* (*C* + {#*L*#}) # M', N', *U*, *k*, *None*) **and**
    C: *C* + {#*L*#} ∈# *clauses Y* **and**
    M'*-C*: M' ⊨*as CNot C* **and**
    undefined-lit (*trail Y*) *L*
    using *propa* **by** (*auto elim*: *propagate-high-levelE*)
  have *init-clss S* = *init-clss Y*
    using *st* **by** *induction* (*auto elim*: *propagateE*)
  then have [*simp*]: N' = *N* **using** *NS Y Z* **by** *simp*
  have *learned-clss Y* = {#}

using *st learned* **by** *induction* (*auto elim*: *propagateE*)
then have [*simp*]: $U = \{\#\}$ **using** *Y* **by** *auto*
have *set M* $\models$*s CNot C*
using *M′-C LM Y* **unfolding** *true-annots-def Ball-def true-annot-def true-clss-def true-cls-def*
by *force*
moreover
have *set M* $\models$ $C + \{\#L\#\}$
using *MN C learned Y NS* ‹*init-clss S = init-clss Y*› ‹*learned-clss Y* = $\{\#\}$›
unfolding *true-clss-def clauses-def* **by** *fastforce*
ultimately have $L \in$ *set M* **by** (*simp add*: *cons consistent-CNot-not*)
then show *?case* **using** *LM len Y Z* **by** *auto*
qed

lemma
assumes *propagate*** *S X*
shows
rtranclp-propagate-init-clss: *init-clss X = init-clss S* **and**
rtranclp-propagate-learned-clss: *learned-clss X = learned-clss S*
using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto elim*: *propagateE*)

lemma *completeness-is-a-full1-propagation*:
fixes *S* :: *′st* **and** *M* :: *′v literal list*
assumes *MN*: *set M* $\models$*s set-mset N*
and *cons*: *consistent-interp* (*set M*)
and *tot*: *total-over-m* (*set M*) (*set-mset N*)
and *alien*: *no-strange-atm S*
and *learned*: *learned-clss S* = $\{\#\}$
and *clsS*[*simp*]: *init-clss S = N*
and *lits*: *lits-of-l* (*trail S*) $\subseteq$ *set M*
shows $\exists S'$. *propagate*** *S S′* $\wedge$ *full cdcl*$_W$*-cp S S′*
proof −
obtain *S′* **where** *full*: *full cdcl*$_W$*-cp S S′*
using *always-exists-full-cdcl*$_W$*-cp-step alien* **by** *blast*
then consider (*propa*) *propagate*** *S S′*
| (*confl*) $\exists X$. *propagate*** *S X* $\wedge$ *conflict X S′*
using *rtranclp-cdcl*$_W$*-cp-propagate-confl* **unfolding** *full-def* **by** *blast*
then show *?thesis*
proof *cases*
case *propa* **then show** *?thesis* **using** *full* **by** *blast*
next
case *confl*
then obtain *X* **where**
X: *propagate*** *S X* **and**
Xconf: *conflict X S′*
by *blast*
have *clsX*: *init-clss X = init-clss S*
using *X* **by** (*blast dest*: *rtranclp-propagate-init-clss*)
have *learnedX*: *learned-clss X* = $\{\#\}$
using *X learned* **by** (*auto dest*: *rtranclp-propagate-learned-clss*)
obtain *E* **where**
E: $E \in\#$ *init-clss X + learned-clss X* **and**
Not-E: *trail X* $\models$*as CNot E*
using *Xconf* **by** (*auto simp add*: *clauses-def elim*!: *conflictE*)
have *lits-of-l* (*trail X*) $\subseteq$ *set M*
using *cdcl*$_W$*-cp-propagate-completeness*[*OF assms*(*1−3*) *lits* - *X learned*] *learned* **by** *auto*
then have *MNE*: *set M* $\models$*s CNot E*

using *Not-E*
by (*fastforce simp add*: *true-annots-def true-annot-def true-clss-def true-cls-def*)
have ¬ *set M* $\models$*s set-mset N*
using *E consistent-CNot-not*[*OF cons MNE*]
unfolding *learnedX true-clss-def* **unfolding** *clsX clsS* **by** *auto*
then show *?thesis* **using** *MN* **by** *blast*
qed
qed

See also theorem *rtranclp-cdcl*$_W$*-cp-dropWhile-trail*

lemma *rtranclp-propagate-is-trail-append*:
propagate** *S T* $\Longrightarrow$ $\exists$ *c. trail T* = *c* @ *trail S*
by (*induction rule*: *rtranclp-induct*) (*auto elim*: *propagateE*)

lemma *rtranclp-propagate-is-update-trail*:
propagate** *S T* $\Longrightarrow$ *cdcl*$_W$*-M-level-inv S* $\Longrightarrow$
init-clss S = *init-clss T* ∧ *learned-clss S* = *learned-clss T* ∧ *backtrack-lvl S* = *backtrack-lvl T*
∧ *conflicting S* = *conflicting T*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **unfolding** *state-eq-def* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
next
case (*step T U*) **note** *IH* = *this*(*3*)[*OF this*(*4*)]
moreover have *cdcl*$_W$*-M-level-inv U*
using *rtranclp-cdcl*$_W$*-consistent-inv* ‹*propagate*** *S T*› ‹*propagate T U*›
rtranclp-mono[*of propagate cdcl*$_W$] *cdcl*$_W$*-cp-consistent-inv propagate′*
rtranclp-propagate-is-rtranclp-cdcl$_W$ *step.prems* **by** *blast*
then have *no-dup* (*trail U*) **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
ultimately show *?case* **using** ‹*propagate T U*› **unfolding** *state-eq-def*
by (*fastforce simp*: *elim*: *propagateE*)
qed

lemma *cdcl*$_W$*-stgy-strong-completeness-n*:
assumes
MN: *set M* $\models$*s set-mset N* **and**
cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset N*) **and**
atm-incl: *atm-of* ‘ (*set M*) ⊆ *atms-of-mm N* **and**
distM: *distinct M* **and**
length: *n* ≤ *length M*
shows
∃ *M′ k S. length M′* ≥ *n* ∧
lits-of-l M′ ⊆ *set M* ∧
no-dup M′ ∧
state S = (*M′*, *N*, {#}, *k*, *None*) ∧
cdcl$_W$*-stgy*** (*init-state N*) *S*
using *length*
proof (*induction n*)
case *0*
have *state* (*init-state N*) = ([], *N*, {#}, *0*, *None*)
by (*auto simp*: *state-eq-def simp del*: *state-simp*)
moreover have
0 ≤ *length* [] **and**
lits-of-l [] ⊆ *set M* **and**
cdcl$_W$*-stgy*** (*init-state N*) (*init-state N*)
and *no-dup* []

by (*auto simp*: *state-eq-def simp del*: *state-simp*)
ultimately show *?case* **using** *state-eq-sym* **by** *blast*
next
case (*Suc n*) **note** $IH = this(1)$ **and** $n = this(2)$
then obtain M' k S **where**
l-M': *length* $M' \geq n$ **and**
M': *lits-of-l* $M' \subseteq$ *set* M **and**
n-d[*simp*]: *no-dup* M' **and**
S: *state* $S = (M', N, \{\#\}, k, None)$ **and**
st: $cdcl_W$-*stgy*** (*init-state* N) S
by *auto*
have
M: $cdcl_W$-*M-level-inv* S **and**
alien: *no-strange-atm* S
using $cdcl_W$-*M-level-inv-S0-*$cdcl_W$ *rtranclp-*$cdcl_W$*-stgy-consistent-inv st* **apply** *blast*
using $cdcl_W$-*M-level-inv-S0-*$cdcl_W$ *no-strange-atm-S0 rtranclp-*$cdcl_W$*-no-strange-atm-inv*
rtranclp-$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ *st* **by** *blast*

{ assume *no-step*: ¬*no-step propagate* S
obtain S' **where** S': *propagate*** S S' **and** *full*: *full* $cdcl_W$-*cp* S S'
using *completeness-is-a-full1-propagation*[*OF assms*(1−3), *of* S] *alien* M' S
by (*auto simp*: *comp-def*)
have *lev*: $cdcl_W$-*M-level-inv* S'
using M S' *rtranclp-*$cdcl_W$*-consistent-inv rtranclp-propagate-is-rtranclp-*$cdcl_W$ **by** *blast*
then have *n-d'*[*simp*]: *no-dup* (*trail* S')
unfolding $cdcl_W$-*M-level-inv-def* **by** *auto*
have *length* (*trail* S) $\leq$ *length* (*trail* S') $\wedge$ *lits-of-l* (*trail* S') $\subseteq$ *set* M
using S' *full* $cdcl_W$*-cp-propagate-completeness*[*OF assms*(1−3), *of* S] M' S
by (*auto simp*: *comp-def*)
moreover
have *full*: *full1* $cdcl_W$-*cp* S S'
using *full no-step no-step-*$cdcl_W$*-cp-no-conflict-no-propagate*(2) **unfolding** *full1-def full-def*
rtranclp-unfold **by** *blast*
then have $cdcl_W$-*stgy* S S' **by** (*simp add*: $cdcl_W$-*stgy.conflict'*)
moreover
have *propa*: *propagate*$^{++}$ S S' **using** S' *full* **unfolding** *full1-def* **by** (*metis rtranclpD tranclpD*)
have *trail* $S = M'$
using S **by** (*auto simp*: *comp-def rev-map*)
with *propa* **have** *length* (*trail* S') $> n$
using l-M' *propa* **by** (*induction rule*: *tranclp.induct*) (*auto elim*: *propagateE*)
moreover
have *stS'*: $cdcl_W$-*stgy*** (*init-state* N) S'
using *st* $cdcl_W$-*stgy.conflict'*[*OF full*] **by** *auto*
then have *init-clss* $S' = N$
using *stS'* *rtranclp-*$cdcl_W$*-stgy-no-more-init-clss* **by** *fastforce*
moreover
have
[*simp*]:*learned-clss* $S' = \{\#\}$ **and**
[*simp*]: *init-clss* S' = *init-clss* S **and**
[*simp*]: *conflicting* S' = *None*
using *tranclp-into-rtranclp*[*OF* ‹*propagate*$^{++}$ S S'›] S
rtranclp-propagate-is-update-trail[*of* S S'] S M **unfolding** *state-eq-def*
by (*auto simp*: *comp-def*)
have *S-S'*: *state* $S' = (trail\ S', N, \{\#\}, backtrack\text{-}lvl\ S', None)$
using S **by** *auto*
have $cdcl_W$-*stgy*** (*init-state* N) S'

```
      apply (rule rtranclp.rtrancl-into-rtrancl)
       using st apply simp
      using ‹cdcl_W-stgy S S'› by simp
  ultimately have ?case
    apply −
    apply (rule exI[of - trail S'], rule exI[of - backtrack-lvl S'], rule exI[of - S'])
    using S-S' by (auto simp: state-eq-def simp del: state-simp)
}
moreover {
  assume no-step: no-step propagate S
  have ?case
    proof (cases length M' ≥ Suc n)
      case True
      then show ?thesis using l-M' M' st M alien S n-d by blast
    next
      case False
      then have n': length M' = n using l-M' by auto
      have no-confl: no-step conflict S
        proof −
          { fix D
            assume D ∈# N and M' ⊨as CNot D
            then have set M ⊨ D using MN unfolding true-clss-def by auto
            moreover have set M ⊨s CNot D
              using ‹M' ⊨as CNot D› M'
              by (metis le-iff-sup true-annots-true-cls true-clss-union-increase)
            ultimately have False using cons consistent-CNot-not by blast
          }
          then show ?thesis
            using S by (auto simp: true-clss-def comp-def rev-map
              clauses-def elim!: conflictE)
        qed
      have lenM: length M = card (set M) using distM by (induction M) auto
      have no-dup M' using S M unfolding cdcl_W-M-level-inv-def by auto
      then have card (lits-of-l M') = length M'
        by (induction M') (auto simp add: lits-of-def card-insert-if)
      then have lits-of-l M' ⊂ set M
        using n M' n' lenM by auto
      then obtain L where L: L ∈ set M and undef-m: L ∉ lits-of-l M' by auto
      moreover have undef: undefined-lit M' L
        using M' Decided-Propagated-in-iff-in-lits-of-l calculation(1,2) cons
        consistent-interp-def by (metis (no-types, lifting) subset-eq)
      moreover have atm-of L ∈ atms-of-mm (init-clss S)
        using atm-incl calculation S by auto
      ultimately
        have dec: decide S (cons-trail (Decided L) (incr-lvl S))
          using decide-rule[of S - cons-trail (Decided L) (incr-lvl S)] S
          by auto
      let ?S' = cons-trail (Decided L) (incr-lvl S)
      have lits-of-l (trail ?S') ⊆ set M using L M' S undef by auto
      moreover have no-strange-atm ?S'
        using alien dec M by (meson cdcl_W-no-strange-atm-inv decide other)
      ultimately obtain S'' where S'': propagate** ?S' S'' and full: full cdcl_W-cp ?S' S''
        using completeness-is-a-full1-propagation[OF assms(1−3), of ?S'] S undef
        by auto
      have cdcl_W-M-level-inv ?S'
        using M dec rtranclp-mono[of decide cdcl_W] by (meson cdcl_W-consistent-inv decide other)
```

then have *lev''*: $cdcl_W$*-M-level-inv S''*
using *S'' rtranclp-*$cdcl_W$*-consistent-inv rtranclp-propagate-is-rtranclp-*$cdcl_W$ **by** *blast*
then have *n-d''*: *no-dup* (*trail S''*)
unfolding $cdcl_W$*-M-level-inv-def* **by** *auto*
have *length* (*trail ?S'*) $\leq$ *length* (*trail S''*) $\wedge$ *lits-of-l* (*trail S''*) $\subseteq$ *set M*
using *S'' full* $cdcl_W$*-cp-propagate-completeness*[*OF assms*(*1−3*), *of ?S' S''*] *L M' S undef*
by *simp*
then have *Suc n* $\leq$ *length* (*trail S''*) $\wedge$ *lits-of-l* (*trail S''*) $\subseteq$ *set M*
using *l-M' S undef* **by** *auto*
moreover
have $cdcl_W$*-M-level-inv* (*cons-trail* (*Decided L*)
(*update-backtrack-lvl* (*Suc* (*backtrack-lvl S*)) *S*))
using *S* ‹$cdcl_W$*-M-level-inv* (*cons-trail* (*Decided L*) (*incr-lvl S*))› **by** *auto*
then have *S''*:
state S'' = (*trail S''*, *N*, {#}, *backtrack-lvl S''*, *None*)
using *rtranclp-propagate-is-update-trail*[*OF S''*] *S undef n-d'' lev''*
by *auto*
then have $cdcl_W$*-stgy*** (*init-state N*) *S''*
using $cdcl_W$*-stgy.intros*(*2*)[*OF decide*[*OF dec*] *- full*] *no-step no-confl st*
by (*auto simp*: $cdcl_W$*-cp.simps*)
ultimately show *?thesis* **using** *S'' n-d''* **by** *blast*
qed
}
ultimately show *?case* **by** *blast*
qed

theorem 2.9.11 page 84 of Weidenbach's book (with strategy)

lemma $cdcl_W$*-stgy-strong-completeness*:
assumes
MN: *set M* $\models$*s set-mset N* **and**
cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset N*) **and**
atm-incl: *atm-of* ‘ (*set M*) $\subseteq$ *atms-of-mm N* **and**
distM: *distinct M*
shows
$\exists$ *M' k S*.
lits-of-l M' = *set M* $\wedge$
state S = (*M'*, *N*, {#}, *k*, *None*) $\wedge$
$cdcl_W$*-stgy*** (*init-state N*) *S* $\wedge$
final-$cdcl_W$*-state S*
proof −
from $cdcl_W$*-stgy-strong-completeness-n*[*OF assms*, *of length M*]
obtain *M' k T* **where**
l: *length M* $\leq$ *length M'* **and**
M'-M: *lits-of-l M'* $\subseteq$ *set M* **and**
no-dup: *no-dup M'* **and**
T: *state T* = (*M'*, *N*, {#}, *k*, *None*) **and**
st: $cdcl_W$*-stgy*** (*init-state N*) *T*
by *auto*
have *card* (*set M*) = *length M* **using** *distM* **by** (*simp add*: *distinct-card*)
moreover
have $cdcl_W$*-M-level-inv T*
using *rtranclp-*$cdcl_W$*-stgy-consistent-inv*[*OF st*] *T* **by** *auto*
then have *card* (*set* ((*map* (λ*l. atm-of* (*lit-of l*)) *M'*))) = *length M'*
using *distinct-card no-dup* **by** *fastforce*
moreover have *card* (*lits-of-l M'*) = *card* (*set* ((*map* (λ*l. atm-of* (*lit-of l*)) *M'*)))

using *no-dup* **unfolding** *lits-of-def* **apply** (*induction M'*) **by** (*auto simp add*: *card-insert-if*)
ultimately have *card* (*set M*) $\leq$ *card* (*lits-of-l M'*) **using** *l* **unfolding** *lits-of-def* **by** *auto*
then have *set M* = *lits-of-l M'*
using *M'-M card-seteq* **by** *blast*
moreover
then have *M'* $\models$*asm N*
using *MN* **unfolding** *true-annots-def Ball-def true-annot-def true-clss-def* **by** *auto*
then have *final-cdcl*$_W$*-state T*
using *T no-dup* **unfolding** *final-cdcl*$_W$*-state-def* **by** *auto*
ultimately show *?thesis* **using** *st T* **by** *blast*
qed

No conflict with only variables of level less than backtrack level

This invariant is stronger than the previous argument in the sense that it is a property about all possible conflicts.

definition *no-smaller-confl* (*S* ::*'st*) $\equiv$
($\forall$ *M K M' D. M'* @ *Decided K* # *M* = *trail S* $\longrightarrow$ *D* $\in$# *clauses S*
$\longrightarrow$ $\neg$*M* $\models$*as CNot D*)

lemma *no-smaller-confl-init-sate*[*simp*]:
no-smaller-confl (*init-state N*) **unfolding** *no-smaller-confl-def* **by** *auto*

lemma *cdcl*$_W$*-o-no-smaller-confl-inv*:
fixes *S S'* :: *'st*
assumes
cdcl$_W$*-o S S'* **and**
lev: *cdcl*$_W$*-M-level-inv S* **and**
max-lev: *conflict-is-false-with-level S* **and**
smaller: *no-smaller-confl S* **and**
no-f: *no-clause-is-false S*
shows *no-smaller-confl S'*
using *assms*(*1*,*2*) **unfolding** *no-smaller-confl-def*
proof (*induct rule*: *cdcl*$_W$*-o-induct*)
case (*decide L T*) **note** *confl* = *this*(*1*) **and** *undef* = *this*(*2*) **and** *T* = *this*(*4*)
have [*simp*]: *clauses T* = *clauses S*
using *T undef* **by** *auto*
show *?case*
proof (*intro allI impI*)
fix *M'' K M' Da*
assume *M''* @ *Decided K* # *M'* = *trail T*
and *D*: *Da* $\in$# *local.clauses T*
then have *tl M''* @ *Decided K* # *M'* = *trail S*
$\vee$ (*M''* = [] $\wedge$ *Decided K* # *M'* = *Decided L* # *trail S*)
using *T undef* **by** (*cases M''*) *auto*
moreover {
assume *tl M''* @ *Decided K* # *M'* = *trail S*
then have $\neg$*M'* $\models$*as CNot Da*
using *D T undef no-f confl smaller* **unfolding** *no-smaller-confl-def smaller* **by** *fastforce*
}
moreover {
assume *Decided K* # *M'* = *Decided L* # *trail S*
then have $\neg$*M'* $\models$*as CNot Da* **using** *no-f D confl T* **by** *auto*
}
ultimately show $\neg$*M'* $\models$*as CNot Da* **by** *fast*

qed
next
case *resolve*
then show *?case* **using** *smaller no-f max-lev* **unfolding** *no-smaller-confl-def* **by** *auto*
next
case *skip*
then show *?case* **using** *smaller no-f max-lev* **unfolding** *no-smaller-confl-def* **by** *auto*
next
case (*backtrack L D K i M1 M2 T*) **note** *confl* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *decomp* = *this*(*3*) **and**
T = *this*(*8*)
obtain *c* **where** *M*: *trail S* = *c* @ *M2* @ *Decided K* # *M1*
using *decomp* **by** *auto*

show *?case*
proof (*intro allI impI*)
fix *M ia K' M' Da*
assume *M'* @ *Decided K'* # *M* = *trail T*
then have *tl M'* @ *Decided K'* # *M* = *M1*
using *T decomp lev* **by** (*cases M'*) (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
let *?S'* = (*cons-trail* (*Propagated L D*)
(*reduce-trail-to M1* (*add-learned-cls D*
(*update-backtrack-lvl i* (*update-conflicting None S*)))))
assume *D*: *Da* ∈# *clauses T*
moreover{
assume *Da* ∈# *clauses S*
then have ¬*M* ⊨*as CNot Da* **using** ‹*tl M'* @ *Decided K'* # *M* = *M1*› *M confl smaller*
unfolding *no-smaller-confl-def* **by** *auto*
}
moreover {
assume *Da*: *Da* = *D*
have ¬*M* ⊨*as CNot Da*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have −*L* ∈ *lits-of-l M*
using *LD* **unfolding** *Da* **by** (*simp add*: *in-CNot-implies-uminus*(*2*))
then have −*L* ∈ *lits-of-l* (*Propagated L D* # *M1*)
using *UnI2* ‹*tl M'* @ *Decided K'* # *M* = *M1*›
by *auto*
moreover
have *backtrack S ?S'*
using *backtrack-rule*[*of S*] *backtrack.hyps*
by (*force simp*: *state-eq-def simp del*: *state-simp*)
then have $cdcl_W$*-M-level-inv ?S'*
using $cdcl_W$*-consistent-inv*[*OF - lev*] *other*[*OF bj*] **by** (*auto intro*: $cdcl_W$*-bj.intros*)
then have *no-dup* (*Propagated L D* # *M1*)
using *decomp lev* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
ultimately show *False*
using *Decided-Propagated-in-iff-in-lits-of-l defined-lit-map* **by** *auto*
qed
}
ultimately show ¬*M* ⊨*as CNot Da*
using *T decomp lev* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *fastforce*
qed
qed

lemma *conflict-no-smaller-confl-inv*:
  assumes *conflict* S S'
  and *no-smaller-confl* S
  shows *no-smaller-confl* S'
  using *assms* **unfolding** *no-smaller-confl-def* **by** (*fastforce elim*: *conflictE*)

lemma *propagate-no-smaller-confl-inv*:
  assumes *propagate*: *propagate* S S'
  and *n-l*: *no-smaller-confl* S
  shows *no-smaller-confl* S'
  unfolding *no-smaller-confl-def*
proof (*intro allI impI*)
  fix M' K M'' D
  assume M': M'' @ *Decided* K # M' = *trail* S'
  and $D \in\#$ *clauses* S'
  obtain M N U k C L **where**
    S: *state* S = (M, N, U, k, *None*) **and**
    S': *state* S' = (*Propagated* L (C + {#L#}) # M, N, U, k, *None*) **and**
    C + {#L#} $\in\#$ *clauses* S **and**
    M $\models$*as* *CNot* C **and**
    undefined-lit M L
    using *propagate* **by** (*auto elim*: *propagate-high-levelE*)
  have *tl* M'' @ *Decided* K # M' = *trail* S **using** M' S S'
    by (*metis Pair-inject list.inject list.sel*(*3*) *ann-lit.distinct*(*1*) *self-append-conv2*
      tl-append2)
  then have $\neg M'$ $\models$*as* *CNot* D
    using ‹$D \in\#$ *clauses* S'› *n-l* S S' *clauses-def* **unfolding** *no-smaller-confl-def* **by** *auto*
  then show $\neg M'$ $\models$*as* *CNot* D **by** *auto*
qed

lemma *cdcl*$_W$*-cp-no-smaller-confl-inv*:
  assumes *propagate*: *cdcl*$_W$*-cp* S S'
  and *n-l*: *no-smaller-confl* S
  shows *no-smaller-confl* S'
  using *assms*
proof (*induct rule*: *cdcl*$_W$*-cp.induct*)
  case (*conflict'* S S')
  then show *?case* **using** *conflict-no-smaller-confl-inv*[*of* S S'] **by** *blast*
next
  case (*propagate'* S S')
  then show *?case* **using** *propagate-no-smaller-confl-inv*[*of* S S'] **by** *fastforce*
qed

lemma *rtrancp-cdcl*$_W$*-cp-no-smaller-confl-inv*:
  assumes *propagate*: *cdcl*$_W$*-cp*** S S'
  and *n-l*: *no-smaller-confl* S
  shows *no-smaller-confl* S'
  using *assms*
proof (*induct rule*: *rtranclp-induct*)
  case *base*
  then show *?case* **by** *simp*
next
  case (*step* S' S'')
  then show *?case* **using** *cdcl*$_W$*-cp-no-smaller-confl-inv*[*of* S' S''] **by** *fast*
qed

lemma *trancp-cdcl$_W$-cp-no-smaller-confl-inv*:
  assumes *propagate*: *cdcl$_W$-cp^{++} S S'*
  and *n-l*: *no-smaller-confl S*
  shows *no-smaller-confl S'*
  using *assms*
proof (*induct rule*: *tranclp.induct*)
  case (*r-into-trancl S S'*)
  then show *?case* **using** *cdcl$_W$-cp-no-smaller-confl-inv*[*of S S'*] **by** *blast*
next
  case (*trancl-into-trancl S S' S''*)
  then show *?case* **using** *cdcl$_W$-cp-no-smaller-confl-inv*[*of S' S''*] **by** *fast*
qed

lemma *full-cdcl$_W$-cp-no-smaller-confl-inv*:
  assumes *full cdcl$_W$-cp S S'*
  and *n-l*: *no-smaller-confl S*
  shows *no-smaller-confl S'*
  using *assms* **unfolding** *full-def*
  using *rtrancp-cdcl$_W$-cp-no-smaller-confl-inv*[*of S S'*] **by** *blast*

lemma *full1-cdcl$_W$-cp-no-smaller-confl-inv*:
  assumes *full1 cdcl$_W$-cp S S'*
  and *n-l*: *no-smaller-confl S*
  shows *no-smaller-confl S'*
  using *assms* **unfolding** *full1-def*
  using *trancp-cdcl$_W$-cp-no-smaller-confl-inv*[*of S S'*] **by** *blast*

lemma *cdcl$_W$-stgy-no-smaller-confl-inv*:
  assumes *cdcl$_W$-stgy S S'*
  and *n-l*: *no-smaller-confl S*
  and *conflict-is-false-with-level S*
  and *cdcl$_W$-M-level-inv S*
  shows *no-smaller-confl S'*
  using *assms*
proof (*induct rule*: *cdcl$_W$-stgy.induct*)
  case (*conflict' S'*)
  then show *?case* **using** *full1-cdcl$_W$-cp-no-smaller-confl-inv*[*of S S'*] **by** *blast*
next
  case (*other' S' S''*)
  have *no-smaller-confl S'*
    using *cdcl$_W$-o-no-smaller-confl-inv*[*OF other'.hyps*(*1*) *other'.prems*(*3*,*2*,*1*)]
    not-conflict-not-any-negated-init-clss other'.hyps(*2*) *cdcl$_W$-cp.simps* **by** *auto*
  then show *?case* **using** *full-cdcl$_W$-cp-no-smaller-confl-inv*[*of S' S''*] *other'.hyps* **by** *blast*
qed

lemma *is-conflicting-exists-conflict*:
  assumes $\neg$($\forall$ *D*$\in$#*init-clss S' + learned-clss S'*. $\neg$ *trail S'* $\models$*as CNot D*)
  and *conflicting S' = None*
  shows $\exists$ *S''*. *conflict S' S''*
  using *assms clauses-def not-conflict-not-any-negated-init-clss* **by** *fastforce*

lemma *cdcl$_W$-o-conflict-is-no-clause-is-false*:
  fixes *S S'* :: *'st*
  assumes
    cdcl$_W$-o S S' **and**
    lev: *cdcl$_W$-M-level-inv S* **and**

```
    max-lev: conflict-is-false-with-level S and
    no-f: no-clause-is-false S and
    no-l: no-smaller-confl S
  shows no-clause-is-false S'
    ∨ (conflicting S' = None
        ⟶ (∀ D ∈# clauses S'. trail S' ⊨as CNot D
            ⟶ (∃ L. L ∈# D ∧ get-level (trail S') L = backtrack-lvl S')))
  using assms(1,2)
proof (induct rule: cdcl_W-o-induct)
  case (decide L T) note S = this(1) and undef = this(2) and T = this(4)
  show ?case
    proof (rule HOL.disjI2, clarify)
      fix D
      assume D: D ∈# clauses T and M-D: trail T ⊨as CNot D
      let ?M = trail S
      let ?M' = trail T
      let ?k = backtrack-lvl S
      have ¬?M ⊨as CNot D
          using no-f D S T undef by auto
      have −L ∈# D
        proof (rule ccontr)
          assume ¬ ?thesis
          have ?M ⊨as CNot D
            unfolding true-annots-def Ball-def true-annot-def CNot-def true-cls-def
            proof (intro allI impI)
              fix x
              assume x: x ∈ {{#− L#} |L. L ∈# D}

              then obtain L' where L': x = {#−L'#} L' ∈# D by auto
              obtain L'' where L'' ∈# x and L'': lits-of-l (Decided L # ?M) ⊨l L''
                using M-D x T undef unfolding true-annots-def Ball-def true-annot-def CNot-def
                true-cls-def Bex-def by auto
              show ∃ L ∈# x. lits-of-l ?M ⊨l L unfolding Bex-def
                using L'(1) L'(2) ‹− L ∉# D› ‹L'' ∈# x›
                ‹lits-of-l (Decided L # trail S) ⊨l L''› by auto
            qed
          then show False using ‹¬ ?M ⊨as CNot D› by auto
        qed
      have atm-of L ∉ atm-of ` (lits-of-l ?M)
        using undef defined-lit-map unfolding lits-of-def by fastforce
      then have get-level (Decided L # ?M) (−L) = ?k + 1
        using lev unfolding cdcl_W-M-level-inv-def by auto
      then have −L ∈# D ∧ get-level ?M' (−L) = backtrack-lvl T
        using ‹−L ∈# D› T undef by auto
      then show ∃ La. La ∈# D ∧ get-level ?M' La = backtrack-lvl T
        by blast
    qed
next
  case resolve
  then show ?case by auto
next
  case skip
  then show ?case by auto
next
  case (backtrack L D K i M1 M2 T) note decomp = this(3) and lev-K = this(7) and T = this(8)
  show ?case
```

```
    proof (rule HOL.disjI2, clarify)
      fix Da
      assume Da: Da ∈# clauses T and M-D: trail T ⊨as CNot Da
      obtain c where M: trail S = c @ M2 @ Decided K # M1
        using decomp by auto
      have tr-T: trail T = Propagated L D # M1
        using T decomp lev by (auto simp: cdcl_W-M-level-inv-decomp)
      have backtrack S T
        using backtrack-rule[of S] backtrack.hyps T
        by (force simp del: state-simp simp: state-eq-def)
      then have lev': cdcl_W-M-level-inv T
        using cdcl_W-consistent-inv lev other cdcl_W-bj.backtrack cdcl_W-o.bj by blast
      then have − L ∉ lits-of-l M1
        using lev cdcl_W-M-level-inv-def tr-T unfolding consistent-interp-def by (metis insert-iff
          list.simps(15) lits-of-insert ann-lit.sel(2))
      { assume Da ∈# clauses S
        then have ¬M1 ⊨as CNot Da using no-l M unfolding no-smaller-confl-def by auto
      }
      moreover {
        assume Da: Da = D
        have ¬M1 ⊨as CNot Da using ‹− L ∉ lits-of-l M1› unfolding Da
          using backtrack.hyps(2) in-CNot-implies-uminus(2) by auto
      }
      ultimately have ¬M1 ⊨as CNot Da
        using Da T decomp lev by (fastforce simp: cdcl_W-M-level-inv-decomp)
      then have −L ∈# Da
        using M-D ‹− L ∉ lits-of-l M1› T unfolding tr-T true-annots-true-cls true-clss-def
        by (auto simp: uminus-lit-swap)
      have no-dup (Propagated L D # M1)
        using lev lev' T decomp unfolding cdcl_W-M-level-inv-def by auto
      then have L: atm-of L ∉ atm-of ‘ lits-of-l M1 unfolding lits-of-def by auto
      have get-level (Propagated L D # M1) (−L) = i
        using lev-K lev unfolding cdcl_W-M-level-inv-def
        by (simp add: M image-Un atm-lit-of-set-lits-of-l)

      then have −L ∈# Da ∧ get-level (trail T) (−L) = backtrack-lvl T
        using ‹−L ∈# Da› T decomp lev by (auto simp: cdcl_W-M-level-inv-def)
      then show ∃La. La ∈# Da ∧ get-level (trail T) La = backtrack-lvl T
        by blast
    qed
qed

lemma full1-cdcl_W-cp-exists-conflict-decompose:
  assumes
    confl: ∃D∈#clauses S. trail S ⊨as CNot D and
    full: full cdcl_W-cp S U and
    no-confl: conflicting S = None and
    lev: cdcl_W-M-level-inv S
  shows ∃T. propagate** S T ∧ conflict T U
proof −
  consider (propa) propagate** S U
      | (confl) T where propagate** S T and conflict T U
   using full unfolding full-def by (blast dest:rtranclp-cdcl_W-cp-propa-or-propa-confl)
  then show ?thesis
    proof cases
      case confl
```

    then show *?thesis* **by** *blast*
  next
    case *propa*
    then have *conflicting* $U = None$ **and**
      [*simp*]: *learned-clss* $U =$ *learned-clss* S **and**
      [*simp*]: *init-clss* $U =$ *init-clss* S
      using *no-confl rtranclp-propagate-is-update-trail lev* **by** *auto*
    moreover
      obtain D **where** D: $D \in\#$ *clauses* U **and**
        trS: *trail* $S \models as$ *CNot* D
        using *confl clauses-def* **by** *auto*
      obtain M **where** M: *trail* $U = M$ @ *trail* S
        using *full rtranclp-cdcl*$_W$*-cp-dropWhile-trail* **unfolding** *full-def* **by** *meson*
      have *tr-U*: *trail* $U \models as$ *CNot* D
        apply (*rule true-annots-mono*)
        using *trS* **unfolding** M **by** *simp-all*
    have $\exists V$. *conflict* U V
      using ‹*conflicting* $U = None$› D *clauses-def not-conflict-not-any-negated-init-clss tr-U*
      by *meson*
    then have *False* **using** *full cdcl*$_W$*-cp.conflict'* **unfolding** *full-def* **by** *blast*
    then show *?thesis* **by** *fast*
  qed
qed

lemma *full1-cdcl*$_W$*-cp-exists-conflict-full1-decompose*:
  assumes
    confl: $\exists D \in\#$*clauses* S. *trail* $S \models as$ *CNot* D **and**
    full: *full cdcl*$_W$*-cp* S U **and**
    no-confl: *conflicting* $S = None$**and**
    lev: *cdcl*$_W$*-M-level-inv* S
  shows $\exists T$ D. *propagate*** S T $\wedge$ *conflict* T U
    $\wedge$ *trail* $T \models as$ *CNot* D $\wedge$ *conflicting* $U = Some$ D $\wedge$ $D \in\#$ *clauses* S
proof −
  obtain T **where** *propa*: *propagate*** S T **and** *conf*: *conflict* T U
    using *full1-cdcl*$_W$*-cp-exists-conflict-decompose*[*OF assms*] **by** *blast*
  have *p*: *learned-clss* $T =$ *learned-clss* S *init-clss* $T =$ *init-clss* S
     using *propa lev rtranclp-propagate-is-update-trail* **by** *auto*
  have *c*: *learned-clss* $U =$ *learned-clss* T *init-clss* $U =$ *init-clss* T
     using *conf* **by** (*auto elim*: *conflictE*)
  obtain D **where** *trail* $T \models as$ *CNot* D $\wedge$ *conflicting* $U = Some$ D $\wedge$ $D \in\#$ *clauses* S
    using *conf p c* **by** (*fastforce simp*: *clauses-def elim*!: *conflictE*)
  then show *?thesis*
    using *propa conf* **by** *blast*
qed

lemma *cdcl*$_W$*-stgy-no-smaller-confl*:
  assumes
    cdcl$_W$*-stgy* S S' **and**
    n-l: *no-smaller-confl* S **and**
    conflict-is-false-with-level S **and**
    cdcl$_W$*-M-level-inv* S **and**
    no-clause-is-false S **and**
    distinct-cdcl$_W$*-state* S **and**
    cdcl$_W$*-conflicting* S
  shows *no-smaller-confl* S'
  using *assms*

proof (*induct rule*: *cdcl*$_W$*-stgy.induct*)
  case (*conflict′ S′*)
  show *no-smaller-confl S′*
    using *conflict′.hyps conflict′.prems*(*1*) *full1-cdcl*$_W$*-cp-no-smaller-confl-inv* **by** *blast*
next
  case (*other′ S′ S″*)
  have *lev′*: *cdcl*$_W$*-M-level-inv S′*
    using *cdcl*$_W$*-consistent-inv other other′.hyps*(*1*) *other′.prems*(*3*) **by** *blast*
  show *no-smaller-confl S″*
    using *cdcl*$_W$*-stgy-no-smaller-confl-inv*[*OF cdcl*$_W$*-stgy.other′*[*OF other′.hyps*(*1−3*)]]
    other′.prems(*1−3*) **by** *blast*
qed

lemma *cdcl*$_W$*-stgy-ex-lit-of-max-level*:
  assumes
    cdcl$_W$*-stgy S S′* **and**
    n-l: *no-smaller-confl S* **and**
    conflict-is-false-with-level S **and**
    cdcl$_W$*-M-level-inv S* **and**
    no-clause-is-false S **and**
    distinct-cdcl$_W$*-state S* **and**
    cdcl$_W$*-conflicting S*
  shows *conflict-is-false-with-level S′*
  using *assms*
proof (*induct rule*: *cdcl*$_W$*-stgy.induct*)
  case (*conflict′ S′*)
  have *no-smaller-confl S′*
    using *conflict′.hyps conflict′.prems*(*1*) *full1-cdcl*$_W$*-cp-no-smaller-confl-inv* **by** *blast*
  moreover have *conflict-is-false-with-level S′*
    using *conflict′.hyps conflict′.prems*(*2−4*)
    rtranclp-cdcl$_W$*-co-conflict-ex-lit-of-max-level*[*of S S′*]
    unfolding *full-def full1-def rtranclp-unfold* **by** *presburger*
  then show *?case* **by** *blast*
next
  case (*other′ S′ S″*)
  have *lev′*: *cdcl*$_W$*-M-level-inv S′*
    using *cdcl*$_W$*-consistent-inv other other′.hyps*(*1*) *other′.prems*(*3*) **by** *blast*
  moreover
    have *no-clause-is-false S′*
      ∨ (*conflicting S′* = *None* ⟶ (∀ *D*∈#*clauses S′*. *trail S′* ⊨*as CNot D*
          ⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S′*) *L* = *backtrack-lvl S′*)))
      using *cdcl*$_W$*-o-conflict-is-no-clause-is-false*[*of S S′*] *other′.hyps*(*1*) *other′.prems*(*1−4*) **by** *fast*
  moreover {
    assume *no-clause-is-false S′*
    {
      assume *conflicting S′* = *None*
      then have *conflict-is-false-with-level S′* **by** *auto*
      moreover have *full cdcl*$_W$*-cp S′ S″*
        by (*metis* (*no-types*) *other′.hyps*(*3*))
      ultimately have *conflict-is-false-with-level S″*
        using *rtranclp-cdcl*$_W$*-co-conflict-ex-lit-of-max-level*[*of S′ S″*] *lev′* ‹*no-clause-is-false S′*›
        by *blast*
    }
    moreover
    {
      assume *c*: *conflicting S′* ≠ *None*

    have *conflicting S ≠ None* **using** *other'.hyps(1) c*
      by (*induct rule*: *cdcl_W-o-induct*) *auto*
    then have *conflict-is-false-with-level S'*
      using *cdcl_W-o-conflict-is-false-with-level-inv*[*OF other'.hyps*(*1*)]
      other'.prems(*3*,*5*,*6*,*2*) **by** *blast*
    moreover have *cdcl_W-cp*** *S' S''* **using** *other'.hyps*(*3*) **unfolding** *full-def* **by** *auto*
    then have *S' = S''* **using** *c*
      by (*induct rule*: *rtranclp-induct*)
        (*fastforce intro*: *option.exhaust*)+
    ultimately have *conflict-is-false-with-level S''* **by** *auto*
  }
  ultimately have *conflict-is-false-with-level S''* **by** *blast*
}
moreover {
   assume
     confl: *conflicting S' = None* **and**
     D-L: ∀ *D* ∈# *clauses S'*. *trail S'* ⊨*as CNot D*
       ⟶ (∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S'*) *L* = *backtrack-lvl S'*)
   { **assume** ∀ *D*∈#*clauses S'*. ¬ *trail S'* ⊨*as CNot D*
     then have *no-clause-is-false S'* **using** *confl* **by** *simp*
     then have *conflict-is-false-with-level S''* **using** *calculation*(*3*) **by** *presburger*
   }
   moreover {
     assume ¬(∀ *D*∈#*clauses S'*. ¬ *trail S'* ⊨*as CNot D*)
     then obtain *T D* **where**
       *propagate*** *S' T* **and**
       conflict T S'' **and**
       D: *D* ∈# *clauses S'* **and**
       trail S'' ⊨*as CNot D* **and**
       conflicting S'' = Some D
       using *full1-cdcl_W-cp-exists-conflict-full1-decompose*[*OF - - confl*]
       other'(*3*) *lev'* **by** (*metis* (*mono-tags*, *lifting*) *conflictE state-eq-trail*
         trail-update-conflicting)
     obtain *M* **where** *M*: *trail S'' = M @ trail S'* **and** *nm*: ∀ *m*∈*set M*. ¬*is-decided m*
       using *rtranclp-cdcl_W-cp-dropWhile-trail other'*(*3*) **unfolding** *full-def* **by** *meson*
     have *btS*: *backtrack-lvl S'' = backtrack-lvl S'*
       using *other'.hyps*(*3*) **unfolding** *full-def* **by** (*metis rtranclp-cdcl_W-cp-backtrack-lvl*)
     have *inv*: *cdcl_W-M-level-inv S''*
       by (*metis* (*no-types*) *cdcl_W-stgy.conflict' cdcl_W-stgy-consistent-inv full-unfold lev'*
         other'.hyps(*3*))
     then have *nd*: *no-dup* (*trail S''*)
       by (*metis* (*no-types*) *cdcl_W-M-level-inv-decomp*(*2*))
     have *conflict-is-false-with-level S''*
       proof *cases*
         assume *trail S'* ⊨*as CNot D*
         moreover then obtain *L* **where**
           L ∈# *D* **and**
           lev-L: *get-level* (*trail S'*) *L* = *backtrack-lvl S'*
           using *D-L D* **by** *blast*
         moreover
           have *LS'*: −*L* ∈ *lits-of-l* (*trail S'*)
             using ‹*trail S'* ⊨*as CNot D*› ‹*L* ∈# *D*› *in-CNot-implies-uminus*(*2*) **by** *blast*
           { **fix** *x* :: ('*v*, '*v clause*) *ann-lit* **and**
               xb :: ('*v*, '*v clause*) *ann-lit*
             assume *a1*: *x* ∈ *set* (*trail S'*) **and**
               a2: *xb* ∈ *set M* **and**

```
          a3: (λl. atm-of (lit-of l)) ` set M ∩ (λl. atm-of (lit-of l)) ` set (trail S')
            = {} and
           a4: − L = lit-of x and
           a5: atm-of L = atm-of (lit-of xb)
         moreover have atm-of (lit-of x) = atm-of L
           using a4 by (metis (no-types) atm-of-uminus)
         ultimately have False
           using a5 a3 a2 a1 by auto
       }
       then have atm-of L ∉ atm-of ` lits-of-l M
         using nd LS' unfolding M by (auto simp add: lits-of-def)
       then have get-level (trail S'') L = get-level (trail S') L
         unfolding M by (simp add: lits-of-def)
     ultimately show ?thesis using btS ‹conflicting S'' = Some D› by auto
   next
     assume ¬trail S' ⊨as CNot D
     then obtain L where L ∈# D and LM: −L ∈ lits-of-l M
       using ‹trail S'' ⊨as CNot D› unfolding M
         by (auto simp add: true-cls-def M true-annots-def true-annot-def
             split: if-split-asm)
     { fix x :: ('v, 'v clause) ann-lit and
         xb :: ('v, 'v clause) ann-lit
       assume a1: xb ∈ set (trail S') and
         a2: x ∈ set M and
         a3: atm-of L = atm-of (lit-of xb) and
         a4: − L = lit-of x and
         a5: (λl. atm-of (lit-of l)) ` set M ∩ (λl. atm-of (lit-of l)) ` set (trail S')
           = {}
       moreover have atm-of (lit-of xb) = atm-of (− L)
         using a3 by simp
       ultimately have False
         by auto }
     then have LS': atm-of L ∉ atm-of ` lits-of-l (trail S')
       using nd ‹L ∈# D› LM unfolding M by (auto simp add: lits-of-def)
     show ?thesis
       proof −
         have atm-of L ∈ atm-of ` lits-of-l M
           using ‹−L ∈ lits-of-l M›
           by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def)
         then have get-level (M @ trail S') L = backtrack-lvl S'
           using lev' LS' nm unfolding cdcl_W-M-level-inv-def by auto
         then show ?thesis
           using nm ‹L∈#D› ‹conflicting S'' = Some D›
           unfolding lits-of-def btS M
           by auto
       qed
   qed
 }
 ultimately have conflict-is-false-with-level S'' by blast
}
moreover
{
  assume conflicting S' ≠ None
  have no-clause-is-false S' using ‹conflicting S' ≠ None› by auto
  then have conflict-is-false-with-level S'' using calculation(3) by presburger
}
```

ultimately show *?case* **by** *blast*
qed

lemma *rtranclp-cdcl$_W$-stgy-no-smaller-confl-inv*:
assumes
cdcl$_W$-stgy** S S' **and**
n-l: *no-smaller-confl S* **and**
cls-false: *conflict-is-false-with-level S* **and**
lev: *cdcl$_W$-M-level-inv S* **and**
no-f: *no-clause-is-false S* **and**
dist: *distinct-cdcl$_W$-state S* **and**
conflicting: *cdcl$_W$-conflicting S* **and**
decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**
learned: *cdcl$_W$-learned-clause S* **and**
alien: *no-strange-atm S*
shows *no-smaller-confl S'* $\wedge$ *conflict-is-false-with-level S'*
using *assms*(*1*)
proof (*induct rule*: *rtranclp-induct*)
case *base*
then show *?case* **using** *n-l cls-false* **by** *auto*
next
case (*step S' S''*) **note** *st* = *this*(*1*) **and** *cdcl* = *this*(*2*) **and** *IH* = *this*(*3*)
have *no-smaller-confl S'* **and** *conflict-is-false-with-level S'*
using *IH* **by** *blast+*
moreover have *cdcl$_W$-M-level-inv S'*
using *st lev rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$*
by (*blast intro*: *rtranclp-cdcl$_W$-consistent-inv*)+
moreover have *no-clause-is-false S'*
using *st no-f rtranclp-cdcl$_W$-stgy-not-non-negated-init-clss* **by** *presburger*
moreover have *distinct-cdcl$_W$-state S'*
using *rtanclp-distinct-cdcl$_W$-state-inv*[*of S S'*] *lev rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$*[*OF st*]
dist **by** *auto*
moreover have *cdcl$_W$-conflicting S'*
using *rtranclp-cdcl$_W$-all-inv*(*6*)[*of S S'*] *st alien conflicting decomp dist learned lev*
rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$ **by** *blast*
ultimately show *?case*
using *cdcl$_W$-stgy-no-smaller-confl*[*OF cdcl*] *cdcl$_W$-stgy-ex-lit-of-max-level*[*OF cdcl*] **by** *fast*
qed

Final States are Conclusive

lemma *full-cdcl$_W$-stgy-final-state-conclusive-non-false*:
fixes *S'* :: *'st*
assumes *full*: *full cdcl$_W$-stgy* (*init-state N*) *S'*
and *no-d*: *distinct-mset-mset N*
and *no-empty*: $\forall$ *D*$\in$#*N*. *D* $\neq$ {#}
shows (*conflicting S'* = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss S'*)))
$\vee$ (*conflicting S'* = *None* $\wedge$ *trail S'* $\models$*asm init-clss S'*)
proof −
let *?S* = *init-state N*
have
termi: $\forall$ *S''*. $\neg$*cdcl$_W$-stgy S' S''* **and**
step: *cdcl$_W$-stgy** ?S S'* **using** *full* **unfolding** *full-def* **by** *auto*
moreover have
learned: *cdcl$_W$-learned-clause S'* **and**
level-inv: *cdcl$_W$-M-level-inv S'* **and**

alien: *no-strange-atm* S' **and**
no-dup: *distinct-cdcl*$_W$*-state* S' **and**
confl: *cdcl*$_W$*-conflicting* S' **and**
decomp: *all-decomposition-implies-m* (*init-clss* S') (*get-all-ann-decomposition* (*trail* S'))
using *no-d tranclp-cdcl*$_W$*-stgy-tranclp-cdcl*$_W$[*of ?S* S'] *step rtranclp-cdcl*$_W$*-all-inv*(*1*−*6*)[*of ?S* S']
unfolding *rtranclp-unfold* **by** *auto*
moreover
have ∀ *D*∈#*N*. ¬ [] ⊨*as CNot D* **using** *no-empty* **by** *auto*
then have *confl-k*: *conflict-is-false-with-level* S'
using *rtranclp-cdcl*$_W$*-stgy-no-smaller-confl-inv*[*OF step*] *no-d* **by** *auto*
show *?thesis*
using *cdcl*$_W$*-stgy-final-state-conclusive*[*OF termi decomp learned level-inv alien no-dup confl confl-k*] **.**
qed

lemma *conflict-is-full1-cdcl*$_W$*-cp*:
assumes *cp*: *conflict S* S'
shows *full1 cdcl*$_W$*-cp S* S'
proof −
have *cdcl*$_W$*-cp S* S' **and** *conflicting* $S' \neq$ *None*
using *cp cdcl*$_W$*-cp.intros* **by** (*auto elim*!: *conflictE simp*: *state-eq-def simp del*: *state-simp*)
then have *cdcl*$_W$*-cp*$^{++}$ *S* S' **by** *blast*
moreover have *no-step cdcl*$_W$*-cp* S'
using ‹*conflicting* $S' \neq$ *None*› **by** (*metis cdcl*$_W$*-cp-conflicting-not-empty option.exhaust*)
ultimately show *full1 cdcl*$_W$*-cp S* S' **unfolding** *full1-def* **by** *blast+*
qed

lemma *cdcl*$_W$*-cp-fst-empty-conflicting-false*:
assumes
cdcl$_W$*-cp S* S' **and**
trail S = [] **and**
conflicting S ≠ *None*
shows *False*
using *assms* **by** (*induct rule*: *cdcl*$_W$*-cp.induct*) (*auto elim*: *propagateE conflictE*)

lemma *cdcl*$_W$*-o-fst-empty-conflicting-false*:
assumes *cdcl*$_W$*-o S* S'
and *trail S* = []
and *conflicting S* ≠ *None*
shows *False*
using *assms* **by** (*induct rule*: *cdcl*$_W$*-o-induct*) *auto*

lemma *cdcl*$_W$*-stgy-fst-empty-conflicting-false*:
assumes *cdcl*$_W$*-stgy S* S'
and *trail S* = []
and *conflicting S* ≠ *None*
shows *False*
using *assms* **apply** (*induct rule*: *cdcl*$_W$*-stgy.induct*)
using *tranclpD cdcl*$_W$*-cp-fst-empty-conflicting-false* **unfolding** *full1-def* **apply** *metis*
using *cdcl*$_W$*-o-fst-empty-conflicting-false* **by** *blast*
thm *cdcl*$_W$*-cp.induct*[*split-format*(*complete*)]

lemma *cdcl*$_W$*-cp-conflicting-is-false*:
cdcl$_W$*-cp S* S' ⟹ *conflicting S* = *Some* {#} ⟹ *False*

by (*induction rule*: $cdcl_W$*-cp.induct*) (*auto elim*: *propagateE conflictE*)

lemma *rtranclp-*$cdcl_W$*-cp-conflicting-is-false*:
$cdcl_W$*-cp*$^{++}$ $S\ S' \Longrightarrow$ *conflicting* S = *Some* {#} $\Longrightarrow$ *False*
apply (*induction rule*: *tranclp.induct*)
by (*auto dest*: $cdcl_W$*-cp-conflicting-is-false*)

lemma $cdcl_W$*-o-conflicting-is-false*:
$cdcl_W$*-o* $S\ S' \Longrightarrow$ *conflicting* S = *Some* {#} $\Longrightarrow$ *False*
by (*induction rule*: $cdcl_W$*-o-induct*) *auto*

lemma $cdcl_W$*-stgy-conflicting-is-false*:
$cdcl_W$*-stgy* $S\ S' \Longrightarrow$ *conflicting* S = *Some* {#} $\Longrightarrow$ *False*
apply (*induction rule*: $cdcl_W$*-stgy.induct*)
unfolding *full1-def* **apply** (*metis* (*no-types*) $cdcl_W$*-cp-conflicting-not-empty tranclpD*)
unfolding *full-def* **by** (*metis conflict-with-false-implies-terminated other*)

lemma *rtranclp-*$cdcl_W$*-stgy-conflicting-is-false*:
$cdcl_W$*-stgy*** $S\ S' \Longrightarrow$ *conflicting* S = *Some* {#} $\Longrightarrow S' = S$
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
using $cdcl_W$*-stgy-conflicting-is-false* **by** *blast*

lemma *full-*$cdcl_W$*-init-clss-with-false-normal-form*:
assumes
$\forall\ m \in$ *set* M. $\neg$*is-decided* m **and**
E = *Some* D **and**
state $S = (M, N, U, 0, E)$
full $cdcl_W$*-stgy* $S\ S'$ **and**
all-decomposition-implies-m (*init-clss* S) (*get-all-ann-decomposition* (*trail* S))
$cdcl_W$*-learned-clause* S
$cdcl_W$*-M-level-inv* S
no-strange-atm S
distinct-$cdcl_W$*-state* S
$cdcl_W$*-conflicting* S
shows $\exists M''$. *state* $S' = (M'', N, U, 0,$ *Some* {#})
using *assms*(*10*,*9*,*8*,*7*,*6*,*5*,*4*,*3*,*2*,*1*)
proof (*induction M arbitrary*: *E D S*)
case *Nil*
then show *?case*
using *rtranclp-*$cdcl_W$*-stgy-conflicting-is-false* **unfolding** *full-def* $cdcl_W$*-conflicting-def*
by *fastforce*
next
case (*Cons L M*) **note** *IH* = *this*(*1*) **and** *full* = *this*(*8*) **and** *E* = *this*(*10*) **and** *inv* = *this*(*2*$-$*7*) **and**
S = *this*(*9*) **and** *nm* = *this*(*11*)
obtain *K p* **where** *K*: *L* = *Propagated K p*
using *nm* **by** (*cases L*) *auto*
have *every-mark-is-a-conflict S* **using** *inv* **unfolding** $cdcl_W$*-conflicting-def* **by** *auto*
then have *MpK*: $M \models as$ *CNot* (p − {#K#}) **and** *Kp*: $K \in\#\ p$
using *S* **unfolding** *K* **by** *fastforce+*
then have *p*: $p = (p - \{\#K\#\}) + \{\#K\#\}$
by (*auto simp add*: *multiset-eq-iff*)
then have *K'*: *L* = *Propagated K* ((p − {#K#}) + {#K#})
using *K* **by** *auto*
obtain p' **where**
p': *hd-trail S* = *Propagated K* p' **and**

pp': $p' = p$
using *S K* **by** (*cases hd-trail S*) *auto*
have *conflicting S = Some D*
using *S E* **by** (*cases conflicting S*) *auto*
then have *DD*: *D = D*
using *S E* **by** *auto*
consider (*D*) *D* = {#} | (*D'*) *D* ≠ {#} **by** *blast*
then show *?case*
proof *cases*
case *D*
then show *?thesis*
using *full rtranclp-cdcl$_W$-stgy-conflicting-is-false S* **unfolding** *full-def E D* **by** *auto*
next
case *D'*
then have *no-p*: *no-step propagate S* **and** *no-c*: *no-step conflict S*
using *S E* **by** (*auto elim*: *propagateE conflictE*)
then have *no-step cdcl$_W$-cp S* **by** (*auto simp*: *cdcl$_W$-cp.simps*)
have *res-skip*: ∃ *T*. (*resolve S T* ∧ *no-step skip S* ∧ *full cdcl$_W$-cp T T*)
∨ (*skip S T* ∧ *no-step resolve S* ∧ *full cdcl$_W$-cp T T*)
proof *cases*
assume −*lit-of L* ∉# *D*
then obtain *T* **where** *sk*: *skip S T*
using *S D' K skip-rule* **unfolding** *E* **by** *fastforce*
then have *res*: *no-step resolve S*
using ‹−*lit-of L* ∉# *D*› *S D' K* **unfolding** *E*
by (*auto elim*!: *skipE resolveE*)
have *full cdcl$_W$-cp T T*
using *sk* **by** (*auto intro*!: *option-full-cdcl$_W$-cp elim*: *skipE*)
then show *?thesis*
using *sk res* **by** *blast*
next
assume *LD*: ¬−*lit-of L* ∉# *D*
then have *D*: *Some D = Some* ((*D* − {#−*lit-of L*#}) + {#−*lit-of L*#})
by (*auto simp add*: *multiset-eq-iff*)

have ⋀*L*. *get-level M L = 0*
by (*simp add*: *nm*)
then have *get-maximum-level* (*Propagated K* (*p* − {#*K*#} + {#*K*#}) # *M*) (*D* − {#−
K#}) = *0*
using *LD get-maximum-level-exists-lit-of-max-level*
proof −
obtain *L'* **where** *get-level* (*L*#*M*) *L'* = *get-maximum-level* (*L*#*M*) *D*
using *LD get-maximum-level-exists-lit-of-max-level*[*of D L*#*M*] **by** *fastforce*
then show *?thesis* **by** (*metis* (*mono-tags*) *K' get-level-skip-all-not-decided*
get-maximum-level-exists-lit nm not-gr0)
qed
then obtain *T* **where** *sk*: *resolve S T*
using *resolve-rule*[*of S K p' D*] *S p'* ‹*K* ∈# *p*› *D LD*
unfolding *K' D E pp'* **by** *auto*
then have *res*: *no-step skip S*
using *LD S D' K* **unfolding** *E*
by (*auto elim*!: *skipE resolveE*)
have *full cdcl$_W$-cp T T*
using *sk* **by** (*auto simp*: *option-full-cdcl$_W$-cp elim*: *resolveE*)
then show *?thesis*
using *sk res* **by** *blast*

qed
then have *step-s*: $\exists$ *T*. *cdcl*$_W$*-stgy S T*
using ‹*no-step cdcl*$_W$*-cp S*› *other*′ **by** (*meson bj resolve skip*)
have *get-all-ann-decomposition* (*L* # *M*) = [([], *L*#*M*)]
using *nm* **unfolding** *K* **apply** (*induction M rule*: *ann-lit-list-induct*, *simp*)
by (*rename-tac L xs*, *case-tac hd* (*get-all-ann-decomposition xs*), *auto*)+
then have *no-b*: *no-step backtrack S*
using *nm S* **by** (*auto elim*: *backtrackE*)
have *no-d*: *no-step decide S*
using *S E* **by** (*auto elim*: *decideE*)

have *full-S-S*: *full cdcl*$_W$*-cp S S*
using *S E* **by** (*auto simp add*: *option-full-cdcl*$_W$*-cp*)
then have *no-f*: *no-step* (*full1 cdcl*$_W$*-cp*) *S*
unfolding *full-def full1-def rtranclp-unfold* **by** (*meson tranclpD*)
obtain *T* **where**
s: *cdcl*$_W$*-stgy S T* **and** *st*: *cdcl*$_W$*-stgy*** *T S*′
using *full step-s full* **unfolding** *full-def* **by** (*metis rtranclp-unfold tranclpD*)
have *resolve S T* $\vee$ *skip S T*
using *s no-b no-d res-skip full-S-S cdcl*$_W$*-cp-state-eq-compatible resolve-unique skip-unique* **unfolding** *cdcl*$_W$*-stgy.simps cdcl*$_W$*-o.simps full-unfold full1-def* **by** (*blast dest*!: *tranclpD elim*!: *cdcl*$_W$*-bj.cases*)+
then obtain *D*′ **where** *T*: *state T* = (*M*, *N*, *U*, *0*, *Some D*′)
using *S E* **by** (*auto elim*!: *skipE resolveE simp*: *state-eq-def simp del*: *state-simp*)

have *st-c*: *cdcl*$_W$** *S T*
using *E T rtranclp-cdcl*$_W$*-stgy-rtranclp-cdcl*$_W$ *s* **by** *blast*
have *cdcl*$_W$*-conflicting T*
using *rtranclp-cdcl*$_W$*-all-inv*(*6*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **.**
show *?thesis*
apply (*rule IH*[*of T*])
using *rtranclp-cdcl*$_W$*-all-inv*(*6*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **apply** *blast*
using *rtranclp-cdcl*$_W$*-all-inv*(*5*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **apply** *blast*
using *rtranclp-cdcl*$_W$*-all-inv*(*4*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **apply** *blast*
using *rtranclp-cdcl*$_W$*-all-inv*(*3*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **apply** *blast*
using *rtranclp-cdcl*$_W$*-all-inv*(*2*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **apply** *blast*
using *rtranclp-cdcl*$_W$*-all-inv*(*1*)[*OF st-c inv*(*6*,*5*,*4*,*3*,*2*,*1*)] **apply** *blast*
apply (*metis full-def st full*)
using *T E* **apply** *blast*
apply *auto*[]
using *nm* **by** *simp*
qed
qed

lemma *full-cdcl*$_W$*-stgy-final-state-conclusive-is-one-false*:
fixes *S*′ :: ′*st*
assumes *full*: *full cdcl*$_W$*-stgy* (*init-state N*) *S*′
and *no-d*: *distinct-mset-mset N*
and *empty*: {#} $\in$# *N*
shows *conflicting S*′ = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss S*′))
proof −
let *?S* = *init-state N*
have *cdcl*$_W$*-stgy*** *?S S*′ **and** *no-step cdcl*$_W$*-stgy S*′ **using** *full* **unfolding** *full-def* **by** *auto*
then have *plus-or-eq*: *cdcl*$_W$*-stgy*$^{++}$ *?S S*′ $\vee$ *S*′ = *?S* **unfolding** *rtranclp-unfold* **by** *auto*
have $\exists$ *S*″. *conflict ?S S*″
using *empty not-conflict-not-any-negated-init-clss*[*of init-state N*] **by** *auto*

then have *cdcl*$_W$*-stgy*: $\exists$ *S'*. *cdcl*$_W$*-stgy ?S S'*
  using *cdcl*$_W$*-cp.conflict'*[*of ?S*] *conflict-is-full1-cdcl*$_W$*-cp cdcl*$_W$*-stgy.intros*(*1*) **by** *metis*
have *S'* $\neq$ *?S* **using** ‹*no-step cdcl*$_W$*-stgy S'*› *cdcl*$_W$*-stgy* **by** *blast*

then obtain *St* :: *'st* **where** *St*: *cdcl*$_W$*-stgy ?S St* **and** *cdcl*$_W$*-stgy*** *St S'*
  using *plus-or-eq* **by** (*metis* (*no-types*) ‹*cdcl*$_W$*-stgy*** *?S S'*› *converse-rtranclpE*)
have *st*: *cdcl*$_W$** *?S St*
  by (*simp add*: *rtranclp-unfold* ‹*cdcl*$_W$*-stgy ?S St*› *cdcl*$_W$*-stgy-tranclp-cdcl*$_W$)

have $\exists$ *T*. *conflict ?S T*
  using *empty not-conflict-not-any-negated-init-clss*[*of ?S*] **by** *force*
then have *fullSt*: *full1 cdcl*$_W$*-cp ?S St*
  using *St* **unfolding** *cdcl*$_W$*-stgy.simps* **by** *blast*
then have *bt*: *backtrack-lvl St* = (*0*::*nat*)
  using *rtranclp-cdcl*$_W$*-cp-backtrack-lvl* **unfolding** *full1-def*
  by (*fastforce dest*!: *tranclp-into-rtranclp*)
have *cls-St*: *init-clss St* = *N*
  using *fullSt cdcl*$_W$*-stgy-no-more-init-clss*[*OF St*] **by** *auto*
have *conflicting St* $\neq$ *None*
  proof (*rule ccontr*)
    assume *conf*: $\neg$ *?thesis*
    obtain *E* **where**
      ES: *E* $\in$# *init-clss St* **and**
      E: *E* = {#}
      using *empty cls-St* **by** *metis*
    then have $\exists$ *T*. *conflict St T*
      using *empty cls-St conflict-rule*[*of St E*] *ES conf* **unfolding** *E*
      by (*auto simp*: *clauses-def dest*:)
    then show *False* **using** *fullSt* **unfolding** *full1-def* **by** *blast*
  qed

have *1*: $\forall$ *m*$\in$*set* (*trail St*). $\neg$ *is-decided m*
  using *fullSt* **unfolding** *full1-def* **by** (*auto dest*!: *tranclp-into-rtranclp*
    rtranclp-cdcl$_W$*-cp-dropWhile-trail*)
have *2*: *full cdcl*$_W$*-stgy St S'*
  using ‹*cdcl*$_W$*-stgy*** *St S'*› ‹*no-step cdcl*$_W$*-stgy S'*› *bt* **unfolding** *full-def* **by** *auto*
have *3*: *all-decomposition-implies-m*
    (*init-clss St*)
    (*get-all-ann-decomposition*
      (*trail St*))
 using *rtranclp-cdcl*$_W$*-all-inv*(*1*)[*OF st*] *no-d bt* **by** *simp*
have *4*: *cdcl*$_W$*-learned-clause St*
  using *rtranclp-cdcl*$_W$*-all-inv*(*2*)[*OF st*] *no-d bt bt* **by** *simp*
have *5*: *cdcl*$_W$*-M-level-inv St*
  using *rtranclp-cdcl*$_W$*-all-inv*(*3*)[*OF st*] *no-d bt* **by** *simp*
have *6*: *no-strange-atm St*
  using *rtranclp-cdcl*$_W$*-all-inv*(*4*)[*OF st*] *no-d bt* **by** *simp*
have *7*: *distinct-cdcl*$_W$*-state St*
  using *rtranclp-cdcl*$_W$*-all-inv*(*5*)[*OF st*] *no-d bt* **by** *simp*
have *8*: *cdcl*$_W$*-conflicting St*
  using *rtranclp-cdcl*$_W$*-all-inv*(*6*)[*OF st*] *no-d bt* **by** *simp*
have *init-clss S'* = *init-clss St* **and** *conflicting S'* = *Some* {#}
   using ‹*conflicting St* $\neq$ *None*› *full-cdcl*$_W$*-init-clss-with-false-normal-form*[*OF 1*, *of - - St*]
   2 3 4 5 6 7 8 St **apply** (*metis* ‹*cdcl*$_W$*-stgy*** *St S'*› *rtranclp-cdcl*$_W$*-stgy-no-more-init-clss*)

using ⟨*conflicting St* $\neq$ *None*⟩ *full-cdcl*$_W$*-init-clss-with-false-normal-form*[*OF 1*, *of - - St - - S'*] *2 3 4 5 6 7 8* **by** (*metis bt option.exhaust prod.inject*)

moreover have *init-clss S'* = *N*
using ⟨*cdcl*$_W$*-stgy*** (*init-state N*) *S'*⟩ *rtranclp-cdcl*$_W$*-stgy-no-more-init-clss* **by** *fastforce*
moreover have *unsatisfiable* (*set-mset N*)
by (*meson empty satisfiable-def true-cls-empty true-clss-def*)
ultimately show *?thesis* **by** *auto*
qed

theorem 2.9.9 page 83 of Weidenbach's book

lemma *full-cdcl*$_W$*-stgy-final-state-conclusive*:
fixes *S'* :: *'st*
assumes *full*: *full cdcl*$_W$*-stgy* (*init-state N*) *S'* **and** *no-d*: *distinct-mset-mset N*
shows (*conflicting S'* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S'*)))
∨ (*conflicting S'* = *None* ∧ *trail S'* ⊨*asm init-clss S'*)
using *assms full-cdcl*$_W$*-stgy-final-state-conclusive-is-one-false*
full-cdcl$_W$*-stgy-final-state-conclusive-non-false* **by** *blast*

theorem 2.9.9 page 83 of Weidenbach's book

lemma *full-cdcl*$_W$*-stgy-final-state-conclusive-from-init-state*:
fixes *S'* :: *'st*
assumes *full*: *full cdcl*$_W$*-stgy* (*init-state N*) *S'*
and *no-d*: *distinct-mset-mset N*
shows (*conflicting S'* = *Some* {#} ∧ *unsatisfiable* (*set-mset N*))
∨ (*conflicting S'* = *None* ∧ *trail S'* ⊨*asm N* ∧ *satisfiable* (*set-mset N*))
proof −
have *N*: *init-clss S'* = *N*
using *full* **unfolding** *full-def* **by** (*auto dest*: *rtranclp-cdcl*$_W$*-stgy-no-more-init-clss*)
consider
(*confl*) *conflicting S'* = *Some* {#} **and** *unsatisfiable* (*set-mset* (*init-clss S'*))
| (*sat*) *conflicting S'* = *None* **and** *trail S'* ⊨*asm init-clss S'*
using *full-cdcl*$_W$*-stgy-final-state-conclusive*[*OF assms*] **by** *auto*
then show *?thesis*
proof *cases*
case *confl*
then show *?thesis* **by** (*auto simp*: *N*)
next
case *sat*
have *cdcl*$_W$*-M-level-inv* (*init-state N*) **by** *auto*
then have *cdcl*$_W$*-M-level-inv S'*
using *full rtranclp-cdcl*$_W$*-stgy-consistent-inv* **unfolding** *full-def* **by** *blast*
then have *consistent-interp* (*lits-of-l* (*trail S'*)) **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *blast*
moreover have *lits-of-l* (*trail S'*) ⊨*s set-mset* (*init-clss S'*)
using *sat*(*2*) **by** (*auto simp add*: *true-annots-def true-annot-def true-clss-def*)
ultimately have *satisfiable* (*set-mset* (*init-clss S'*)) **by** *simp*
then show *?thesis* **using** *sat* **unfolding** *N* **by** *blast*
qed
qed

end
end
theory *CDCL-W-Termination*
imports *CDCL-W*
begin

context *conflict-driven-clause-learning*$_W$
begin

6.1.6 Termination

The condition that no learned clause is a tautology is overkill (in the sense that the no-duplicate condition is enough), but we can reuse *simple-clss*.

The invariant contains all the structural invariants that holds,

definition $cdcl_W$*-all-struct-inv* **where**
  $cdcl_W$*-all-struct-inv S* $\longleftrightarrow$
    no-strange-atm S $\wedge$
    $cdcl_W$*-M-level-inv S* $\wedge$
    ($\forall$ *s* $\in$# *learned-clss S*. $\neg$*tautology s*) $\wedge$
    distinct-$cdcl_W$*-state S* $\wedge$
    $cdcl_W$*-conflicting S* $\wedge$
    all-decomposition-implies-m (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) $\wedge$
    $cdcl_W$*-learned-clause S*

lemma $cdcl_W$*-all-struct-inv-inv*:
  assumes $cdcl_W$ *S S'* **and** $cdcl_W$*-all-struct-inv S*
  shows $cdcl_W$*-all-struct-inv S'*
  unfolding $cdcl_W$*-all-struct-inv-def*
proof (*intro HOL.conjI*)
  show *no-strange-atm S'*
    using $cdcl_W$*-all-inv*[*OF assms*(*1*)] *assms*(*2*) **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *auto*
  show $cdcl_W$*-M-level-inv S'*
    using $cdcl_W$*-all-inv*[*OF assms*(*1*)] *assms*(*2*) **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *fast*
  show *distinct-*$cdcl_W$*-state S'*
     using $cdcl_W$*-all-inv*[*OF assms*(*1*)] *assms*(*2*) **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *fast*
  show $cdcl_W$*-conflicting S'*
     using $cdcl_W$*-all-inv*[*OF assms*(*1*)] *assms*(*2*) **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *fast*
  show *all-decomposition-implies-m* (*init-clss S'*) (*get-all-ann-decomposition* (*trail S'*))
     using $cdcl_W$*-all-inv*[*OF assms*(*1*)] *assms*(*2*) **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *fast*
  show $cdcl_W$*-learned-clause S'*
     using $cdcl_W$*-all-inv*[*OF assms*(*1*)] *assms*(*2*) **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *fast*

  show $\forall$ *s*$\in$#*learned-clss S'*. $\neg$ *tautology s*
    using *assms*(*1*)[*THEN learned-clss-are-not-tautologies*] *assms*(*2*)
    unfolding $cdcl_W$*-all-struct-inv-def* **by** *fast*
qed

lemma *rtranclp-*$cdcl_W$*-all-struct-inv-inv*:
  assumes $cdcl_W^{**}$ *S S'* **and** $cdcl_W$*-all-struct-inv S*
  shows $cdcl_W$*-all-struct-inv S'*
  using *assms* **by** *induction* (*auto intro*: $cdcl_W$*-all-struct-inv-inv*)

lemma $cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*:
  $cdcl_W$*-stgy S T* $\Longrightarrow$ $cdcl_W$*-all-struct-inv S* $\Longrightarrow$ $cdcl_W$*-all-struct-inv T*
  by (*meson* $cdcl_W$*-stgy-tranclp-*$cdcl_W$ *rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-unfold*)

lemma *rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*:
  $cdcl_W$*-stgy*** *S T* $\Longrightarrow$ $cdcl_W$*-all-struct-inv S* $\Longrightarrow$ $cdcl_W$*-all-struct-inv T*
  by (*induction rule*: *rtranclp-induct*) (*auto intro*: $cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*)

No Relearning of a clause

lemma $cdcl_W$*-o-new-clause-learned-is-backtrack-step*:
assumes *learned*: $D \in\# $ *learned-clss* T **and**
new: $D \notin\#$ *learned-clss* S **and**
$cdcl_W$: $cdcl_W$*-o* S T **and**
lev: $cdcl_W$*-M-level-inv* S
shows *backtrack* S T $\wedge$ *conflicting* S = *Some* D
using $cdcl_W$ *lev learned new*
proof (*induction rule*: $cdcl_W$*-o-induct*)
case (*backtrack L C K i M1 M2 T*) **note** *decomp* = *this*(3) **and** *undef* = *this*(6) **and** T = *this*(8) **and**
D-T = *this*(10) **and** *D-S* = *this*(11)
then have $D = C$
using *not-gr0 lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
then show *?case*
using T *backtrack.hyps*(1−5) *backtrack.intros*[*OF backtrack.hyps*(1,2)] *backtrack.hyps*(3−7)
by *auto*
qed *auto*

lemma $cdcl_W$*-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: $D \in\#$ *learned-clss* T **and**
new: $D \notin\#$ *learned-clss* S **and**
$cdcl_W$: $cdcl_W$*-stgy* S T **and**
lev: $cdcl_W$*-M-level-inv* S
shows $\exists S'$. *backtrack* S S' $\wedge$ $cdcl_W$*-stgy*** S' T $\wedge$ *conflicting* S = *Some* D
using $cdcl_W$ *learned new*
proof (*induction rule*: $cdcl_W$*-stgy.induct*)
case (*conflict'* S')
then show *?case*
unfolding *full1-def* **by** (*metis* (*mono-tags*, *lifting*) *rtranclp-*$cdcl_W$*-cp-learned-clause-inv*
tranclp-into-rtranclp)
next
case (*other'* S' S'')
then have $D \in\#$ *learned-clss* S'
unfolding *full-def* **by** (*auto dest*: *rtranclp-*$cdcl_W$*-cp-learned-clause-inv*)
then show *?case*
using $cdcl_W$*-o-new-clause-learned-is-backtrack-step*[*OF* - ‹$D \notin\#$ *learned-clss* S› ‹$cdcl_W$*-o* S S'›]
‹*full* $cdcl_W$*-cp* S' S''› *lev* **by** (*metis* $cdcl_W$*-stgy.conflict'* *full-unfold r-into-rtranclp*
rtranclp.rtrancl-refl)
qed

lemma *rtranclp-*$cdcl_W$*-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: $D \in\#$ *learned-clss* T **and**
new: $D \notin\#$ *learned-clss* S **and**
$cdcl_W$: $cdcl_W$*-stgy*** S T **and**
lev: $cdcl_W$*-M-level-inv* S
shows $\exists S'$ S''. $cdcl_W$*-stgy*** S S' $\wedge$ *backtrack* S' S'' $\wedge$ *conflicting* S' = *Some* D $\wedge$
$cdcl_W$*-stgy*** S'' T
using $cdcl_W$ *learned new*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *blast*
next
case (*step T U*) **note** *st* = *this*(1) **and** *o* = *this*(2) **and** *IH* = *this*(3) **and**
D-U = *this*(4) **and** *D-S* = *this*(5)

show *?case*
proof (*cases D* $\in\#$ *learned-clss T*)
case *True*
then obtain S' S'' **where**
st': *cdcl*$_W$*-stgy*** *S* S' **and**
bt: *backtrack* S' S'' **and**
confl: *conflicting* S' = *Some D* **and**
st'': *cdcl*$_W$*-stgy*** S'' *T*
using *IH D-S* **by** *metis*
have *cdcl*$_W$*-stgy*$^{++}$ S'' *U*
using *st''* *o* **by** *force*
then show *?thesis*
by (*meson bt confl rtranclp-unfold st'*)
next
case *False*
have *cdcl*$_W$*-M-level-inv T*
using *lev rtranclp-cdcl*$_W$*-stgy-consistent-inv st* **by** *blast*
then obtain S' **where**
bt: *backtrack T* S' **and**
st': *cdcl*$_W$*-stgy*** S' *U* **and**
confl: *conflicting T* = *Some D*
using *cdcl*$_W$*-cp-new-clause-learned-has-backtrack-step*[*OF D-U False o*]
by *metis*
then have *cdcl*$_W$*-stgy*** *S T* **and**
backtrack T S' **and**
conflicting T = *Some D* **and**
cdcl$_W$*-stgy*** S' *U*
using *o st* **by** *auto*
then show *?thesis* **by** *blast*
qed
qed

lemma *propagate-no-more-Decided-lit*:
assumes *propagate S* S'
shows *Decided K* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K* $\in$ *set* (*trail* S')
using *assms* **by** (*auto elim*: *propagateE*)

lemma *conflict-no-more-Decided-lit*:
assumes *conflict S* S'
shows *Decided K* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K* $\in$ *set* (*trail* S')
using *assms* **by** (*auto elim*: *conflictE*)

lemma *cdcl*$_W$*-cp-no-more-Decided-lit*:
assumes *cdcl*$_W$*-cp S* S'
shows *Decided K* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K* $\in$ *set* (*trail* S')
using *assms* **apply** (*induct rule*: *cdcl*$_W$*-cp.induct*)
using *conflict-no-more-Decided-lit propagate-no-more-Decided-lit* **by** *auto*

lemma *rtranclp-cdcl*$_W$*-cp-no-more-Decided-lit*:
assumes *cdcl*$_W$*-cp*** *S* S'
shows *Decided K* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K* $\in$ *set* (*trail* S')
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
using *cdcl*$_W$*-cp-no-more-Decided-lit* **by** *blast+*

lemma *cdcl*$_W$*-o-no-more-Decided-lit*:
assumes *cdcl*$_W$*-o S* S' **and** *lev*: *cdcl*$_W$*-M-level-inv S* **and** $\neg$*decide S* S'

shows *Decided K* $\in$ *set* (*trail S'*) $\longrightarrow$ *Decided K* $\in$ *set* (*trail S*)
using *assms*
proof (*induct rule*: $cdcl_W$*-o-induct*)
case *backtrack* **note** *decomp* = *this*(*3*) **and** *undef* = *this*(*8*) **and** *T* = *this*(*9*)
then show *?case* **using** *lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
next
case (*decide L T*)
then show *?case* **using** *decide-rule*[*OF decide.hyps*] **by** *blast*
qed *auto*

lemma $cdcl_W$*-new-decided-at-beginning-is-decide*:
assumes $cdcl_W$*-stgy S S'* **and**
lev: $cdcl_W$*-M-level-inv S* **and**
trail S' = *M'* @ *Decided L* # *M* **and**
trail S = *M*
shows $\exists$ *T*. *decide S T* $\wedge$ *no-step* $cdcl_W$*-cp S*
using *assms*
proof (*induct rule*: $cdcl_W$*-stgy.induct*)
case (*conflict' S'*) **note** *st* = *this*(*1*) **and** *no-dup* = *this*(*2*) **and** *S'* = *this*(*3*) **and** *S* = *this*(*4*)
have $cdcl_W$*-M-level-inv S'*
using *full1-*$cdcl_W$*-cp-consistent-inv no-dup st* **by** *blast*
then have *Decided L* $\in$ *set* (*trail S'*) **and** *Decided L* $\notin$ *set* (*trail S*)
using *no-dup* **unfolding** *S S'* $cdcl_W$*-M-level-inv-def* **by** (*auto simp add*: *rev-image-eqI*)
then have *False*
using *st rtranclp-*$cdcl_W$*-cp-no-more-Decided-lit*[*of S S'*]
unfolding *full1-def rtranclp-unfold* **by** *blast*
then show *?case* **by** *fast*
next
case (*other' T U*) **note** *o* = *this*(*1*) **and** *ns* = *this*(*2*) **and** *st* = *this*(*3*) **and** *no-dup* = *this*(*4*) **and**
S' = *this*(*5*) **and** *S* = *this*(*6*)
have $cdcl_W$*-M-level-inv U*
by (*metis* (*full-types*) *lev* $cdcl_W$*.simps* $cdcl_W$*-consistent-inv full-def o*
other'.hyps(*3*) *rtranclp-*$cdcl_W$*-cp-consistent-inv*)
then have *Decided L* $\in$ *set* (*trail U*) **and** *Decided L* $\notin$ *set* (*trail S*)
using *no-dup* **unfolding** *S S'* $cdcl_W$*-M-level-inv-def* **by** (*auto simp add*: *rev-image-eqI*)
then have *Decided L* $\in$ *set* (*trail T*)
using *st rtranclp-*$cdcl_W$*-cp-no-more-Decided-lit* **unfolding** *full-def* **by** *blast*
then show *?case*
using $cdcl_W$*-o-no-more-Decided-lit*[*OF o*] ‹*Decided L* $\notin$ *set* (*trail S*)› *ns lev* **by** *meson*
qed

lemma $cdcl_W$*-o-is-decide*:
assumes $cdcl_W$*-o S T* **and** *lev*: $cdcl_W$*-M-level-inv S*
trail T = *drop* (*length* M_0) *M'* @ *Decided L* # *H* @ *M***and**
$\neg$ ($\exists$ *M'*. *trail S* = *M'* @ *Decided L* # *H* @ *M*)
shows *decide S T*
using *assms*
proof (*induction rule*:$cdcl_W$*-o-induct*)
case (*backtrack L D K i M1 M2 T*)
then obtain *c* **where** *trail S* = *c* @ *M2* @ *Decided K* # *M1*
by *auto*
show *?case*
using *backtrack lev*
apply (*cases drop* (*length* M_0) *M'*)
apply (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
using ‹*trail S* = *c* @ *M2* @ *Decided K* # *M1*›

by (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
next
case *decide*
show *?case* **using** *decide-rule*[*of S*] *decide*(*1*−*4*) **by** *auto*
qed *auto*

lemma *rtranclp-cdcl*$_W$*-new-decided-at-beginning-is-decide*:
assumes *cdcl*$_W$*-stgy*** *R U* **and**
trail U = *M'* @ *Decided L* # *H* @ *M* **and**
trail R = *M* **and**
cdcl$_W$*-M-level-inv R*
shows
∃ *S T T'*. *cdcl*$_W$*-stgy*** *R S* ∧ *decide S T* ∧ *cdcl*$_W$*-stgy*** *T U* ∧ *cdcl*$_W$*-stgy*** *S U* ∧
no-step cdcl$_W$*-cp S* ∧ *trail T* = *Decided L* # *H* @ *M* ∧ *trail S* = *H* @ *M* ∧ *cdcl*$_W$*-stgy S T'* ∧
cdcl$_W$*-stgy*** *T' U*
using *assms*
proof (*induct arbitrary*: *M H M' i rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *s* = *this*(*2*) **and**
U = *this*(*4*) **and** *S* = *this*(*5*) **and** *lev* = *this*(*6*)
show *?case*
proof (*cases* ∃ *M'*. *trail T* = *M'* @ *Decided L* # *H* @ *M*)
case *False*
with *s* **show** *?thesis* **using** *U s st S*
proof *induction*
case (*conflict' W*) **note** *cp* = *this*(*1*) **and** *nd* = *this*(*2*) **and** *W* = *this*(*3*)
then obtain M_0 **where** *trail W* = M_0 @ *trail T* **and** *ndecided*: ∀ *l*∈*set* M_0. ¬ *is-decided l*
using *rtranclp-cdcl*$_W$*-cp-dropWhile-trail* **unfolding** *full1-def rtranclp-unfold* **by** *meson*
then have *MV*: *M'* @ *Decided L* # *H* @ *M* = M_0 @ *trail T* **unfolding** *W* **by** *simp*
then have *V*: *trail T* = *drop* (*length* M_0) (*M'* @ *Decided L* # *H* @ *M*)
by *auto*
have *takeWhile* (*Not o is-decided*) *M'* = M_0 @ *takeWhile* (*Not* ∘ *is-decided*) (*trail T*)
using *arg-cong*[*OF MV*, *of takeWhile* (*Not o is-decided*)] *ndecided*
by (*simp add*: *takeWhile-tail*)
from *arg-cong*[*OF this*, *of length*] **have** *length* M_0 ≤ *length M'*
unfolding *length-append* **by** (*metis* (*no-types*, *lifting*) *Nat.le-trans le-add1*
length-takeWhile-le)
then have *False* **using** *nd V* **by** *auto*
then show *?case* **by** *fast*
next
case (*other' T' U*) **note** *o* = *this*(*1*) **and** *ns* = *this*(*2*) **and** *cp* = *this*(*3*) **and** *nd* = *this*(*4*)
and *U* = *this*(*5*) **and** *st* = *this*(*6*)
obtain M_0 **where** *trail U* = M_0 @ *trail T'* **and** *ndecided*: ∀ *l*∈*set* M_0. ¬ *is-decided l*
using *rtranclp-cdcl*$_W$*-cp-dropWhile-trail cp* **unfolding** *full-def* **by** *meson*
then have *MV*: *M'* @ *Decided L* # *H* @ *M* = M_0 @ *trail T'* **unfolding** *U* **by** *simp*
then have *V*: *trail T'* = *drop* (*length* M_0) (*M'* @ *Decided L* # *H* @ *M*)
by *auto*
have *takeWhile* (*Not o is-decided*) *M'* = M_0 @ *takeWhile* (*Not* ∘ *is-decided*) (*trail T'*)
using *arg-cong*[*OF MV*, *of takeWhile* (*Not o is-decided*)] *ndecided*
by (*simp add*: *takeWhile-tail*)
from *arg-cong*[*OF this*, *of length*] **have** *length* M_0 ≤ *length M'*
unfolding *length-append* **by** (*metis* (*no-types*, *lifting*) *Nat.le-trans le-add1*
length-takeWhile-le)
then have *tr-T'*: *trail T'* = *drop* (*length* M_0) *M'* @ *Decided L* # *H* @ *M* **using** *V* **by** *auto*

then have *LT'*: *Decided L* ∈ *set* (*trail T'*) **by** *auto*
moreover
have $cdcl_W$*-M-level-inv T*
using *lev rtranclp-*$cdcl_W$*-stgy-consistent-inv step.hyps*(*1*) **by** *blast*
then have *decide T T'* **using** *o nd tr-T'* $cdcl_W$*-o-is-decide* **by** *metis*
ultimately have *decide T T'* **using** $cdcl_W$*-o-no-more-Decided-lit*[*OF o*] **by** *blast*
then have *1*: $cdcl_W$*-stgy*** *R T* **and** *2*: *decide T T'* **and** *3*: $cdcl_W$*-stgy*** *T' U*
using *st other'.prems*(*4*)
by (*metis* $cdcl_W$*-stgy.conflict' cp full-unfold r-into-rtranclp rtranclp.rtrancl-refl*)+
have [*simp*]: *drop* (*length* M_0) *M'* = []
using ‹*decide T T'*› ‹*Decided L* ∈ *set* (*trail T'*)› *nd tr-T'*
by (*auto simp add*: *Cons-eq-append-conv elim*: *decideE*)
have *T'*: *drop* (*length* M_0) *M'* @ *Decided L* # *H* @ *M* = *Decided L* # *trail T*
using ‹*decide T T'*› ‹*Decided L* ∈ *set* (*trail T'*)› *nd tr-T'*
by (*auto elim*: *decideE*)
have *trail T'* = *Decided L* # *trail T*
using ‹*decide T T'*› ‹*Decided L* ∈ *set* (*trail T'*)› *tr-T'*
by (*auto elim*: *decideE*)
then have *5*: *trail T'* = *Decided L* # *H* @ *M*
using *append.simps*(*1*) *list.sel*(*3*) *local.other'*(*5*) *tl-append2* **by** (*simp add*: *tr-T'*)
have *6*: *trail T* = *H* @ *M*
by (*metis* (*no-types*) ‹*trail T'* = *Decided L* # *trail T*›
‹*trail T'* = *drop* (*length* M_0) *M'* @ *Decided L* # *H* @ *M*› *append-Nil list.sel*(*3*) *nd*
tl-append2)
have *7*: $cdcl_W$*-stgy*** *T U* **using** *other'.prems*(*4*) *st* **by** *auto*
have *8*: $cdcl_W$*-stgy T U* $cdcl_W$*-stgy*** *U U*
using $cdcl_W$*-stgy.other'*[*OF other'*(*1*−*3*)] **by** *simp-all*
show *?case* **apply** (*rule exI*[*of - T*], *rule exI*[*of - T'*], *rule exI*[*of - U*])
using *ns 1 2 3 5 6 7 8* **by** *fast*
qed
next
case *True*
then obtain *M'* **where** *T*: *trail T* = *M'* @ *Decided L* # *H* @ *M* **by** *metis*
from *IH*[*OF this S lev*] **obtain** *S' S'' S'''* **where**
1: $cdcl_W$*-stgy*** *R S'* **and**
2: *decide S' S''* **and**
3: $cdcl_W$*-stgy*** *S'' T* **and**
4: *no-step* $cdcl_W$*-cp S'* **and**
6: *trail S''* = *Decided L* # *H* @ *M* **and**
7: *trail S'* = *H* @ *M* **and**
8: $cdcl_W$*-stgy*** *S' T* **and**
9: $cdcl_W$*-stgy S' S'''* **and**
10: $cdcl_W$*-stgy*** *S''' T*
by *blast*
have $cdcl_W$*-stgy*** *S'' U* **using** *s* ‹$cdcl_W$*-stgy*** *S'' T* › **by** *auto*
moreover have $cdcl_W$*-stgy*** *S' U* **using** *8 s* **by** *auto*
moreover have $cdcl_W$*-stgy*** *S''' U* **using** *10 s* **by** *auto*
ultimately show *?thesis* **apply** − **apply** (*rule exI*[*of - S'*], *rule exI*[*of - S''*])
using *1 2 4 6 7 8 9* **by** *blast*
qed
qed

lemma *rtranclp-*$cdcl_W$*-new-decided-at-beginning-is-decide'*:
assumes $cdcl_W$*-stgy*** *R U* **and**
trail U = *M'* @ *Decided L* # *H* @ *M* **and**
trail R = *M* **and**

cdcl$_W$*-M-level-inv R*
shows $\exists\, y\ y'.$ *cdcl*$_W$*-stgy*** *R y* $\wedge$ *cdcl*$_W$*-stgy y y'* $\wedge\ \neg\ (\exists\, c.$ *trail y* $=$ *c* @ *Decided L* # *H* @ *M*)
$\wedge$ ($\lambda a\ b.$ *cdcl*$_W$*-stgy a b* $\wedge$ ($\exists\, c.$ *trail a* $=$ *c* @ *Decided L* # *H* @ *M*))** *y' U*
proof –
fix *T'*
obtain *S' T T'* **where**
st: *cdcl*$_W$*-stgy*** *R S'* **and**
decide S' T **and**
TU: *cdcl*$_W$*-stgy*** *T U* **and**
no-step cdcl$_W$*-cp S'* **and**
trT: *trail T* = *Decided L* # *H* @ *M* **and**
trS': *trail S'* = *H* @ *M* **and**
S'U: *cdcl*$_W$*-stgy*** *S' U* **and**
S'T': *cdcl*$_W$*-stgy S' T'* **and**
T'U: *cdcl*$_W$*-stgy*** *T' U*
using *rtranclp-cdcl*$_W$*-new-decided-at-beginning-is-decide*[*OF assms*] **by** *blast*
have *n*: $\neg$ ($\exists\, c.$ *trail S'* = *c* @ *Decided L* # *H* @ *M*) **using** *trS'* **by** *auto*
show *?thesis*
using *rtranclp-trans*[*OF st*] *rtranclp-exists-last-with-prop*[*of cdcl*$_W$*-stgy S' T'* -
λa -. $\neg$($\exists\, c.$ *trail a* = *c* @ *Decided L* # *H* @ *M*), *OF S'T' T'U n*]
by *meson*
qed

lemma *beginning-not-decided-invert*:
assumes *A*: *M* @ *A* = *M'* @ *Decided K* # *H* **and**
nm: $\forall\, m \in$ *set M*. $\neg$*is-decided m*
shows $\exists\, M.$ *A* = *M* @ *Decided K* # *H*
proof –
have *A* = *drop* (*length M*) (*M'* @ *Decided K* # *H*)
using *arg-cong*[*OF A*, *of drop* (*length M*)] **by** *auto*
moreover have *drop* (*length M*) (*M'* @ *Decided K* # *H*) = *drop* (*length M*) *M'* @ *Decided K* # *H*
using *nm* **by** (*metis* (*no-types*, *lifting*) *A drop-Cons' drop-append ann-lit.disc*(*1*) *not-gr0*
nth-append nth-append-length nth-mem zero-less-diff)
finally show *?thesis* **by** *fast*
qed

lemma *cdcl*$_W$*-stgy-trail-has-new-decided-is-decide-step*:
assumes *cdcl*$_W$*-stgy S T*
$\neg$ ($\exists\, c.$ *trail S* = *c* @ *Decided L* # *H* @ *M*) **and**
($\lambda a\ b.$ *cdcl*$_W$*-stgy a b* $\wedge$ ($\exists\, c.$ *trail a* = *c* @ *Decided L* # *H* @ *M*))** *T U* **and**
$\exists\, M'.$ *trail U* = *M'* @ *Decided L* # *H* @ *M* **and**
lev: *cdcl*$_W$*-M-level-inv S*
shows $\exists\, S'.$ *decide S S'* $\wedge$ *full cdcl*$_W$*-cp S' T* $\wedge$ *no-step cdcl*$_W$*-cp S*
using *assms*(*3*,*1*,*2*,*4*,*5*)
proof *induction*
case (*step T U*)
then show *?case* **by** *fastforce*
next
case *base*
then show *?case*
proof (*induction rule*: *cdcl*$_W$*-stgy.induct*)
case (*conflict' T*) **note** *cp* = *this*(*1*) **and** *nd* = *this*(*2*) **and** *M'* = *this*(*3*) **and** *no-dup* = *this*(*3*)
then obtain *M'* **where** *M'*: *trail T* = *M'* @ *Decided L* # *H* @ *M* **by** *metis*
obtain *M''* **where** *M''*: *trail T* = *M''* @ *trail S* **and** *nm*: $\forall\, m \in$ *set M''*. $\neg$*is-decided m*
using *cp* **unfolding** *full1-def*
by (*metis rtranclp-cdcl*$_W$*-cp-dropWhile-trail' tranclp-into-rtranclp*)

have *False*
using *beginning-not-decided-invert*[*of* M'' *trail S* M' *L H* @ *M*] M' *nm nd* **unfolding** M''
by *fast*
then show *?case* **by** *fast*
next
case (*other'* *T* U') **note** *o* = *this*(*1*) **and** *ns* = *this*(*2*) **and** *cp* = *this*(*3*) **and** *nd* = *this*(*4*)
and trU' = *this*(*5*)
have $cdcl_W$*-cp*** *T* U' **using** *cp* **unfolding** *full-def* **by** *blast*
from *rtranclp-*$cdcl_W$*-cp-dropWhile-trail*[*OF this*]
have $\exists M'$. *trail T* = M' @ *Decided L* # *H* @ *M*
using trU' *beginning-not-decided-invert*[*of - trail T - L H* @ *M*] **by** *metis*
then obtain M' **where** M': *trail T* = M' @ *Decided L* # *H* @ *M*
by *auto*
with *o lev nd cp ns*
show *?case*
proof (*induction rule*: $cdcl_W$*-o-induct*)
case (*decide L*) **note** *dec* = *this*(*1*) **and** *cp* = *this*(*5*) **and** *ns* = *this*(*4*)
then have *decide S* (*cons-trail* (*Decided L*) (*incr-lvl S*))
using *decide.hyps decide.intros*[*of S*] **by** *force*
then show *?case* **using** *cp decide.prems* **by** (*meson decide-state-eq-compatible ns state-eq-ref state-eq-sym*)
next
case (*backtrack* L' *D K j M1 M2 T*) **note** *decomp* = *this*(*3*) **and** *undef* = *this*(*8*) **and**
T = *this*(*9*) **and** *trT* = *this*(*13*)
obtain *MS3* **where** *MS3*: *trail S* = *MS3* @ *M2* @ *Decided K* # *M1*
using *get-all-ann-decomposition-exists-prepend*[*OF decomp*] **by** *metis*
have *tl* (M' @ *Decided L* # *H* @ *M*) = *tl* M' @ *Decided L* # *H* @ *M*
using *lev trT T lev undef decomp* **by** (*cases* M') (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
then have M'': *M1* = *tl* M' @ *Decided L* # *H* @ *M*
using *arg-cong*[*OF trT*[*simplified*], *of tl*] *T decomp undef lev*
by (*simp add*: $cdcl_W$*-M-level-inv-decomp*)
have *False* **using** *nd MS3 T undef decomp* **unfolding** M'' **by** *auto*
then show *?case* **by** *fast*
qed *auto*
qed
qed

lemma *rtranclp-*$cdcl_W$*-stgy-with-trail-end-has-trail-end*:
assumes (λa *b*. $cdcl_W$*-stgy a b* $\wedge$ ($\exists c$. *trail a* = *c* @ *Decided L* # *H* @ *M*))** *T U* **and**
$\exists M'$. *trail U* = M' @ *Decided L* # *H* @ *M*
shows $\exists M'$. *trail T* = M' @ *Decided L* # *H* @ *M*
using *assms* **by** (*induction rule*: *rtranclp-induct*) *auto*

lemma *remove1-mset-eq-remove1-mset-same*:
remove1-mset L D = *remove1-mset* L' *D* $\Longrightarrow$ *L* $\in$# *D* $\Longrightarrow$ *L* = L'
by (*metis diff-single-trivial insert-DiffM multi-drop-mem-not-eq single-eq-single union-right-cancel*)

lemma $cdcl_W$*-o-cannot-learn*:
assumes
$cdcl_W$*-o y z* **and**
lev: $cdcl_W$*-M-level-inv y* **and**
M: *trail y* = *c* @ *Decided Kh* # *H* **and**
DL: *D* $\notin$# *learned-clss y* **and**
LD: *L* $\in$# *D* **and**
DH: *atms-of* (*remove1-mset L D*) $\subseteq$ *atm-of* ‘ *lits-of-l H* **and**

LH: *atm-of L* ∉ *atm-of* ‘ *lits-of-l H* **and**
learned: ∀ *T*. *conflicting y* = *Some T* ⟶ *trail y* ⊨*as CNot T* **and**
z: *trail z* = *c′* @ *Decided Kh* # *H*
shows *D* ∉# *learned-clss z*
using *assms*(*1*−*2*) *M DL DH LH learned z*
proof (*induction rule*: *cdcl*$_W$*-o-induct*)
case (*backtrack L′ D′ K j M1 M2 T*) **note** *confl* = *this*(*1*) **and** *LD′* = *this*(*2*) **and** *decomp* = *this*(*3*)
and *levL* = *this*(*4*) **and** *levD* = *this*(*5*) **and** *j* = *this*(*6*) **and** *lev-K* = *this*(*7*) **and** *T* = *this*(*8*) **and**
z = *this*(*15*)
def *i* ≡ *get-level* (*trail T*) *Kh*
have *levT*: *cdcl*$_W$*-M-level-inv T*
using *backtrack-rule*[*OF confl LD′ decomp levL levD - - T*] *lev-K j lev*
by (*metis Suc-eq-plus1 cdcl*$_W$*.simps cdcl*$_W$*-bj.simps cdcl*$_W$*-consistent-inv cdcl*$_W$*-o.simps*)
obtain *M3* **where** *M3*: *trail y* = *M3* @ *M2* @ *Decided K* # *M1*
using *decomp get-all-ann-decomposition-exists-prepend* **by** *metis*
have *c′* @ *Decided Kh* # *H* = *Propagated L′ D′* # *trail* (*reduce-trail-to M1 y*)
using *z decomp T lev* **by** (*force simp*: *cdcl*$_W$*-M-level-inv-def*)
then obtain *d* **where** *d*: *M1* = *d* @ *Decided Kh* # *H*
by (*metis* (*no-types*) *decomp in-get-all-ann-decomposition-trail-update-trail list.inject*
list.sel(*3*) *ann-lit.distinct*(*1*) *self-append-conv2 tl-append2*)

have *atm-of Kh* ∉ *atm-of* ‘ *lits-of-l c′*
using *levT* **unfolding** *cdcl*$_W$*-M-level-inv-def z*
by (*auto simp*: *atm-lit-of-set-lits-of-l*)
then have *count-H*: *count-decided H* = *i* − *1 i* > *0*
unfolding *z i-def* **by** *auto*
have *n-d-y*: *no-dup* (*trail y*) **and** *bt-y*: *backtrack-lvl y* = *count-decided* (*trail y*)
using *lev* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
have *tr-T*: *trail T* = *Propagated L′ D′* # *M1*
using *decomp T n-d-y* **by** *auto*

show *?case*
proof
assume *D* ∈# *learned-clss T*
then have *DLD′*: *D* = *D′*
using *DL T neq0-conv decomp n-d-y* **by** *fastforce*
have *L-cKh*: *atm-of L* ∈ *atm-of* ‘ *lits-of-l* (*c* @ [*Decided Kh*])
using *LH learned M DLD′*[*symmetric*] *confl LD′ LD*

apply (*auto simp add*: *image-iff dest*!: *in-CNot-implies-uminus*)
apply (*metis atm-of-uminus*)+ **done**
then consider (*Lc*) *atm-of L* ∈ *atm-of* ‘ *lits-of-l c* **and** *atm-of L* ≠ *atm-of Kh* |
(*LKh*) *atm-of L* = *atm-of Kh* **and** *atm-of L* ∉ *atm-of* ‘ *lits-of-l c*
using *n-d-y M* **by** (*auto simp*: *atm-lit-of-set-lits-of-l*)
then have *lev-L-c-Kh*: *get-level* (*c* @ [*Decided Kh*]) *L* ≥ *1*
by *cases auto*
have *get-level* (*trail y*) *L* = *get-level* (*c* @ [*Decided Kh*]) *L* + *count-decided H*
using *get-rev-level-skip-end*[*OF L-cKh*, *of H*] **unfolding** *M* **by** *simp*
then have *get-level* (*trail y*) *L* ≥ *i*
using *count-H lev-L-c-Kh* **by** *linarith*
then have *i-le-bt-y*: *i* ≤ *backtrack-lvl y*
using *cdcl*$_W$*-M-level-inv-get-level-le-backtrack-lvl*[*OF lev*, *of L*] **by** *linarith*
have *DD′*[*simp*]: *remove1-mset L D* = *D′* − {#*L′*#}
proof (*rule ccontr*)
assume *DD′*: ¬ *?thesis*
then have *L′* ∈# *remove1-mset L D* **using** *DLD′ LD* **by** (*metis LD′ in-remove1-mset-neq*)

then have *get-level* (*trail y*) $L' \leq$ *get-maximum-level* (*trail y*) (*remove1-mset L D*)
using *get-maximum-level-ge-get-level* **by** *blast*
moreover
have $\forall x \in$ *atms-of* (*remove1-mset L D*). $x \notin$ *atm-of* ' *lits-of-l* (*c* @ *Decided Kh* # [])
using *DH n-d-y* **unfolding** *M* **by** (*auto simp*: *atm-lit-of-set-lits-of-l*)
from *get-maximum-level-skip-beginning*[*OF this*, *of H*]
have *get-maximum-level* (*trail y*) (*remove1-mset L D*) =
get-maximum-level H (*remove1-mset L D*)
unfolding *M* **by** (*simp add*: *get-maximum-level-skip-beginning*)
moreover
have *atm-of Kh* $\notin$ *atm-of* ' *lits-of-l c'*
using *levT* **unfolding** $cdcl_W$*-M-level-inv-def z*
by (*auto simp*: *atm-lit-of-set-lits-of-l*)
then have *count-decided H* $<$ *i*
unfolding *i-def z* **by** *auto*
then have $0 < i -$ *count-decided H*
by *presburger*
ultimately have *get-maximum-level* (*trail y*) (*remove1-mset L D*) $<$ *i*
by (*metis* (*no-types*) *count-decided-ge-get-maximum-level diff-is-0-eq diff-le-mono2 not-le*)
moreover
have $L \in\#$ *remove1-mset* L' D'
using DLD'[*symmetric*] DD' *LD* **by** (*metis in-remove1-mset-neq*)
then have *get-maximum-level* (*trail y*) (*remove1-mset* L' D') $\geq$
get-level (*trail y*) *L*
using *get-maximum-level-ge-get-level* **by** *blast*
moreover
have *get-maximum-level* (*trail y*) (*remove1-mset* L' D')
$<$ *get-level* (*trail y*) *L*
using ‹*get-level* (*trail y*) $L' \leq$ *get-maximum-level* (*trail y*) (*remove1-mset L D*)›
calculation(*1*) *i-le-bt-y levL* **by** *linarith*
ultimately show *False* **using** *backtrack.hyps*(*4*) **by** *linarith*
qed
then have LL': $L = L'$
using *LD* LD' *remove1-mset-eq-remove1-mset-same* **unfolding** DLD'[*symmetric*] **by** *fast*

have [*simp*]: *atm-of K* $\notin$ *atm-of* ' *lits-of-l M2* **and**
[*simp*]: *atm-of K* $\notin$ *atm-of* ' *lits-of-l M3*
using *lev* **unfolding** *M3* $cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *atm-lit-of-set-lits-of-l*)
{ assume *D*: *remove1-mset L* $D' = \{\#\}$
then have *j0*: $j = 0$ **using** *levD j* **by** (*simp add*: LL')
have $\forall m \in$ *set M1*. $\neg$*is-decided m*
using *lev-K* **unfolding** *j0 M3* **by** (*auto simp*: *atm-lit-of-set-lits-of-l image-Un filter-empty-conv*)
then have *False* **using** *d* **by** *auto*
}
moreover {
assume *D*[*simp*]: *remove1-mset L* $D' \neq \{\#\}$
have $i \leq j$
using *lev count-H lev-K* **unfolding** *M3 d* $cdcl_W$*-M-level-inv-def* **by** (*auto simp add*: *atm-lit-of-set-lits-of-l*)
have $j > 0$ **apply** (*rule ccontr*)
using ‹$i > 0$› *lev-K* **unfolding** *M3 d*
by (*auto simp add*: *rev-swap*[*symmetric*] *dest*!: *upt-decomp-lt*)
obtain L'' **where**
$L''\in\#$ *remove1-mset L* D' **and**

L''D': *get-level* (*trail y*) L'' = *get-maximum-level* (*trail y*)
(*remove1-mset L D'*)
using *get-maximum-level-exists-lit-of-max-level*[*OF D*, *of trail y*] **by** *auto*
have *L''M*: *atm-of* L'' ∈ *atm-of* ' *lits-of-l* (*trail y*)
using *get-level-ge-0-atm-of-in*[*of 0* L'' *trail y*] ‹*j*>*0*› *levD L''D'*
i-le-bt-y levL **by** (*simp add*: *LL' j*)
then have L'' ∈ *lits-of-l* (*Decided Kh* # *d*)
proof −
{
assume *L''H*: *atm-of* L'' ∈ *atm-of* ' *lits-of-l H*
then have *atm-of* L'' ∉ *atm-of* ' *lits-of-l* (*c* @ [*Decided Kh*])
using *n-d-y* **unfolding** *M* **by** (*auto simp*: *lits-of-def atm-of-eq-atm-of*)
then have *get-level* (*trail y*) L'' = *get-level H* L''
using *L''H* **unfolding** *M* **by** *auto*
moreover have *get-level H* L'' ≤ *count-decided H*
by *auto*
ultimately have *False*
using ‹*j*>*0*› ‹*i* ≤ *j*› *L''D' LL'* ‹*get-level H* L'' ≤ *count-decided H*› *count-H*(*1*) *j*
unfolding *count-H* **by** *presburger*
}
moreover
have *atm-of* L'' ∈ *atm-of* ' *lits-of-l H*
using *DD' DH* ‹L'' ∈# *remove1-mset L D'*› *atm-of-lit-in-atms-of LL' LD*
LD' **by** *fastforce*
ultimately show *?thesis*
using *DD' DH* ‹L'' ∈# *remove1-mset L D'*› *atm-of-lit-in-atms-of*
by *auto*
qed
moreover
have *atm-of* L'' ∈ *atms-of* (*remove1-mset L D'*)
using ‹L''∈# *remove1-mset L D'*› **by** (*auto simp*: *atms-of-def*)

then have *atm-of* L'' ∈ *atm-of* ' *lits-of-l H*
using *DH* **unfolding** *DD'* **unfolding** *LL'* **by** *blast*
ultimately have *False*
using *n-d-y* **unfolding** *M3 d LL'* **by** (*auto simp*: *lits-of-def*)
}
ultimately show *False* **by** *blast*
qed
qed *auto*

lemma *cdcl*$_W$*-stgy-with-trail-end-has-not-been-learned*:
assumes
cdcl$_W$*-stgy y z* **and**
cdcl$_W$*-M-level-inv y* **and**
trail y = *c* @ *Decided Kh* # *H* **and**
D ∉# *learned-clss y* **and**
LD: *L* ∈# *D* **and**
DH: *atms-of* (*remove1-mset L D*) ⊆ *atm-of* ' *lits-of-l H* **and**
LH: *atm-of L* ∉ *atm-of* ' *lits-of-l H* **and**
∀ *T*. *conflicting y* = *Some T* ⟶ *trail y* ⊨*as CNot T* **and**
trail z = *c'* @ *Decided Kh* # *H*
shows *D* ∉# *learned-clss z*
using *assms*
proof *induction*
case *conflict'*

then show *?case*
unfolding *full1-def* **using** *tranclp-cdcl$_W$-cp-learned-clause-inv* **by** *auto*
next
case (*other′ T U*) **note** $o = this(1)$ **and** $cp = this(3)$ **and** $lev = this(4)$ **and** $trY = this(5)$ **and** $notin = this(6)$ **and** $LD = this(7)$ **and** $DH = this(8)$ **and** $LH = this(9)$ **and** $confl = this(10)$ **and** $trU = this(11)$
obtain c' **where** c': *trail T = c′ @ Decided Kh # H*
using *cp beginning-not-decided-invert*[*of - trail T c′ Kh H*]
rtranclp-cdcl$_W$-cp-dropWhile-trail[*of T U*] **unfolding** *trU full-def* **by** *fastforce*
show *?case*
using *cdcl$_W$-o-cannot-learn*[*OF o lev trY notin LD DH LH confl c′*]
rtranclp-cdcl$_W$-cp-learned-clause-inv cp **unfolding** *full-def* **by** *auto*
qed

lemma *rtranclp-cdcl$_W$-stgy-with-trail-end-has-not-been-learned*:
assumes
($\lambda a\ b.$ *cdcl$_W$-stgy a b* $\wedge$ ($\exists c.$ *trail a = c @ Decided K# H @* []))** *S z* **and**
cdcl$_W$-all-struct-inv S **and**
trail S = c @ Decided K # H **and**
D $\notin$# *learned-clss S* **and**
LD: *L* $\in$# *D* **and**
DH: *atms-of* (*remove1-mset L D*) $\subseteq$ *atm-of ‘ lits-of-l H* **and**
LH: *atm-of L* $\notin$ *atm-of ‘ lits-of-l H* **and**
$\exists c'$. *trail z = c′ @ Decided K # H*
shows *D* $\notin$# *learned-clss z*
using *assms*(*1−4*,*8*)
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *auto*[*1*]
next
case (*step T U*) **note** $st = this(1)$ **and** $s = this(2)$ **and** $IH = this(3)$[*OF this*(*4−6*)]
and $lev = this(4)$ **and** $trS = this(5)$ **and** $DL\text{-}S = this(6)$ **and** $trU = this(7)$
obtain *c* **where** *c*: *trail T = c @ Decided K # H* **using** *s* **by** *auto*
obtain c' **where** c': *trail U = c′ @ Decided K # H* **using** *trU* **by** *blast*
have *cdcl$_W$** S T*
proof −
have $\forall p\ pa.\ \exists s\ sa.\ \forall sb\ sc\ sd\ se.$ ($\neg$ p^{**} (*sb*::′*st*) *sc* $\vee$ *p s sa* $\vee$ pa^{**} *sb sc*)
$\wedge$ ($\neg$ *pa s sa* $\vee$ $\neg$ p^{**} *sd se* $\vee$ pa^{**} *sd se*)
by (*metis* (*no-types*) *mono-rtranclp*)
then have *cdcl$_W$-stgy** S T*
using *st* **by** *blast*
then show *?thesis*
using *rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$* **by** *blast*
qed
then have *lev′*: *cdcl$_W$-all-struct-inv T*
using *rtranclp-cdcl$_W$-all-struct-inv-inv*[*of S T*] *lev* **by** *auto*
then have *confl′*: $\forall$ *Ta. conflicting T = Some Ta* $\longrightarrow$ *trail T* $\models$*as CNot Ta*
unfolding *cdcl$_W$-all-struct-inv-def cdcl$_W$-conflicting-def* **by** *blast*
show *?case*
apply (*rule cdcl$_W$-stgy-with-trail-end-has-not-been-learned*[*OF - - c - LD DH LH confl′ c′*])
using *s lev′ IH c* **unfolding** *cdcl$_W$-all-struct-inv-def* **by** *blast+*
qed

lemma *cdcl$_W$-stgy-new-learned-clause*:
assumes *cdcl$_W$-stgy S T* **and**
lev: *cdcl$_W$-M-level-inv S* **and**

$E \notin\#$ *learned-clss* S **and**
$E \in\#$ *learned-clss* T
shows $\exists S'.$ *backtrack* $S\ S' \wedge$ *conflicting* $S =$ *Some* $E \wedge$ *full* $cdcl_W$*-cp* $S'\ T$
using *assms*
proof *induction*
case *conflict'*
then show *?case* **unfolding** *full1-def* **by** (*auto dest*: *tranclp-*$cdcl_W$*-cp-learned-clause-inv*)
next
case (*other'* T U) **note** $o = this(1)$ **and** $cp = this(3)$ **and** $not\text{-}yet = this(5)$ **and** $learned = this(6)$
have $E \in\#$ *learned-clss* T
using *learned cp rtranclp-*$cdcl_W$*-cp-learned-clause-inv* **unfolding** *full-def* **by** *auto*
then have *backtrack* S T **and** *conflicting* $S =$ *Some* E
using $cdcl_W$*-o-new-clause-learned-is-backtrack-step*[*OF - not-yet o*] *lev* **by** *blast+*
then show *?case* **using** *cp* **by** *blast*
qed

theorem 2.9.7 page 83 of Weidenbach's book

lemma $cdcl_W$*-stgy-no-relearned-clause*:
assumes
invR: $cdcl_W$*-all-struct-inv* R **and**
st': $cdcl_W$*-stgy*** R S **and**
bt: *backtrack* S T **and**
confl: *conflicting* $S =$ *Some* E **and**
already-learned: $E \in\#$ *clauses* S **and**
R: *trail* $R = []$
shows *False*
proof $-$
have *M-lev*: $cdcl_W$*-M-level-inv* R
using *invR* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *auto*
have $cdcl_W$*-M-level-inv* S
using *M-lev assms*(*2*) *rtranclp-*$cdcl_W$*-stgy-consistent-inv* **by** *blast*
with *bt* **obtain** L K :: *'v literal* **and** *M1 M2-loc* :: (*'v*, *'v clause*) *ann-lits*
and i :: *nat* **where**
T: $T \sim$ *cons-trail* (*Propagated* L E)
(*reduce-trail-to M1* (*add-learned-cls* E
(*update-backtrack-lvl* i (*update-conflicting None* S))))
and
decomp: (*Decided* K # *M1*, *M2-loc*) $\in$
set (*get-all-ann-decomposition* (*trail* S)) **and**
LD: $L \in\#$ E **and**
k: *get-level* (*trail* S) $L =$ *backtrack-lvl* S **and**
level: *get-level* (*trail* S) $L =$ *get-maximum-level* (*trail* S) E **and**
confl-S: *conflicting* $S =$ *Some* E **and**
i: $i =$ *get-maximum-level* (*trail* S) (*remove1-mset* L E) **and**
lev-K: *get-level* (*trail* S) $K =$ *Suc* i
using *confl* **apply** (*induction rule*: *backtrack.induct*)
apply (*simp del*: *state-simp*)
by *blast*
obtain *M2* **where**
M: *trail* $S =$ *M2* @ *Decided* K # *M1*
using *get-all-ann-decomposition-exists-prepend*[*OF decomp*] **unfolding** i **by** (*metis append-assoc*)
let *?E'* = *remove1-mset* L E
have *invS*: $cdcl_W$*-all-struct-inv* S
using *invR rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ *st'* **by** *blast*
then have *conf*: $cdcl_W$*-conflicting* S **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *blast*
then have *trail* S $\models$*as CNot* E **unfolding** $cdcl_W$*-conflicting-def confl-S* **by** *auto*

then have *MD*: *trail S* $\models$*as CNot E* **by** *auto*
then have *MD'*: *trail S* $\models$*as CNot ?E'* **using** *true-annot-CNot-diff* **by** *blast*
have *lev'*: $cdcl_W$*-M-level-inv S* **using** *invS* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *blast*

have *lev*: $cdcl_W$*-M-level-inv R* **using** *invR* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *blast*
then have *vars-of-D*: *atms-of ?E'* $\subseteq$ *atm-of ‘ lits-of-l M1*
  using *backtrack-atms-of-D-in-M1*[*OF lev' - decomp - - -, of E - i T*] *confl-S conf T decomp k level lev' lev-K* **unfolding** *i* $cdcl_W$*-conflicting-def* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
have *no-dup* (*trail S*) **using** *lev'* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
have *vars-in-M1*:
  $\forall$ *x* $\in$ *atms-of ?E'*. *x* $\notin$ *atm-of ‘ lits-of-l* (*M2* @ [*Decided K*])
  unfolding *Set.Ball-def* **apply** (*intro impI allI*)
    apply (*rule vars-of-D distinct-atms-of-incl-not-in-other*[*of*
    M2 @ *Decided K* # [] *M1 ?E'*])
    using ‹*no-dup* (*trail S*)› *M vars-of-D* **by** *simp-all*
have *M1-D*: *M1* $\models$*as CNot ?E'*
  using *vars-in-M1 true-annots-remove-if-notin-vars*[*of M2* @ *Decided K* # [] *M1 CNot ?E'*]
  MD' M **by** *simp*

have *backtrack-lvl S* > *0* **using** *lev'* **unfolding** $cdcl_W$*-M-level-inv-def M* **by** *auto*

obtain *M1' K' Ls* **where**
  M': *trail S* = *Ls* @ *Decided K'* # *M1'* **and**
  Ls: $\forall$ *l* $\in$ *set Ls*. $\neg$ *is-decided l* **and**
  set M1 $\subseteq$ *set M1'*
  proof −
    let *?Ls* = *takeWhile* (*Not o is-decided*) (*trail S*)
    have *MLs*: *trail S* = *?Ls* @ *dropWhile* (*Not o is-decided*) (*trail S*)
      by *auto*
    have *dropWhile* (*Not o is-decided*) (*trail S*) $\neq$ [] **unfolding** *M* **by** *auto*
    moreover
      from *hd-dropWhile*[*OF this*] **have** *is-decided*(*hd* (*dropWhile* (*Not o is-decided*) (*trail S*)))
        by *simp*
    ultimately
      obtain *K'* **where**
        K'k: *dropWhile* (*Not o is-decided*) (*trail S*)
          = *Decided K'* # *tl* (*dropWhile* (*Not o is-decided*) (*trail S*))
        by (*cases dropWhile* (*Not* $\circ$ *is-decided*) (*trail S*);
            cases hd (*dropWhile* (*Not* $\circ$ *is-decided*) (*trail S*)))
          simp-all
    moreover have $\forall$ *l* $\in$ *set ?Ls*. $\neg$*is-decided l* **using** *set-takeWhileD* **by** *force*
    moreover have *set M1* $\subseteq$ *set* (*tl* (*dropWhile* (*Not o is-decided*) (*trail S*)))
      unfolding *M* **by** (*induction M2*) *auto*
    ultimately show *?thesis* **using** *that*[*of takeWhile* (*Not* $\circ$ *is-decided*) (*trail S*)
      K' tl (*dropWhile* (*Not o is-decided*) (*trail S*))] *MLs* **by** *simp*
  qed

have *M1'-D*: *M1'* $\models$*as CNot ?E'* **using** *M1-D* ‹*set M1* $\subseteq$ *set M1'*› **by** (*auto intro*: *true-annots-mono*)
have −*L* $\in$ *lits-of-l* (*trail S*) **using** *conf confl-S LD* **unfolding** $cdcl_W$*-conflicting-def*
  by (*auto simp*: *in-CNot-implies-uminus*)
have *L-notin*: *atm-of L* $\in$ *atm-of ‘ lits-of-l Ls* $\lor$ *atm-of L* = *atm-of K'*
  proof (*rule ccontr*)
    assume $\neg$ *?thesis*
    then have *atm-of L* $\notin$ *atm-of ‘ lits-of-l* (*Decided K'* # *rev Ls*) **by** *simp*
    then have *get-level* (*trail S*) *L* = *get-level M1' L*
      unfolding *M'* **by** *auto*

moreover
have *get-level M1' L* $\leq$ *count-decided M1'*
by *auto*
then have *get-level M1' L* $<$ *backtrack-lvl S*
using *lev'* **unfolding** $cdcl_W$*-M-level-inv-def M'*
by (*auto simp del*: *count-decided-ge-get-level*)
ultimately show *False* **using** *k* **by** *linarith*
qed
obtain *Y Z* **where**
RY: $cdcl_W$*-stgy*** *R Y* **and**
YZ: $cdcl_W$*-stgy Y Z* **and**
nt: ¬ (∃ *c. trail Y* = *c* @ *Decided K'* # *M1'* @ []) **and**
Z: (λ*a b.* $cdcl_W$*-stgy a b* ∧ (∃ *c. trail a* = *c* @ *Decided K'* # *M1'* @ []))** *Z S*
using *rtranclp-*$cdcl_W$*-new-decided-at-beginning-is-decide'*[*OF st' - - lev*, *of Ls K'*
M1' []] **unfolding** *R M'* **by** *auto*
have [*simp*]: $cdcl_W$*-M-level-inv Y*
using *RY lev rtranclp-*$cdcl_W$*-stgy-consistent-inv* **by** *blast*
obtain *M'* **where** *trZ*: *trail Z* = *M'* @ *Decided K'* # *M1'*
using *rtranclp-*$cdcl_W$*-stgy-with-trail-end-has-trail-end*[*OF Z*] *M'* **by** *auto*
have *no-dup* (*trail Y*)
using *RY lev rtranclp-*$cdcl_W$*-stgy-consistent-inv* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *blast*
then obtain *Y'* **where**
dec: *decide Y Y'* **and**
Y'Z: *full* $cdcl_W$*-cp Y' Z* **and**
no-step $cdcl_W$*-cp Y*
using $cdcl_W$*-stgy-trail-has-new-decided-is-decide-step*[*OF YZ nt Z*] *M'* **by** *auto*
have *trY*: *trail Y* = *M1'*
proof −
obtain *M'* **where** *M*: *trail Z* = *M'* @ *Decided K'* # *M1'*
using *rtranclp-*$cdcl_W$*-stgy-with-trail-end-has-trail-end*[*OF Z*] *M'* **by** *auto*
obtain *M''* **where** *M''*: *trail Z* = *M''* @ *trail Y'* **and** ∀ *m*∈*set M''*. ¬*is-decided m*
using *Y'Z rtranclp-*$cdcl_W$*-cp-dropWhile-trail'* **unfolding** *full-def* **by** *blast*
obtain *M'''* **where** *trail Y'* = *M'''* @ *Decided K'* # *M1'*
using *M''* **unfolding** *M*
by (*metis* (*no-types*, *lifting*) ‹∀ *m*∈*set M''*. ¬ *is-decided m*› *beginning-not-decided-invert*)
then show *?thesis* **using** *dec nt* **by** (*induction M'''*) (*auto elim*: *decideE*)
qed
have *Y-CT*: *conflicting Y* = *None* **using** ‹*decide Y Y'*› **by** (*auto elim*: *decideE*)
have $cdcl_W$** *R Y* **by** (*simp add*: *RY rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$)
then have *init-clss Y* = *init-clss R* **using** *rtranclp-*$cdcl_W$*-init-clss*[*of R Y*] *M-lev* **by** *auto*
{ assume *DL*: *E* ∈# *clauses Y*
have *atm-of L* ∉ *atm-of* ' *lits-of-l M1*
apply (*rule backtrack-lit-skiped*[*of - S*])
using *decomp i k lev' lev-K* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *LM1*: *undefined-lit M1 L*
by (*metis Decided-Propagated-in-iff-in-lits-of-l atm-of-uminus image-eqI*)
have *L-trY*: *undefined-lit* (*trail Y*) *L*
using *L-notin* ‹*no-dup* (*trail S*)› **unfolding** *defined-lit-map trY M'*
by (*auto simp add*: *image-iff lits-of-def*)
have *Ex* (*propagate Y*)
using *propagate-rule*[*of Y E L*] *DL M1'-D L-trY Y-CT trY LD* **by** *auto*
then have *False* **using** ‹*no-step* $cdcl_W$*-cp Y*› *propagate'* **by** *blast*
}
moreover {
assume *DL*: *E* ∉# *clauses Y*
have *lY-lZ*: *learned-clss Y* = *learned-clss Z*

using *dec Y'Z rtranclp-cdcl*$_W$*-cp-learned-clause-inv*[*of Y' Z*] **unfolding** *full-def*
by (*auto elim*: *decideE*)
have *invZ*: *cdcl*$_W$*-all-struct-inv Z*
by (*meson RY YZ invR r-into-rtranclp rtranclp-cdcl*$_W$*-all-struct-inv-inv*
rtranclp-cdcl$_W$*-stgy-rtranclp-cdcl*$_W$)
have *n*: *E* $\notin\#$ *learned-clss Z*
using *DL lY-lZ YZ* **unfolding** *clauses-def* **by** *auto*
have *E* $\notin\#$*learned-clss S*
apply (*rule rtranclp-cdcl*$_W$*-stgy-with-trail-end-has-not-been-learned*[*OF Z invZ trZ*])
apply (*simp add*: *n*)
using *LD* **apply** *simp*
apply (*metis* (*no-types*, *lifting*) ‹*set M1* ⊆ *set M1'*› *image-mono order-trans*
vars-of-D lits-of-def)
using *L-notin* ‹*no-dup* (*trail S*)› **unfolding** *M'* **by** (*auto simp add*: *image-iff lits-of-def*)
then have *False*
using *already-learned DL confl st' M-lev rtranclp-cdcl*$_W$*-stgy-no-more-init-clss*[*of R S*]
unfolding *M'*
by (*simp add*: ‹*init-clss Y* = *init-clss R*› *clauses-def confl-S*
rtranclp-cdcl$_W$*-stgy-no-more-init-clss*)
}
ultimately show *False* **by** *blast*
qed

lemma *rtranclp-cdcl*$_W$*-stgy-distinct-mset-clauses*:
assumes
invR: *cdcl*$_W$*-all-struct-inv R* **and**
st: *cdcl*$_W$*-stgy*** *R S* **and**
dist: *distinct-mset* (*clauses R*) **and**
R: *trail R* = []
shows *distinct-mset* (*clauses S*)
using *st*
proof (*induction*)
case *base*
then show *?case* **using** *dist* **by** *simp*
next
case (*step S T*) **note** *st* = *this*(*1*) **and** *s* = *this*(*2*) **and** *IH* = *this*(*3*)
from *s* **show** *?case*
proof (*cases rule*: *cdcl*$_W$*-stgy.cases*)
case *conflict'*
then show *?thesis*
using *IH* **unfolding** *full1-def* **by** (*auto dest*: *tranclp-cdcl*$_W$*-cp-no-more-clauses*)
next
case (*other' S'*) **note** *o* = *this*(*1*) **and** *full* = *this*(*3*)
have [*simp*]: *clauses T* = *clauses S'*
using *full* **unfolding** *full-def* **by** (*auto dest*: *rtranclp-cdcl*$_W$*-cp-no-more-clauses*)
show *?thesis*
using *o IH*
proof (*cases rule*: *cdcl*$_W$*-o-rule-cases*)
case *backtrack*
moreover
have *cdcl*$_W$*-all-struct-inv S*
using *invR rtranclp-cdcl*$_W$*-stgy-cdcl*$_W$*-all-struct-inv st* **by** *blast*
then have *cdcl*$_W$*-M-level-inv S*
unfolding *cdcl*$_W$*-all-struct-inv-def* **by** *auto*
ultimately obtain *E* **where**
conflicting S = *Some E* **and**

cls-S': *clauses S'* = {#*E*#} + *clauses S*
using ‹*cdcl*$_W$*-M-level-inv S*›
by (*induction rule*: *backtrack.induct*) (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
then have *E* ∉# *clauses S*
using *cdcl*$_W$*-stgy-no-relearned-clause R invR local.backtrack st* **by** *blast*
then show *?thesis* **using** *IH* **by** (*simp add*: *distinct-mset-add-single cls-S'*)
qed (*auto elim*: *decideE skipE resolveE*)
qed
qed

lemma *cdcl*$_W$*-stgy-distinct-mset-clauses*:
assumes
st: *cdcl*$_W$*-stgy*** (*init-state N*) *S* **and**
no-duplicate-clause: *distinct-mset N* **and**
no-duplicate-in-clause: *distinct-mset-mset N*
shows *distinct-mset* (*clauses S*)
using *rtranclp-cdcl*$_W$*-stgy-distinct-mset-clauses*[*OF - st*] *assms*
by (*auto simp*: *cdcl*$_W$*-all-struct-inv-def distinct-cdcl*$_W$*-state-def*)

Decrease of a Measure

fun *cdcl*$_W$*-measure* **where**
cdcl$_W$*-measure S* =
[(*3*::*nat*) ^ (*card* (*atms-of-mm* (*init-clss S*))) − *card* (*set-mset* (*learned-clss S*)),
if conflicting S = None then 1 else 0,
if conflicting S = None then card (*atms-of-mm* (*init-clss S*)) − *length* (*trail S*)
else length (*trail S*)
]

lemma *length-model-le-vars-all-inv*:
assumes *cdcl*$_W$*-all-struct-inv S*
shows *length* (*trail S*) ≤ *card* (*atms-of-mm* (*init-clss S*))
using *assms length-model-le-vars*[*of S*] **unfolding** *cdcl*$_W$*-all-struct-inv-def*
by (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
end

context *conflict-driven-clause-learning*$_W$
begin

lemma *learned-clss-less-upper-bound*:
fixes *S* :: *'st*
assumes
distinct-cdcl$_W$*-state S* **and**
∀ *s* ∈# *learned-clss S*. ¬*tautology s*
shows *card*(*set-mset* (*learned-clss S*)) ≤ *3* ^ *card* (*atms-of-mm* (*learned-clss S*))
proof −
have *set-mset* (*learned-clss S*) ⊆ *simple-clss* (*atms-of-mm* (*learned-clss S*))
apply (*rule simplified-in-simple-clss*)
using *assms* **unfolding** *distinct-cdcl*$_W$*-state-def* **by** *auto*
then have *card*(*set-mset* (*learned-clss S*))
≤ *card* (*simple-clss* (*atms-of-mm* (*learned-clss S*)))
by (*simp add*: *simple-clss-finite card-mono*)
then show *?thesis*
by (*meson atms-of-ms-finite simple-clss-card finite-set-mset order-trans*)
qed

lemma *cdcl*$_W$*-measure-decreasing*:
  fixes S :: *'st*
  assumes
    cdcl$_W$ S S' **and**
    no-restart:
      ¬(*learned-clss* S ⊆# *learned-clss* S' ∧ [] = *trail* S' ∧ *conflicting* S' = *None*)
     and
    no-forget: *learned-clss* S ⊆# *learned-clss* S' **and**
    no-relearn: ⋀S'. *backtrack* S S' ⟹ ∀ T. *conflicting* S = *Some* T ⟶ T ∉# *learned-clss* S
      and
    alien: *no-strange-atm* S **and**
    M-level: *cdcl*$_W$*-M-level-inv* S **and**
    no-taut: ∀ s ∈# *learned-clss* S. ¬*tautology* s **and**
    no-dup: *distinct-cdcl*$_W$*-state* S **and**
    confl: *cdcl*$_W$*-conflicting* S
  shows (*cdcl*$_W$*-measure* S', *cdcl*$_W$*-measure* S) ∈ *lexn less-than 3*
  using *assms*(*1*) *M-level assms*(*2*,*3*)
proof (*induct rule*: *cdcl*$_W$*-all-induct*)
  case (*propagate* C L) **note** *conf* = *this*(*1*) **and** *undef* = *this*(*5*) **and** T = *this*(*6*)
  have *propa*: *propagate* S (*cons-trail* (*Propagated* L C) S)
    using *propagate-rule*[*OF propagate.hyps*(*1*,*2*)] *propagate.hyps* **by** *auto*
  then have *no-dup'*: *no-dup* (*Propagated* L C # *trail* S)
    using *M-level cdcl*$_W$*-M-level-inv-decomp*(*2*) *undef defined-lit-map* **by** *auto*

  let *?N* = *init-clss* S
  have *no-strange-atm* (*cons-trail* (*Propagated* L C) S)
    using *alien cdcl*$_W$*.propagate cdcl*$_W$*-no-strange-atm-inv propa M-level* **by** *blast*
  then have *atm-of* ' *lits-of-l* (*Propagated* L C # *trail* S)
    ⊆ *atms-of-mm* (*init-clss* S)
    using *undef* **unfolding** *no-strange-atm-def* **by** *auto*
  then have *card* (*atm-of* ' *lits-of-l* (*Propagated* L C # *trail* S))
    ≤ *card* (*atms-of-mm* (*init-clss* S))
    by (*meson atms-of-ms-finite card-mono finite-set-mset*)
  then have *length* (*Propagated* L C # *trail* S) ≤ *card* (*atms-of-mm ?N*)
    using *no-dup-length-eq-card-atm-of-lits-of-l no-dup'* **by** *fastforce*
  then have *H*: *card* (*atms-of-mm* (*init-clss* S)) − *length* (*trail* S)
    = *Suc* (*card* (*atms-of-mm* (*init-clss* S)) − *Suc* (*length* (*trail* S)))
    by *simp*
  show *?case* **using** *conf* T *undef* **by** (*auto simp*: *H lexn3-conv*)
next
  case (*decide* L) **note** *conf* = *this*(*1*) **and** *undef* = *this*(*2*) **and** T = *this*(*4*)
  moreover
    have *dec*: *decide* S (*cons-trail* (*Decided* L) (*incr-lvl* S))
      using *decide-rule decide.hyps* **by** *force*
    then have *cdcl*$_W$:*cdcl*$_W$ S (*cons-trail* (*Decided* L) (*incr-lvl* S))
      using *cdcl*$_W$*.simps cdcl*$_W$*-o.intros* **by** *blast*
  moreover
    have *lev*: *cdcl*$_W$*-M-level-inv* (*cons-trail* (*Decided* L) (*incr-lvl* S))
      using *cdcl*$_W$ *M-level cdcl*$_W$*-consistent-inv*[*OF cdcl*$_W$] **by** *auto*
    then have *no-dup*: *no-dup* (*Decided* L # *trail* S)
      using *undef* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
    have *no-strange-atm* (*cons-trail* (*Decided* L) (*incr-lvl* S))
      using *M-level alien calculation*(*4*) *cdcl*$_W$*-no-strange-atm-inv* **by** *blast*
    then have *length* (*Decided* L # (*trail* S))
      ≤ *card* (*atms-of-mm* (*init-clss* S))

    using *no-dup undef*
    length-model-le-vars[*of cons-trail* (*Decided L*) (*incr-lvl S*)]
    by *fastforce*
  ultimately show *?case* **using** *conf* **by** (*simp add*: *lexn3-conv*)
next
  case (*skip L C' M D*) **note** *tr* = *this*(*1*) **and** *conf* = *this*(*2*) **and** *T* = *this*(*5*)
  show *?case* **using** *conf T* **by** (*simp add*: *tr lexn3-conv*)
next
  case *conflict*
  then show *?case* **by** (*simp add*: *lexn3-conv*)
next
  case *resolve*
  then show *?case* **using** *finite* **by** (*simp add*: *lexn3-conv*)
next
  case (*backtrack L D K i M1 M2 T*) **note** *conf* = *this*(*1*) **and** *decomp* = *this*(*3*) **and** *T* = *this*(*8*) **and** *lev* = *this*(*9*)
  have *bt*: *backtrack S T*
    using *backtrack-rule*[*OF backtrack.hyps*] **by** *auto*
  have $D \notin\#$ *learned-clss S*
    using *no-relearn conf bt* **by** *auto*
  then have *card-T*:
    card (*set-mset* ({#*D*#} + *learned-clss S*)) = *Suc* (*card* (*set-mset* (*learned-clss S*)))
    by *simp*
  have *distinct-cdcl$_W$-state T*
    using *bt M-level distinct-cdcl$_W$-state-inv no-dup other cdcl$_W$-o.intros cdcl$_W$-bj.intros* **by** *blast*
  moreover have $\forall s \in\#$*learned-clss T*. $\neg$ *tautology s*
    using *learned-clss-are-not-tautologies*[*OF cdcl$_W$.other*[*OF cdcl$_W$-o.bj*[*OF*
      cdcl$_W$-bj.backtrack[*OF bt*]]]] *M-level no-taut confl* **by** *auto*
  ultimately have *card* (*set-mset* (*learned-clss T*)) $\leq$ *3* ^ *card* (*atms-of-mm* (*learned-clss T*))
      by (*auto simp*: *learned-clss-less-upper-bound*)
    then have *H*: *card* (*set-mset* ({#*D*#} + *learned-clss S*))
      $\leq$ *3* ^ *card* (*atms-of-mm* ({#*D*#} + *learned-clss S*))
      using *T decomp M-level* **by** (*simp add*: *cdcl$_W$-M-level-inv-decomp*)
  moreover
    have *atms-of-mm* ({#*D*#} + *learned-clss S*) $\subseteq$ *atms-of-mm* (*init-clss S*)
      using *alien conf* **unfolding** *no-strange-atm-def* **by** *auto*
    then have *card-f*: *card* (*atms-of-mm* ({#*D*#} + *learned-clss S*))
      $\leq$ *card* (*atms-of-mm* (*init-clss S*))
      by (*meson atms-of-ms-finite card-mono finite-set-mset*)
    then have (*3*::*nat*) ^ *card* (*atms-of-mm* ({#*D*#} + *learned-clss S*))
      $\leq$ *3* ^ *card* (*atms-of-mm* (*init-clss S*)) **by** *simp*
  ultimately have (*3*::*nat*) ^ *card* (*atms-of-mm* (*init-clss S*))
    $\geq$ *card* (*set-mset* ({#*D*#} + *learned-clss S*))
    using *le-trans* **by** *blast*
  then show *?case* **using** *decomp diff-less-mono2 card-T T M-level*
    by (*auto simp*: *cdcl$_W$-M-level-inv-decomp lexn3-conv*)
next
  case *restart*
  then show *?case* **using** *alien* **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)
next
  case (*forget C T*) **note** *no-forget* = *this*(*9*)
  then have $C \in\#$ *learned-clss S* **and** $C \notin\#$ *learned-clss T*
    using *forget.hyps* **by** *auto*
  then have $\neg$ *learned-clss S* $\subseteq\#$ *learned-clss T*
     by (*auto simp add*: *mset-leD*)
  then show *?case* **using** *no-forget* **by** *blast*

qed

lemma *propagate-measure-decreasing*:
  fixes S :: *'st*
  assumes *propagate* S S' **and** $cdcl_W$*-all-struct-inv* S
  shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
  apply (*rule* $cdcl_W$*-measure-decreasing*)
  using *assms*(*1*) *propagate* **apply** *blast*
      using *assms*(*1*) **apply** (*auto simp add*: *propagate.simps*)[*3*]
    using *assms*(*2*) **apply** (*auto simp add*: $cdcl_W$*-all-struct-inv-def*)
  done

lemma *conflict-measure-decreasing*:
  fixes S :: *'st*
  assumes *conflict* S S' **and** $cdcl_W$*-all-struct-inv* S
  shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
  apply (*rule* $cdcl_W$*-measure-decreasing*)
  using *assms*(*1*) *conflict* **apply** *blast*
      using *assms*(*1*) **apply** (*auto simp*: *state-eq-def simp del*: *state-simp elim*!: *conflictE*)[*3*]
    using *assms*(*2*) **apply** (*auto simp add*: $cdcl_W$*-all-struct-inv-def elim*: *conflictE*)
  done

lemma *decide-measure-decreasing*:
  fixes S :: *'st*
  assumes *decide* S S' **and** $cdcl_W$*-all-struct-inv* S
  shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
  apply (*rule* $cdcl_W$*-measure-decreasing*)
  using *assms*(*1*) *decide other* **apply** *blast*
      using *assms*(*1*) **apply** (*auto simp*: *state-eq-def simp del*: *state-simp elim*!: *decideE*)[*3*]
    using *assms*(*2*) **apply** (*auto simp add*: $cdcl_W$*-all-struct-inv-def elim*: *decideE*)
  done

lemma $cdcl_W$*-cp-measure-decreasing*:
  fixes S :: *'st*
  assumes $cdcl_W$*-cp* S S' **and** $cdcl_W$*-all-struct-inv* S
  shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
  using *assms*
proof *induction*
  case *conflict'*
  then show *?case* **using** *conflict-measure-decreasing* **by** *blast*
next
  case *propagate'*
  then show *?case* **using** *propagate-measure-decreasing* **by** *blast*
qed

lemma *tranclp-*$cdcl_W$*-cp-measure-decreasing*:
  fixes S :: *'st*
  assumes $cdcl_W$*-cp*$^{++}$ S S' **and** $cdcl_W$*-all-struct-inv* S
  shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
  using *assms*
proof *induction*
  case *base*
  then show *?case* **using** $cdcl_W$*-cp-measure-decreasing* **by** *blast*
next
  case (*step T U*) **note** *st* = *this*(*1*) **and** *step* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *inv* = *this*(*4*)
  then have ($cdcl_W$*-measure* T, $cdcl_W$*-measure* S) $\in$ *lexn less-than 3* **by** *blast*

moreover have ($cdcl_W$*-measure U*, $cdcl_W$*-measure T*) ∈ *lexn less-than 3*
using $cdcl_W$*-cp-measure-decreasing*[*OF step*] *rtranclp-*$cdcl_W$*-all-struct-inv-inv inv*
tranclp-$cdcl_W$*-cp-tranclp-*$cdcl_W$[*OF st*]
unfolding *trans-def rtranclp-unfold*
by *blast*
ultimately show *?case* **using** *lexn-transI*[*OF trans-less-than*] **unfolding** *trans-def* **by** *blast*
qed

lemma $cdcl_W$*-stgy-step-decreasing*:
fixes *R S T* :: *'st*
assumes $cdcl_W$*-stgy S T* **and**
$cdcl_W$*-stgy*** *R S*
trail R = [] **and**
$cdcl_W$*-all-struct-inv R*
shows ($cdcl_W$*-measure T*, $cdcl_W$*-measure S*) ∈ *lexn less-than 3*
proof −
have $cdcl_W$*-all-struct-inv S*
using *assms*
by (*metis rtranclp-unfold rtranclp-*$cdcl_W$*-all-struct-inv-inv tranclp-*$cdcl_W$*-stgy-tranclp-*$cdcl_W$)
with *assms* **show** *?thesis*
proof *induction*
case (*conflict' V*) **note** *cp* = *this*(*1*) **and** *inv* = *this*(*5*)
show *?case*
using *tranclp-*$cdcl_W$*-cp-measure-decreasing*[*OF HOL.conjunct1*[*OF cp*[*unfolded full1-def*]] *inv*]
.
next
case (*other' T U*) **note** *st* = *this*(*1*) **and** *H* = *this*(*4*,*5*,*6*,*7*) **and** *cp* = *this*(*3*)
have $cdcl_W$*-all-struct-inv T*
using $cdcl_W$*-all-struct-inv-inv other other'.hyps*(*1*) *other'.prems*(*4*) **by** *blast*
from *tranclp-*$cdcl_W$*-cp-measure-decreasing*[*OF - this*]
have *le-or-eq*: ($cdcl_W$*-measure U*, $cdcl_W$*-measure T*) ∈ *lexn less-than 3* ∨
$cdcl_W$*-measure U* = $cdcl_W$*-measure T*
using *cp* **unfolding** *full-def rtranclp-unfold* **by** *blast*
moreover
have $cdcl_W$*-M-level-inv S*
using $cdcl_W$*-all-struct-inv-def other'.prems*(*4*) **by** *blast*
with *st* **have** ($cdcl_W$*-measure T*, $cdcl_W$*-measure S*) ∈ *lexn less-than 3*
proof (*induction rule*:$cdcl_W$*-o-induct*)
case (*decide T*)
then show *?case* **using** *decide-measure-decreasing H decide.intros*[*OF decide.hyps*] **by** *blast*
next
case (*backtrack L D K i M1 M2 T*) **note** *conf* = *this*(*1*) **and** *decomp* = *this*(*3*) **and**
undef = *this*(*8*) **and** *T* = *this*(*9*)
have *bt*: *backtrack S T*
apply (*rule backtrack-rule*)
using *backtrack.hyps* **by** *auto*
then have *no-relearn*: ∀ *T*. *conflicting S* = *Some T* ⟶ *T* ∉# *learned-clss S*
using $cdcl_W$*-stgy-no-relearned-clause*[*of R S T*] *H conf*
unfolding $cdcl_W$*-all-struct-inv-def clauses-def* **by** *auto*
have *inv*: $cdcl_W$*-all-struct-inv S*
using ‹$cdcl_W$*-all-struct-inv S*› **by** *blast*
show *?case*
apply (*rule* $cdcl_W$*-measure-decreasing*)
using *bt* $cdcl_W$*-bj.backtrack* $cdcl_W$*-o.bj other* **apply** *simp*
using *bt T undef decomp inv* **unfolding** $cdcl_W$*-all-struct-inv-def*

$cdcl_W$*-M-level-inv-def* **apply** *auto*[]
using *bt T undef decomp inv* **unfolding** $cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-M-level-inv-def* **apply** *auto*[]
using *bt no-relearn* **apply** *auto*[]
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **apply** *simp*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **apply** *simp*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **apply** *simp*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **apply** *simp*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *simp*
next
case *skip*
then show *?case* **by** (*auto simp*: *lexn3-conv*)
next
case *resolve*
then show *?case* **by** (*auto simp*: *lexn3-conv*)
qed
ultimately show *?case*
by (*metis* (*full-types*) *lexn-transI transD trans-less-than*)
qed
qed

Roughly corresponds to theorem 2.9.15 page 86 of Weidenbach's book (using a different bound)

lemma *tranclp-*$cdcl_W$*-stgy-decreasing*:
fixes *R S T* :: *'st*
assumes $cdcl_W$*-stgy*$^{++}$ *R S*
trail R = [] **and**
$cdcl_W$*-all-struct-inv R*
shows ($cdcl_W$*-measure S*, $cdcl_W$*-measure R*) ∈ *lexn less-than 3*
using *assms*
apply *induction*
using $cdcl_W$*-stgy-step-decreasing*[*of R - R*] **apply** *blast*
using $cdcl_W$*-stgy-step-decreasing*[*of - - R*] *tranclp-into-rtranclp*[*of* $cdcl_W$*-stgy R*]
lexn-transI[*OF trans-less-than*, *of 3*] **unfolding** *trans-def* **by** *blast*

lemma *tranclp-*$cdcl_W$*-stgy-S0-decreasing*:
fixes *R S T* :: *'st*
assumes
pl: $cdcl_W$*-stgy*$^{++}$ (*init-state N*) *S* **and**
no-dup: *distinct-mset-mset N*
shows ($cdcl_W$*-measure S*, $cdcl_W$*-measure* (*init-state N*)) ∈ *lexn less-than 3*
proof −
have $cdcl_W$*-all-struct-inv* (*init-state N*)
using *no-dup* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *auto*
then show *?thesis* **using** *pl tranclp-*$cdcl_W$*-stgy-decreasing init-state-trail* **by** *blast*
qed

lemma *wf-tranclp-*$cdcl_W$*-stgy*:
wf {(*S*::*'st*, *init-state N*)|
S N. *distinct-mset-mset N* ∧ $cdcl_W$*-stgy*$^{++}$ (*init-state N*) *S*}
apply (*rule wf-wf-if-measure'-notation2*[*of lexn less-than 3 - -* $cdcl_W$*-measure*])
apply (*simp add*: *wf wf-lexn*)
using *tranclp-*$cdcl_W$*-stgy-S0-decreasing* **by** *blast*

lemma $cdcl_W$*-cp-wf-all-inv*:
wf {(*S'*, *S*). $cdcl_W$*-all-struct-inv S* ∧ $cdcl_W$*-cp S S'*}
(**is** *wf ?R*)

proof (*rule wf-bounded-measure*[*of* -
  λS. *card* (*atms-of-mm* (*init-clss S*))+*1*
  λS. *length* (*trail S*) + (*if conflicting S* = *None then 0 else 1*)], *goal-cases*)
 case (*1 S S'*)
 then have $cdcl_W$*-all-struct-inv S* **and** $cdcl_W$*-cp S S'* **by** *auto*
 moreover then have $cdcl_W$*-all-struct-inv S'*
  using $cdcl_W$*-cp.simps* $cdcl_W$*-all-struct-inv-inv conflict* $cdcl_W$*.intros* $cdcl_W$*-all-struct-inv-inv*
  by *blast+*
 ultimately show *?case*
  by (*auto simp*:$cdcl_W$*-cp.simps state-eq-def simp del*: *state-simp elim*!: *conflictE propagateE*
   dest: *length-model-le-vars-all-inv*)
qed

end

end

6.2 Merging backjump rules

theory *CDCL-W-Merge*
imports *CDCL-W-Termination*
begin

Before showing that Weidenbach's CDCL is included in NOT's CDCL, we need to work on a variant of Weidenbach's calculus: NOT's backjump assumes the existence of a clause that is suitable to backjump. This clause is obtained in W's CDCL by applying:

1. *conflict-driven-clause-learning*$_W$*.conflict* to find the conflict
2. the conflict is analysed by repetitive application of *conflict-driven-clause-learning*$_W$*.resolve* and *conflict-driven-clause-learning*$_W$*.skip*,
3. finally *conflict-driven-clause-learning*$_W$*.backtrack* is used to backtrack.

We show that this new calculus has the same final states than Weidenbach's CDCL if the calculus starts in a state such that the invariant holds and no conflict has been found yet. The latter condition holds for initial states.

6.2.1 Inclusion of the states

context *conflict-driven-clause-learning*$_W$
begin
declare $cdcl_W$*.intros*[*intro*] $cdcl_W$*-bj.intros*[*intro*] $cdcl_W$*-o.intros*[*intro*]

lemma *backtrack-no-*$cdcl_W$*-bj*:
 assumes *cdcl*: $cdcl_W$*-bj T U* **and** *inv*: $cdcl_W$*-M-level-inv V*
 shows ¬*backtrack V T*
 using *cdcl inv*
 apply (*induction rule*: $cdcl_W$*-bj.induct*)
  apply (*elim skipE*, *force elim*!: *backtrackE simp*: $cdcl_W$*-M-level-inv-def*)
  apply (*elim resolveE*, *force elim*!: *backtrackE simp*: $cdcl_W$*-M-level-inv-def*)
 apply *standard*
 apply (*elim backtrackE*)
 apply (*force simp del*: *state-simp simp add*: *state-eq-def* $cdcl_W$*-M-level-inv-decomp*)

done

skip-or-resolve corresponds to the *analyze* function in the code of MiniSAT.

inductive *skip-or-resolve* :: *'st* ⇒ *'st* ⇒ *bool* **where**
s-or-r-skip[*intro*]: *skip S T* ⟹ *skip-or-resolve S T* |
s-or-r-resolve[*intro*]: *resolve S T* ⟹ *skip-or-resolve S T*

lemma *rtranclp-cdcl$_W$-bj-skip-or-resolve-backtrack*:
assumes *cdcl$_W$-bj** S U* **and** *inv*: *cdcl$_W$-M-level-inv S*
shows *skip-or-resolve** S U* ∨ (∃ *T*. *skip-or-resolve** S T* ∧ *backtrack T U*)
using *assms*
proof (*induction*)
case *base*
then show *?case* **by** *simp*
next
case (*step U V*) **note** *st* = *this*(*1*) **and** *bj* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*)]
consider
(*SU*) *S* = *U*
| (*SUp*) *cdcl$_W$-bj^{++} S U*
using *st* **unfolding** *rtranclp-unfold* **by** *blast*
then show *?case*
proof *cases*
case *SUp*
have ⋀*T*. *skip-or-resolve** S T* ⟹ *cdcl$_W$** S T*
using *mono-rtranclp*[*of skip-or-resolve cdcl$_W$*]
by (*blast intro*: *skip-or-resolve.cases*)
then have *skip-or-resolve** S U*
using *bj IH inv backtrack-no-cdcl$_W$-bj rtranclp-cdcl$_W$-consistent-inv*[*OF - inv*] **by** *meson*
then show *?thesis*
using *bj* **by** (*auto simp*: *cdcl$_W$-bj.simps dest*!: *skip-or-resolve.intros*)
next
case *SU*
then show *?thesis*
using *bj* **by** (*auto simp*: *cdcl$_W$-bj.simps dest*!: *skip-or-resolve.intros*)
qed
qed

lemma *rtranclp-skip-or-resolve-rtranclp-cdcl$_W$*:
skip-or-resolve** S T ⟹ *cdcl$_W$** S T*
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *cdcl$_W$-bj.intros cdcl$_W$.intros cdcl$_W$-o.intros simp*: *skip-or-resolve.simps*)

definition *backjump-l-cond* :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **where**
backjump-l-cond ≡ λ*C C' L' S T*. *True*

definition *inv$_{NOT}$* :: *'st* ⇒ *bool* **where**
inv$_{NOT}$ ≡ λ*S*. *no-dup* (*trail S*)

declare *inv$_{NOT}$-def*[*simp*]
end

context *conflict-driven-clause-learning$_W$*
begin

6.2.2 More lemmas conflict–propagate and backjumping

Termination

lemma *cdcl*$_W$*-cp-normalized-element-all-inv*:
  assumes *inv*: *cdcl*$_W$*-all-struct-inv S*
  obtains *T* **where** *full cdcl*$_W$*-cp S T*
  using *assms cdcl*$_W$*-cp-normalized-element* **unfolding** *cdcl*$_W$*-all-struct-inv-def* **by** *blast*
thm *backtrackE*

lemma *cdcl*$_W$*-bj-measure*:
  assumes *cdcl*$_W$*-bj S T* **and** *cdcl*$_W$*-M-level-inv S*
  shows *length* (*trail S*) + (*if conflicting S* = *None then 0 else 1*)
    > *length* (*trail T*) + (*if conflicting T* = *None then 0 else 1*)
  using *assms* **by** (*induction rule*: *cdcl*$_W$*-bj.induct*)
  (*force dest*: *arg-cong*[*of - - length*]
    intro: *get-all-ann-decomposition-exists-prepend*
    elim!: *backtrackE skipE resolveE*
    simp: *cdcl*$_W$*-M-level-inv-def*)+

lemma *wf-cdcl*$_W$*-bj*:
  wf {(*b*,*a*). *cdcl*$_W$*-bj a b* ∧ *cdcl*$_W$*-M-level-inv a*}
  apply (*rule wfP-if-measure*[*of* λ-. *True*
      - λ*T*. *length* (*trail T*) + (*if conflicting T* = *None then 0 else 1*), *simplified*])
  using *cdcl*$_W$*-bj-measure* **by** *simp*

lemma *cdcl*$_W$*-bj-exists-normal-form*:
  assumes *lev*: *cdcl*$_W$*-M-level-inv S*
  shows ∃ *T*. *full cdcl*$_W$*-bj S T*
proof −
  obtain *T* **where** *T*: *full* (λ*a b*. *cdcl*$_W$*-bj a b* ∧ *cdcl*$_W$*-M-level-inv a*) *S T*
    using *wf-exists-normal-form-full*[*OF wf-cdcl*$_W$*-bj*] **by** *auto*
  then have *cdcl*$_W$*-bj*** *S T*
     by (*auto dest*: *rtranclp-and-rtranclp-left simp*: *full-def*)
  moreover
    then have *cdcl*$_W$** *S T*
      using *mono-rtranclp*[*of cdcl*$_W$*-bj cdcl*$_W$] **by** *blast*
    then have *cdcl*$_W$*-M-level-inv T*
      using *rtranclp-cdcl*$_W$*-consistent-inv lev* **by** *auto*
  ultimately show *?thesis* **using** *T* **unfolding** *full-def* **by** *auto*
qed
lemma *rtranclp-skip-state-decomp*:
  assumes *skip*** *S T* **and** *no-dup* (*trail S*)
  shows
    ∃ *M*. *trail S* = *M* @ *trail T* ∧ (∀ *m*∈*set M*. ¬*is-decided m*)
    init-clss S = *init-clss T*
    learned-clss S = *learned-clss T*
    backtrack-lvl S = *backtrack-lvl T*
    conflicting S = *conflicting T*
  using *assms* **by** (*induction rule*: *rtranclp-induct*)
  (*auto simp del*: *state-simp simp*: *state-eq-def elim*!: *skipE*)

More backjumping

Backjumping after skipping or jump directly **lemma** *rtranclp-skip-backtrack-backtrack*:
  assumes
    skip** *S T* **and**

backtrack T W **and**
cdcl$_W$-all-struct-inv S
shows *backtrack S W*
using *assms*
proof *induction*
case *base*
then show *?case* **by** *simp*
next
case (*step T V*) **note** *st* = *this*(*1*) **and** *skip* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *bt* = *this*(*4*) **and**
inv = *this*(*5*)
have *skip** S V*
using *st skip* **by** *auto*
then have *cdcl$_W$-all-struct-inv V*
using *rtranclp-mono*[*of skip cdcl$_W$*] *assms*(*3*) *rtranclp-cdcl$_W$-all-struct-inv-inv mono-rtranclp*
by (*auto dest*!: *bj other cdcl$_W$-bj.skip*)
then have *cdcl$_W$-M-level-inv V*
unfolding *cdcl$_W$-all-struct-inv-def* **by** *auto*
then obtain *K i M1 M2 L D* **where**
conf: *conflicting V* = *Some D* **and**
LD: *L* $\in\#$ *D* **and**
decomp: (*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail V*)) **and**
lev-L: *get-level* (*trail V*) *L* = *backtrack-lvl V* **and**
max: *get-level* (*trail V*) *L* = *get-maximum-level* (*trail V*) *D* **and**
max-D: *get-maximum-level* (*trail V*) (*remove1-mset L D*) $\equiv$ *i* **and**
lev-k: *get-level* (*trail V*) *K* = *Suc i* **and**
W: *W* $\sim$ *cons-trail* (*Propagated L D*)
(*reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None V*))))
using *bt inv* **by** (*elim backtrackE*) *metis+*
obtain *L' C' M E* **where**
tr: *trail T* = *Propagated L' C'* # *M* **and**
raw: *conflicting T* = *Some E* **and**
LE: $-L' \notin\#$ *E* **and**
E: *E* $\neq$ {#} **and**
V: *V* $\sim$ *tl-trail T*
using *skip* **by** (*elim skipE*) *metis*
let *?M* = *Propagated L' C'* # *trail V*
have *tr-M*: *trail T* = *?M*
using *tr V* **by** *auto*
have *MT*: *M* = *tl* (*trail T*) **and** *MV*: *M* = *trail V*
using *tr V* **by** *auto*
have *DE*[*simp*]: *D* = *E*
using *V conf raw* **by** (*auto simp add*: *state-eq-def simp del*: *state-simp*)
have *cdcl$_W$** S T* **using** *bj cdcl$_W$-bj.skip mono-rtranclp*[*of skip cdcl$_W$ S T*] *other st* **by** *meson*
then have *inv'*: *cdcl$_W$-all-struct-inv T*
using *rtranclp-cdcl$_W$-all-struct-inv-inv inv* **by** *blast*
have *M-lev*: *cdcl$_W$-M-level-inv T* **using** *inv'* **unfolding** *cdcl$_W$-all-struct-inv-def* **by** *auto*
then have *n-d'*: *no-dup ?M*
using *tr-M* **unfolding** *cdcl$_W$-M-level-inv-def* **by** *auto*
let *?k* = *backtrack-lvl T*
have [*simp*]:
backtrack-lvl V = *?k*
using *V* **by** *simp*
have *?k* > *0*

using *decomp M-lev V tr* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have *atm-of L* ∈ *atm-of ‘ lits-of-l* (*trail V*)
using *lev-L get-level-ge-0-atm-of-in*[*of 0 L trail V*] **by** *auto*
then have *L-L′*: *atm-of L* ≠ *atm-of L′*
using *n-d′* **unfolding** *lits-of-def* **by** *auto*
have *L′-M*: *atm-of L′* ∉ *atm-of ‘ lits-of-l* (*trail V*)
using *n-d′* **unfolding** *lits-of-def* **by** *auto*
have *?M* ⊨*as CNot D*
using *inv′ raw* **unfolding** *cdcl*$_W$*-conflicting-def cdcl*$_W$*-all-struct-inv-def tr-M* **by** *auto*
then have *L′* ∉# (*remove1-mset L D*)
using *L-L′ L′-M* ‹*Propagated L′ C′* # *trail V* ⊨*as CNot D*›
unfolding *true-annots-true-cls true-clss-def*
by (*auto simp*: *uminus-lit-swap atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set dest*!: *in-diffD*)
have [*simp*]: *trail* (*reduce-trail-to M1 T*) = *M1*
using *decomp tr W V* **by** *auto*
have *skip*** *S V*
using *st skip* **by** *auto*
have *no-dup* (*trail S*)
using *inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have [*simp*]: *init-clss S* = *init-clss V* **and** [*simp*]: *learned-clss S* = *learned-clss V*
using *rtranclp-skip-state-decomp*[*OF* ‹*skip*** *S V*›] *V*
by (*auto simp del*: *state-simp simp*: *state-eq-def*)
then have
W-S: *W* ∼ *cons-trail* (*Propagated L E*) (*reduce-trail-to M1*
(*add-learned-cls E* (*update-backtrack-lvl i* (*update-conflicting None T*))))
using *W V M-lev decomp tr*
by (*auto simp del*: *state-simp simp*: *state-eq-def cdcl*$_W$*-M-level-inv-def*)

obtain *M2′* **where**
decomp′: (*Decided K* # *M1*, *M2′*) ∈ *set* (*get-all-ann-decomposition* (*trail T*))
using *decomp V* **unfolding** *tr-M* **by** (*cases hd* (*get-all-ann-decomposition* (*trail V*)),
cases get-all-ann-decomposition (*trail V*)) *auto*
moreover
from *L-L′* **have** *get-level ?M L* = *?k*
using *lev-L V* **by** (*auto split*: *if-split-asm*)
moreover
have *atm-of L′* ∉ *atms-of D*
by (*metis DE LE L-L′* ‹*L′* ∉# (*remove1-mset L D*)› *in-remove1-mset-neq*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def)
then have *get-level ?M L* = *get-maximum-level ?M D*
using *calculation*(*2*) *lev-L max* **by** *auto*
moreover
have *atm-of L′* ∉ *atms-of* ((*remove1-mset L D*))
by (*metis DE LE* ‹*L′* ∉# (*remove1-mset L D*)›
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def in-remove1-mset-neq
in-atms-of-remove1-mset-in-atms-of)
have *i* = *get-maximum-level ?M* ((*remove1-mset L D*))
using *max-D* ‹*atm-of L′* ∉ *atms-of* ((*remove1-mset L D*))› **by** *auto*
moreover have *atm-of L′* ≠ *atm-of K*
using *inv′ get-all-ann-decomposition-exists-prepend*[*OF decomp*]
unfolding *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def tr MV* **by** *auto*
ultimately have *backtrack T W*
apply −
apply (*rule backtrack-rule*[*of T - L K M1 M2′ i W*, *OF raw*])
unfolding *tr-M*[*symmetric*]
using *LD* **apply** *simp*

apply *simp*
apply *simp*
apply *simp*
apply *auto*[]
using *W-S lev-k tr MV* **apply** *auto*[]
using *W-S lev-k* **apply** *auto*[]
done
then show *?thesis* **using** *IH inv* **by** *blast*
qed

See also theorem *rtranclp-skip-backtrack-backtrack*

lemma *rtranclp-skip-backtrack-backtrack-end*:
assumes
skip: $skip^{**}$ *S T* **and**
bt: *backtrack S W* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows *backtrack T W*
using *assms*
proof −
have *M-lev*: $cdcl_W$*-M-level-inv S*
using *bt inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** (*auto elim*!: *backtrackE*)
then obtain *K i M1 M2 L D* **where**
S: *conflicting S* = *Some D* **and**
LD: *L* ∈# *D* **and**
decomp: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
lev-l: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
lev-l-D: *get-level* (*trail S*) *L* = *get-maximum-level* (*trail S*) *D* **and**
i: *get-maximum-level* (*trail S*) (*remove1-mset L D*) ≡ *i* **and**
lev-K: *get-level* (*trail S*) *K* = *Suc i* **and**
W: *W* ∼ *cons-trail* (*Propagated L D*)
(*reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None S*))))
using *bt* **by** (*elim backtrackE*)
(*simp-all add*: $cdcl_W$*-M-level-inv-decomp state-eq-def del*: *state-simp*)
let *?D* = *remove1-mset L D*

have [*simp*]: *no-dup* (*trail S*)
using *M-lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
have $cdcl_W$*-all-struct-inv T*
using *mono-rtranclp*[*of skip* $cdcl_W$] **by** (*smt bj* $cdcl_W$*-bj.skip inv local.skip other*
rtranclp-$cdcl_W$*-all-struct-inv-inv*)
then have [*simp*]: *no-dup* (*trail T*)
unfolding $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*

obtain *MS* M_T **where** *M*: *trail S* = *MS* @ M_T **and** M_T: M_T = *trail T* **and** *nm*: ∀ *m*∈*set MS*.
¬*is-decided m*
using *rtranclp-skip-state-decomp*(*1*)[*OF skip*] *S M-lev* **by** *auto*
have *T*: *state T* = (M_T, *init-clss S*, *learned-clss S*, *backtrack-lvl S*, *Some D*)
using M_T *rtranclp-skip-state-decomp*[*of S T*] *skip S*
by (*auto simp del*: *state-simp simp*: *state-eq-def*)

have $cdcl_W$*-all-struct-inv T*
apply (*rule rtranclp-*$cdcl_W$*-all-struct-inv-inv*[*OF - inv*])
using *bj* $cdcl_W$*-bj.skip local.skip other rtranclp-mono*[*of skip* $cdcl_W$] **by** *blast*

```
  then have M_T ⊨as CNot D
    unfolding cdcl_W-all-struct-inv-def cdcl_W-conflicting-def using T by blast
  then have ∀ L∈#D. atm-of L ∈ atm-of ‘ lits-of-l M_T
    by (meson atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      true-annots-true-cls-def-iff-negation-in-model)
  moreover have no-dup (trail S)
    using inv unfolding cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def by auto
  ultimately have ∀ L∈#D. atm-of L ∉ atm-of ‘ lits-of-l MS
    unfolding M unfolding lits-of-def by auto
  then have H: ⋀L. L∈#D ⟹ get-level (trail S) L = get-level M_T L
    unfolding M by (fastforce simp: lits-of-def)
  have [simp]: get-maximum-level (trail S) D = get-maximum-level M_T D
    using ‹M_T ⊨as CNot D› M nm ‹∀ L∈#D. atm-of L ∉ atm-of ‘ lits-of-l MS›
    by (auto simp: get-maximum-level-skip-un-decided-not-present)

  have lev-l': get-level M_T L = backtrack-lvl S
    using lev-l LD by (auto simp: H)
  have [simp]: trail (reduce-trail-to M1 T) = M1
    using T decomp M nm by (smt M_T append-assoc beginning-not-decided-invert
      get-all-ann-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp)
  have W: W ∼ cons-trail (Propagated L D) (reduce-trail-to M1
    (add-learned-cls D (update-backtrack-lvl i (update-conflicting None T))))
    using W T i decomp by (auto simp del: state-simp simp: state-eq-def)
  have lev-l-D': get-level M_T L = get-maximum-level M_T D
    using lev-l-D LD by (auto simp: H)
  have [simp]: get-maximum-level (trail S) ?D = get-maximum-level M_T ?D
    by (smt H get-maximum-level-exists-lit get-maximum-level-ge-get-level in-diffD le-antisym
      not-gr0 not-less)
  then have i': i = get-maximum-level M_T ?D
    using i by auto
  have Decided K # M1 ∈ set (map fst (get-all-ann-decomposition (trail S)))
    using Set.imageI[OF decomp, of fst] by auto
  then have Decided K # M1 ∈ set (map fst (get-all-ann-decomposition M_T))
    using fst-get-all-ann-decomposition-prepend-not-decided[OF nm] unfolding M by auto
  then obtain M2' where decomp':(Decided K # M1, M2') ∈ set (get-all-ann-decomposition M_T)
    by auto
  moreover
    have atm-of K ∉ atm-of ‘ lits-of-l MS
      using ‹no-dup (trail S)› decomp' unfolding M M_T
      by (auto simp: lits-of-def)
    then have get-level (trail T) K = get-level (trail S) K
      unfolding M M_T by auto
  ultimately show backtrack T W
    apply –
    apply (rule backtrack.intros[of T D])
      using T lev-l' lev-l-D' i' W LD lev-K i apply auto[7]
    using T W unfolding i'[symmetric]
    by (auto simp del: state-simp simp: state-eq-def)
qed

lemma cdcl_W-bj-decomp-resolve-skip-and-bj:
  assumes cdcl_W-bj** S T and inv: cdcl_W-M-level-inv S
  shows (skip-or-resolve** S T
    ∨ (∃ U. skip-or-resolve** S U ∧ backtrack U T))
  using assms
proof induction
```

```
  case base
  then show ?case by simp
next
  case (step T U) note st = this(1) and bj = this(2) and IH = this(3)
  have IH: skip-or-resolve** S T
    proof -
      { assume ∃ U. skip-or-resolve** S U ∧ backtrack U T
        then obtain V where
          bt: backtrack V T and
          skip-or-resolve** S V
          by blast
        have cdcl_W** S V
          using ‹skip-or-resolve** S V› rtranclp-skip-or-resolve-rtranclp-cdcl_W by blast
        then have cdcl_W-M-level-inv V and cdcl_W-M-level-inv S
          using rtranclp-cdcl_W-consistent-inv inv by blast+
        with bj bt have False using backtrack-no-cdcl_W-bj by simp
      }
      then show ?thesis using IH inv by blast
    qed
  show ?case
    using bj
    proof (cases rule: cdcl_W-bj.cases)
      case backtrack
      then show ?thesis using IH by blast
    qed (metis (no-types, lifting) IH rtranclp.simps skip-or-resolve.simps)+
qed

lemma resolve-skip-deterministic:
  resolve S T ⟹ skip S U ⟹ False
  by (auto elim!: skipE resolveE)

lemma list-same-level-decomp-is-same-decomp:
  assumes M-K: M = M1 @ Decided K # M2 and M-K': M = M1' @ Decided K' # M2' and
  lev-KK': get-level M K = get-level M K' and
  n-d: no-dup M
  shows K = K' and M1 = M1' and M2 = M2'
proof -
  {
    fix j j' K K' M1 M1' M2 M2'
    assume
      M-K: M = M1 @ Decided K # M2 and
      M-K': M = M1' @ Decided K' # M2' and
      levKK': get-level M K = get-level M K' and
      j: M ! j = Decided K and j-M: j < length M and
      j': M ! j' = Decided K' and j'-M: j' < length M and
      jj: j' > j
    have j ≥ length M1
      proof (rule ccontr)
        assume ¬ length M1 ≤ j
        then have j < length M1
          by auto
        then have Decided K ∈ set M1
          using j unfolding M-K
          by (auto simp: nth-append in-set-conv-nth split: if-splits)
        from Set.imageI[OF this, of λL. atm-of (lit-of L)]
        show False using n-d unfolding M-K by auto
```

```
      qed
    moreover then have j' − Suc (length M1) < length M2
      using j'-M jj M-K unfolding M-K' by (metis One-nat-def Suc-eq-plus1 add.left-commute
        le-less-trans length-append less-diff-conv2 list.size(4) not-less not-less-eq)
    ultimately have dec: Decided K' ∈ set M2
      using jj j j' j'-M unfolding M-K by (auto simp: nth-append in-set-conv-nth List.nth-Cons')
    obtain xs ys where
      M2: M2 = xs @ Decided K' # ys
      using List.split-list[OF dec] by auto
    have [simp]: atm-of K ≠ atm-of K'
      using n-d unfolding M-K M2 by auto
    have atm-of K ∉ atm-of ‘ lits-of-l M1 and atm-of K' ∉ atm-of ‘ lits-of-l M1 and
    atm-of K' ∉ atm-of ‘ lits-of-l xs
      using n-d Set.imageI[OF dec, of λL. atm-of (lit-of L)] unfolding M-K
      using n-d unfolding M-K M2
      by (auto simp: lits-of-def)
    then have False
      using M2 levKK' unfolding M-K by (auto simp: split: if-splits )
  } note H = this
  have Decided K ∈ set M and Decided K' ∈ set M
     using M-K apply simp
    using M-K' by simp
  then obtain j j' where
    j: M ! j = Decided K and j-M: j < length M and
    j': M ! j' = Decided K' and j'-M: j' < length M
      using in-set-conv-nth by metis

  have [simp]: j = j' using H[OF M-K M-K' - j j-M j' j'-M]
    H[OF M-K' M-K - j' j'-M j j-M] lev-KK' by presburger
  then show KK': K = K' using j j' by auto

  have j-M1: j = length M1
    proof (rule ccontr)
      assume j ≠ length M1
      moreover then have j − Suc (length M1) < length M2 ∨ j < length M1
        using j-M M-K unfolding M-K' by force
      ultimately have Decided K ∈ set (M1 @ M2)
        using j unfolding M-K by (auto simp: nth-append in-set-conv-nth split: if-splits)
      from Set.imageI[OF this, of λL. atm-of (lit-of L)]
      show False using n-d unfolding M-K by auto
    qed
  have j-M2: j' = length M1'
    proof (rule ccontr)
      assume j' ≠ length M1'
      moreover then have j' − Suc (length M1') < length M2' ∨ j' < length M1'
        using j'-M M-K' unfolding M-K by force
      ultimately have Decided K' ∈ set (M1' @ M2')
        using j' unfolding M-K' by (auto simp: nth-append in-set-conv-nth split: if-splits)
      from Set.imageI[OF this, of λL. atm-of (lit-of L)]
      show False using n-d unfolding M-K' by auto
    qed

  show M1 = M1' M2 = M2'
    using arg-cong[OF M-K, of take j] j-M1 arg-cong[OF M-K', of take j'] j-M2
    using arg-cong[OF M-K, of drop (j+1)] j-M1 arg-cong[OF M-K', of drop (j'+1)] j-M2
    by auto
```

qed

lemma *backtrack-unique*:
  assumes
    bt-T: *backtrack S T* **and**
    bt-U: *backtrack S U* **and**
    inv: $cdcl_W$*-all-struct-inv S*
  shows $T \sim U$
proof −
  have *lev*: $cdcl_W$*-M-level-inv S*
    using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *auto*
  then obtain *K i M1 M2 L D* **where**
    S: *conflicting S = Some D* **and**
    LD: $L \in\# D$ **and**
    decomp: (*Decided K # M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
    lev-l: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
    lev-l-D: *get-level* (*trail S*) *L* = *get-maximum-level* (*trail S*) *D* **and**
    i: *get-maximum-level* (*trail S*) (*remove1-mset L D*) ≡ *i* **and**
    lev-K: *get-level* (*trail S*) *K* = *Suc i* **and**
    T: *T* ∼ *cons-trail* (*Propagated L D*)
              (*reduce-trail-to M1*
                (*add-learned-cls D*
                  (*update-backtrack-lvl i*
                    (*update-conflicting None S*))))
    using *bt-T* **by** (*elim backtrackE*) (*force simp*: $cdcl_W$*-M-level-inv-def*)+

  obtain *K′ i′ M1′ M2′ L′ D′* **where**
    S′: *conflicting S = Some D′* **and**
    LD′: $L' \in\# D'$ **and**
    decomp′: (*Decided K′ # M1′*, *M2′*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
    lev-l: *get-level* (*trail S*) *L′* = *backtrack-lvl S* **and**
    lev-l-D: *get-level* (*trail S*) *L′* = *get-maximum-level* (*trail S*) *D′* **and**
    i′: *get-maximum-level* (*trail S*) (*remove1-mset L′ D′*) ≡ *i′* **and**
    lev-K′: *get-level* (*trail S*) *K′* = *Suc i′* **and**
    U: *U* ∼ *cons-trail* (*Propagated L′ D′*)
              (*reduce-trail-to M1′*
                (*add-learned-cls D′*
                  (*update-backtrack-lvl i′*
                    (*update-conflicting None S*))))
    using *bt-U lev* **by** (*elim backtrackE*) (*force simp*: $cdcl_W$*-M-level-inv-def*)+
  obtain *c* **where** *M*: *trail S = c @ M2 @ Decided K # M1*
    using *decomp* **by** *auto*
  obtain *c′* **where** *M′*: *trail S = c′ @ M2′ @ Decided K′ # M1′*
    using *decomp′* **by** *auto*
  have *n-d*: *no-dup* (*trail S*) **and** *bt*: *backtrack-lvl S = count-decided* (*trail S*)
    using *lev* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
  then have *atm-of K* ∉ *atm-of ‘ lits-of-l* (*c @ M2*)
    by (*auto simp*: *lits-of-def M*)
  then have *i* < *backtrack-lvl S*
    using *lev-K* **unfolding** *M bt* **by** (*auto simp add*: *image-Un*)

  have [*simp*]: *L′ = L*
    proof (*rule ccontr*)
      assume ¬ *?thesis*
      then have $L' \in\#$ *remove1-mset L D*
        using *S S′ LD LD′* **by** (*simp add*: *in-remove1-mset-neq*)

then have *get-maximum-level* (*trail S*) (*remove1-mset L D*) $\geq$ *backtrack-lvl S*
using ‹*get-level* (*trail S*) L' = *backtrack-lvl S*› *get-maximum-level-ge-get-level*
by *metis*
then show *False* **using** i' i ‹i < *backtrack-lvl S*› **by** *auto*
qed
then have [*simp*]: $D' = D$
using S S' **by** *auto*
have [*simp*]: $i' = i$
using i i' **by** *auto*
have [*simp*]: $K = K'$ **and** [*simp*]: $M1 = M1'$
apply (*rule list-same-level-decomp-is-same-decomp*[*of trail S c* @ *M2 K M1*
c' @ $M2'$ K' $M1'$])
using *lev-K lev-K'* M M' *n-d* **apply** (*auto*)[*4*]
apply (*rule list-same-level-decomp-is-same-decomp*[*of trail S c* @ *M2 K M1*
c' @ $M2'$ K' $M1'$])
using *lev-K lev-K'* M M' *n-d* **apply** (*auto*)[*4*]
done
show *?thesis* **using** *T U inv decomp* **by** (*auto simp del*: *state-simp simp*: *state-eq-def*
$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-decomp*)
qed

lemma *if-can-apply-backtrack-no-more-resolve*:
assumes
skip: $skip^{**}$ *S U* **and**
bt: *backtrack S T* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows ¬*resolve U V*
proof (*rule ccontr*)
assume *resolve*: ¬¬*resolve U V*

obtain *L E D* **where**
U: *trail U* $\neq$ [] **and**
tr-U: *hd-trail U* = *Propagated L E* **and**
LE: $L \in\#$ *E* **and**
confl-U: *conflicting U* = *Some D* **and**
LD: $-L \in\#$ *D* **and**
get-maximum-level (*trail U*) ((*remove1-mset* ($-L$) *D*)) = *backtrack-lvl U* **and**
V: *V* $\sim$ *update-conflicting* (*Some* (*resolve-cls L D E*)) (*tl-trail U*)
using *resolve* **by** (*auto elim*!: *resolveE*)
have *inv-U*: $cdcl_W$*-all-struct-inv U*
using *mono-rtranclp*[*of skip* $cdcl_W$] **by** (*meson bj* $cdcl_W$*-bj.skip inv local.skip other*
rtranclp-$cdcl_W$*-all-struct-inv-inv*)
then have [*iff*]: *no-dup* (*trail S*) $cdcl_W$*-M-level-inv S* **and** [*iff*]: *no-dup* (*trail U*)
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *blast+*
have *inv-V*: $cdcl_W$*-all-struct-inv V*
using *mono-rtranclp*[*of resolve* $cdcl_W$] *inv-U resolve* $cdcl_W$*.simps* $cdcl_W$*-all-struct-inv-inv*
$cdcl_W$*-bj.resolve* $cdcl_W$*-o.simps* **by** *blast*
have
S: *init-clss U* = *init-clss S*
learned-clss U = *learned-clss S*
backtrack-lvl U = *backtrack-lvl S*
backtrack-lvl V = *backtrack-lvl S*
conflicting S = *Some D*
using *rtranclp-skip-state-decomp*[*OF skip*] *U confl-U V*
by (*auto simp del*: *state-simp simp*: *state-eq-def*)
obtain M_0 **where**

tr-S: *trail S* = M_0 @ *trail U* **and**
nm: ∀ *m*∈*set* M_0. ¬*is-decided m*
using *rtranclp-skip-state-decomp*[*OF skip*] **by** *blast*

obtain *K′ i′ M1′ M2′ L′ D′* **where**
S′: *conflicting S* = *Some D′* **and**
LD′: *L′* ∈# *D′* **and**
decomp′: (*Decided K′* # *M1′*, *M2′*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
lev-l: *get-level* (*trail S*) *L′* = *backtrack-lvl S* **and**
lev-l-D: *get-level* (*trail S*) *L′* = *get-maximum-level* (*trail S*) *D′* **and**
i′: *get-maximum-level* (*trail S*) (*remove1-mset L′ D′*) ≡ *i′* **and**
lev-K′: *get-level* (*trail S*) *K′* = *Suc i′* **and**
R: *T* ∼ *cons-trail* (*Propagated L′ D′*)
(*reduce-trail-to M1′*
(*add-learned-cls D′*
(*update-backtrack-lvl i′*
(*update-conflicting None S*))))
using *bt* **by** (*elim backtrackE*) *metis*
obtain *c* **where** *M*: *trail S* = *c* @ *M2′* @ *Decided K′* # *M1′*
using *get-all-ann-decomposition-exists-prepend*[*OF decomp′*] **by** *auto*
have *i′* < *backtrack-lvl S*
using *count-decided-ge-get-level*[*of K′ trail S*] *inv*
unfolding $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def lev-K′*
by *linarith*

have *U*: *trail U* = *Propagated L E* # *trail V*
using *tr-S U S V tr-U* ⟨*trail U* ≠ []⟩ **by** (*cases trail U*) (*auto simp*: *lits-of-def*)
have *DD′*[*simp*]: *D′* = *D*
using *U S′ S* **by** *auto*
have [*simp*]: *L′* = −*L*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have −*L* ∈# *remove1-mset L′ D′*
using *DD′ LD′ LD* **by** (*simp add*: *in-remove1-mset-neq*)
moreover
have *M′*: *trail S* = M_0 @ *Propagated L E* # *trail V*
using *tr-S* **unfolding** *U* **by** *auto*
have *no-dup* (*trail S*)
using *inv U* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *atm-L-notin-M*: *atm-of L* ∉ *atm-of* ‘ (*lits-of-l* (*trail V*))
using *M′ U S* **by** (*auto simp*: *lits-of-def*)
have *get-lev-L*:
get-level(*Propagated L E* # *trail V*) *L* = *backtrack-lvl V*
using *inv-V* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
have *atm-of L* ∉ *atm-of* ‘ (*lits-of-l* (*rev* M_0))
using ⟨*no-dup* (*trail S*)⟩ *M′* **by** (*auto simp*: *lits-of-def*)
then have *get-level* (*trail S*) *L* = *backtrack-lvl S*
using *get-lev-L S* **unfolding** *M′* **by** *auto*
ultimately
have *get-maximum-level* (*trail S*) (*remove1-mset L′ D′*) ≥ *backtrack-lvl S*
by (*metis get-maximum-level-ge-get-level get-level-uminus*)
then show *False*
using ⟨*i′* < *backtrack-lvl S*⟩ *i′* **by** *auto*
qed
have $cdcl_W^{**}$ *S U*
using *bj* $cdcl_W$*-bj.skip local.skip mono-rtranclp*[*of skip* $cdcl_W$ *S U*] *other* **by** *meson*

then have $cdcl_W$*-all-struct-inv U*
using *inv rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
then have *Propagated L E # trail V* $\models$*as CNot D'*
using *U confl-U* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def* **by** *auto*
then have $\forall$ $L'\in\#$ (*remove1-mset L' D'*) .
atm-of L' $\in$ *atm-of ' lits-of-l* (*Propagated L E # trail U*)
using *U atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-CNot-implies-uminus*(*2*)
by (*fastforce dest*: *in-diffD*)
then have $\forall$ $L'\in\#$ (*remove1-mset L' D'*) .
atm-of L' $\notin$ *atm-of ' lits-of-l* M_0
using ‹*no-dup* (*trail S*)› **unfolding** *tr-S U* **by** (*fastforce simp*: *lits-of-def image-image*)
then have *get-maximum-level* (*trail S*) (*remove1-mset L' D'*) = *backtrack-lvl S*
using *get-maximum-level-skip-un-decided-not-present*[*of remove1-mset L' D'*
M_0 *trail U*] *tr-S nm U*
‹*get-maximum-level* (*trail U*) ((*remove1-mset* (− *L*) *D*)) = *backtrack-lvl U*›
by (*auto simp*: *S*)
then show *False*
using *i'* ‹*i'* < *backtrack-lvl S*› **by** *auto*
qed

lemma *if-can-apply-resolve-no-more-backtrack*:
assumes
skip: $skip^{**}$ *S U* **and**
resolve: *resolve S T* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows ¬*backtrack U V*
using *assms*
by (*meson if-can-apply-backtrack-no-more-resolve rtranclp.rtrancl-refl*
rtranclp-skip-backtrack-backtrack)

lemma *if-can-apply-backtrack-skip-or-resolve-is-skip*:
assumes
bt: *backtrack S T* **and**
skip: *skip-or-resolve*** *S U* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows $skip^{**}$ *S U*
using *assms*(*2*,*3*,*1*)
by *induction* (*simp-all add*: *if-can-apply-backtrack-no-more-resolve skip-or-resolve.simps*)

lemma $cdcl_W$*-bj-bj-decomp*:
assumes $cdcl_W$*-bj*** *S W* **and** $cdcl_W$*-all-struct-inv S*
shows
($\exists$ *T U V*. (λ*S T*. *skip-or-resolve S T* $\wedge$ *no-step backtrack S*)** *S T*
$\wedge$ (λ*T U*. *resolve T U* $\wedge$ *no-step backtrack T*) *T U*
$\wedge$ $skip^{**}$ *U V* $\wedge$ *backtrack V W*)
$\vee$ ($\exists$ *T U*. (λ*S T*. *skip-or-resolve S T* $\wedge$ *no-step backtrack S*)** *S T*
$\wedge$ (λ*T U*. *resolve T U* $\wedge$ *no-step backtrack T*) *T U* $\wedge$ $skip^{**}$ *U W*)
$\vee$ ($\exists$ *T*. $skip^{**}$ *S T* $\wedge$ *backtrack T W*)
$\vee$ $skip^{**}$ *S W* (**is** *?RB S W* $\vee$ *?R S W* $\vee$ *?SB S W* $\vee$ *?S S W*)
using *assms*
proof *induction*
case *base*
then show *?case* **by** *simp*
next
case (*step W X*) **note** *st* = *this*(*1*) **and** *bj* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*)] **and** *inv* = *this*(*4*)

have ¬*?RB S W* **and** ¬*?SB S W*
proof (*clarify*, *goal-cases*)
case (*1 T U V*)
have *skip-or-resolve*** *S T*
using *1*(*1*) **by** (*auto dest*!: *rtranclp-and-rtranclp-left*)
then show *False*
by (*metis* (*no-types*, *lifting*) *1*(*2*) *1*(*4*) *1*(*5*) *backtrack-no-*$cdcl_W$*-bj*
$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-o.bj local.bj other*
resolve rtranclp-$cdcl_W$*-all-struct-inv-inv rtranclp-skip-backtrack-backtrack*
rtranclp-skip-or-resolve-rtranclp-$cdcl_W$ *step.prems*)
next
case *2*
then show *?case* **by** (*meson assms*(*2*) $cdcl_W$*-all-struct-inv-def backtrack-no-*$cdcl_W$*-bj*
local.bj rtranclp-skip-backtrack-backtrack)
qed
then have *IH*: *?R S W* ∨ *?S S W* **using** *IH* **by** *blast*

have $cdcl_W{}^{**}$ *S W* **using** *mono-rtranclp*[*of* $cdcl_W$*-bj* $cdcl_W$] *st* **by** *blast*
then have *inv-W*: $cdcl_W$*-all-struct-inv W* **by** (*simp add*: *rtranclp-*$cdcl_W$*-all-struct-inv-inv*
step.prems)
consider
(*BT*) *X'* **where** *backtrack W X'*
| (*skip*) *no-step backtrack W* **and** *skip W X*
| (*resolve*) *no-step backtrack W* **and** *resolve W X*
using *bj* $cdcl_W$*-bj.cases* **by** *meson*
then show *?case*
proof *cases*
case (*BT X'*)
then consider
(*bt*) *backtrack W X*
| (*sk*) *skip W X*
using *bj if-can-apply-backtrack-no-more-resolve*[*of W W X' X*] *inv-W* $cdcl_W$*-bj.cases* **by** *fast*
then show *?thesis*
proof *cases*
case *bt*
then show *?thesis* **using** *IH* **by** *auto*
next
case *sk*
then show *?thesis* **using** *IH* **by** (*meson rtranclp-trans r-into-rtranclp*)
qed
next
case *skip*
then show *?thesis* **using** *IH* **by** (*meson rtranclp.rtrancl-into-rtrancl*)
next
case *resolve* **note** *no-bt* = *this*(*1*) **and** *res* = *this*(*2*)
consider
(*RS*) *T U* **where**
(λ*S T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T* **and**
resolve T U **and**
no-step backtrack T **and**
skip** *U W*
| (*S*) *skip*** *S W*
using *IH* **by** *auto*
then show *?thesis*
proof *cases*
case (*RS T U*)

have $cdcl_W^{**}$ S T
  using $RS(1)$ $cdcl_W$*-bj.resolve* $cdcl_W$*-o.bj other skip*
  mono-rtranclp[*of* (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S) $cdcl_W$ S T]
  by (*meson skip-or-resolve.cases*)
then have $cdcl_W$*-all-struct-inv* U
  by (*meson* $RS(2)$ $cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-bj.resolve* $cdcl_W$*-o.bj other*
    rtranclp-$cdcl_W$*-all-struct-inv-inv step.prems*)
{ fix U'
  assume $skip^{**}$ U U' **and** $skip^{**}$ U' W
  have $cdcl_W$*-all-struct-inv* U'
    using ‹$cdcl_W$*-all-struct-inv* U› ‹$skip^{**}$ U U'› *rtranclp-*$cdcl_W$*-all-struct-inv-inv*
      $cdcl_W$*-o.bj rtranclp-mono*[*of skip* $cdcl_W$] *other skip* **by** *blast*
  then have *no-step backtrack* U'
    using *if-can-apply-backtrack-no-more-resolve*[*OF* ‹$skip^{**}$ U' W›] *res* **by** *blast*
}
with ‹$skip^{**}$ U W›
have (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** U W
   proof *induction*
     case *base*
     then show *?case* **by** *simp*
   next
     case (*step* V W) **note** st = *this*(*1*) **and** *skip* = *this*(*2*) **and** *IH* = *this*(*3*) **and** H = *this*(*4*)
     have $\bigwedge U'$. $skip^{**}$ U' V $\Longrightarrow$ $skip^{**}$ U' W
       using *skip* **by** *auto*
     then have (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** U V
       using *IH* H **by** *blast*
     moreover have (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** V W

       by (*simp add*: *local.skip r-into-rtranclp st step.prems skip-or-resolve.intros*)
     ultimately show *?case* **by** *simp*
   qed
then show *?thesis*
  proof –
    have *f1*: $\forall$ *p pa pb pc*. $\neg$ *p* (*pa*) *pb* $\vee$ $\neg$ p^{**} *pb pc* $\vee$ p^{**} *pa pc*
      by (*meson converse-rtranclp-into-rtranclp*)
    have *skip-or-resolve* T U $\wedge$ *no-step backtrack* T
      using $RS(2)$ $RS(3)$ **by** *force*
    then have (λp *pa*. *skip-or-resolve p pa* $\wedge$ *no-step backtrack p*)** T W
      proof –
        have ($\exists$ *vr19 vr16 vr17 vr18*. *vr19* (*vr16*::'*st*) *vr17* $\wedge$ $vr19^{**}$ *vr17 vr18*
            $\wedge$ $\neg$ $vr19^{**}$ *vr16 vr18*)
          $\vee$ $\neg$ (*skip-or-resolve* T U $\wedge$ *no-step backtrack* T)
          $\vee$ $\neg$ (λ*uu uua*. *skip-or-resolve uu uua* $\wedge$ *no-step backtrack uu*)** U W
          $\vee$ (λ*uu uua*. *skip-or-resolve uu uua* $\wedge$ *no-step backtrack uu*)** T W
          by *force*
        then show *?thesis*
          by (*metis* (*no-types*) ‹(λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** U W›
            ‹*skip-or-resolve* T U $\wedge$ *no-step backtrack* T› *f1*)
      qed
    then have (λp *pa*. *skip-or-resolve p pa* $\wedge$ *no-step backtrack p*)** S W
      using $RS(1)$ **by** *force*
    then show *?thesis*
      using *no-bt res* **by** *blast*
  qed
next
  case S

{ **fix** U'
  assume *skip*** S U' **and** *skip*** U' W
  then have $cdcl_W{}^{**}$ S U'
    using *mono-rtranclp*[*of skip* $cdcl_W$ S U'] **by** (*simp add*: $cdcl_W$*-o.bj other skip*)
  then have $cdcl_W$*-all-struct-inv* U'
    by (*metis* (*no-types*, *hide-lams*) ‹$cdcl_W$*-all-struct-inv* S›
      rtranclp-$cdcl_W$*-all-struct-inv-inv*)
  then have *no-step backtrack* U'
    using *if-can-apply-backtrack-no-more-resolve*[*OF* ‹*skip*** U' W›] *res* **by** *blast*
}
with S
have (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** S W
  proof *induction*
    case *base*
    then show *?case* **by** *simp*
  next
    case (*step* V W) **note** *st* = *this*(*1*) **and** *skip* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *H* = *this*(*4*)
    have $\bigwedge U'$. *skip*** U' V $\Longrightarrow$ *skip*** U' W
      using *skip* **by** *auto*
    then have (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** S V
      using *IH H* **by** *blast*
    moreover have (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** V W

      by (*simp add*: *local.skip r-into-rtranclp st step.prems skip-or-resolve.intros*)
    ultimately show *?case* **by** *simp*
  qed
then show *?thesis* **using** *res no-bt* **by** *blast*
qed
qed
qed

The case distinction is needed, since $T \sim V$ does not imply that R^{**} T V.

lemma $cdcl_W$*-bj-strongly-confluent*:
  assumes
    $cdcl_W$*-bj*** S V **and**
    $cdcl_W$*-bj*** S T **and**
    n-s: *no-step* $cdcl_W$*-bj* V **and**
    inv: $cdcl_W$*-all-struct-inv* S
  shows $T \sim V \vee$ $cdcl_W$*-bj*** T V
  using *assms*(*2*)
proof *induction*
  case *base*
  then show *?case* **by** (*simp add*: *assms*(*1*))
next
  case (*step* T U) **note** *st* = *this*(*1*) **and** *s-o-r* = *this*(*2*) **and** *IH* = *this*(*3*)
  have $cdcl_W{}^{**}$ S T
    using *st mono-rtranclp*[*of* $cdcl_W$*-bj* $cdcl_W$] *other* **by** *blast*
  then have *lev-T*: $cdcl_W$*-M-level-inv* T
    using *inv rtranclp-*$cdcl_W$*-consistent-inv*[*of* S T]
    unfolding $cdcl_W$*-all-struct-inv-def* **by** *auto*

  consider
      (TV) $T \sim V$
    | (*bj-TV*) $cdcl_W$*-bj*** T V
    using *IH* **by** *blast*
  then show *?case*

```
proof cases
  case TV
  have no-step cdcl_W-bj T
    using ‹cdcl_W-M-level-inv T› n-s cdcl_W-bj-state-eq-compatible[of T - V] TV
    by (meson backtrack-state-eq-compatible cdcl_W-bj.simps resolve-state-eq-compatible
      skip-state-eq-compatible state-eq-ref)
  then show ?thesis
    using s-o-r by auto
next
  case bj-TV
  then obtain U' where
    T-U': cdcl_W-bj T U' and
    cdcl_W-bj** U' V
    using IH n-s s-o-r by (metis rtranclp-unfold tranclpD)
  have cdcl_W** S T
    by (metis (no-types, hide-lams) bj mono-rtranclp[of cdcl_W-bj cdcl_W] other st)
  then have inv-T: cdcl_W-all-struct-inv T
    by (metis (no-types, hide-lams) inv rtranclp-cdcl_W-all-struct-inv-inv)

  have lev-U: cdcl_W-M-level-inv U
    using s-o-r cdcl_W-consistent-inv lev-T other by blast
  show ?thesis
    using s-o-r
    proof cases
      case backtrack
      then obtain V0 where skip** T V0 and backtrack V0 V
        using IH if-can-apply-backtrack-skip-or-resolve-is-skip[OF backtrack - inv-T]
         cdcl_W-bj-decomp-resolve-skip-and-bj
         by (meson bj-TV cdcl_W-bj.backtrack inv-T lev-T n-s
           rtranclp-skip-backtrack-backtrack-end)
      then have cdcl_W-bj** T V0 and cdcl_W-bj V0 V
        using rtranclp-mono[of skip cdcl_W-bj] by blast+
      then show ?thesis
        using ‹backtrack V0 V› ‹skip** T V0› backtrack-unique inv-T local.backtrack
        rtranclp-skip-backtrack-backtrack by auto
    next
      case resolve
      then have U ∼ U'
        by (meson T-U' cdcl_W-bj.simps if-can-apply-backtrack-no-more-resolve inv-T
          resolve-skip-deterministic resolve-unique rtranclp.rtrancl-refl)
      then show ?thesis
        using ‹cdcl_W-bj** U' V› unfolding rtranclp-unfold
        by (meson T-U' bj cdcl_W-consistent-inv lev-T other state-eq-ref state-eq-sym
          tranclp-cdcl_W-bj-state-eq-compatible)
    next
      case skip
      consider
          (sk) skip T U'
        | (bt) backtrack T U'
        using T-U' by (meson cdcl_W-bj.cases local.skip resolve-skip-deterministic)
      then show ?thesis
        proof cases
          case sk
          then show ?thesis
            using ‹cdcl_W-bj** U' V› unfolding rtranclp-unfold
            by (meson T-U' bj cdcl_W-all-inv(3) cdcl_W-all-struct-inv-def inv-T local.skip other
```

tranclp-cdcl$_W$-bj-state-eq-compatible skip-unique state-eq-ref)
next
case *bt*
have *skip*$^{++}$ *T U*
using *local.skip* **by** *blast*
have *cdcl$_W$-bj U U'*
by (*meson* ‹*skip*$^{++}$ *T U*› *backtrack bt inv-T rtranclp-skip-backtrack-backtrack-end tranclp-into-rtranclp*)
then have *cdcl$_W$-bj*$^{++}$ *U V*
using ‹*cdcl$_W$-bj*** *U' V*› **by** *auto*
then show *?thesis*
by (*meson tranclp-into-rtranclp*)
qed
qed
qed
qed

lemma *cdcl$_W$-bj-unique-normal-form*:
assumes
ST: *cdcl$_W$-bj*** *S T* **and** *SU*: *cdcl$_W$-bj*** *S U* **and**
n-s-U: *no-step cdcl$_W$-bj U* **and**
n-s-T: *no-step cdcl$_W$-bj T* **and**
inv: *cdcl$_W$-all-struct-inv S*
shows $T \sim U$
proof –
have $T \sim U \vee$ *cdcl$_W$-bj*** *T U*
using *ST SU cdcl$_W$-bj-strongly-confluent inv n-s-U* **by** *blast*
then show *?thesis*
by (*metis* (*no-types*) *n-s-T rtranclp-unfold state-eq-ref tranclp-unfold-begin*)
qed

lemma *full-cdcl$_W$-bj-unique-normal-form*:
assumes *full cdcl$_W$-bj S T* **and** *full cdcl$_W$-bj S U* **and**
inv: *cdcl$_W$-all-struct-inv S*
shows $T \sim U$
using *cdcl$_W$-bj-unique-normal-form assms* **unfolding** *full-def* **by** *blast*

6.2.3 CDCL with Merging

inductive *cdcl$_W$-merge-restart* :: *'st* ⇒ *'st* ⇒ *bool* **where**
fw-r-propagate: *propagate S S'* ⟹ *cdcl$_W$-merge-restart S S'* |
fw-r-conflict: *conflict S T* ⟹ *full cdcl$_W$-bj T U* ⟹ *cdcl$_W$-merge-restart S U* |
fw-r-decide: *decide S S'* ⟹ *cdcl$_W$-merge-restart S S'*|
fw-r-rf: *cdcl$_W$-rf S S'* ⟹ *cdcl$_W$-merge-restart S S'*

lemma *rtranclp-cdcl$_W$-bj-rtranclp-cdcl$_W$*:
cdcl$_W$-bj** *S T* ⟹ *cdcl$_W$*** *S T*
using *mono-rtranclp*[*of cdcl$_W$-bj cdcl$_W$*] **by** *blast*

lemma *cdcl$_W$-merge-restart-cdcl$_W$*:
assumes *cdcl$_W$-merge-restart S T*
shows *cdcl$_W$*** *S T*
using *assms*
proof *induction*
case (*fw-r-conflict S T U*) **note** *confl* = *this*(*1*) **and** *bj* = *this*(*2*)

have $cdcl_W$ *S T* **using** *confl* **by** (*simp add*: $cdcl_W$*.intros r-into-rtranclp*)
moreover
have $cdcl_W$*-bj*** *T U* **using** *bj* **unfolding** *full-def* **by** *auto*
then have $cdcl_W{}^{**}$ *T U* **using** *rtranclp-*$cdcl_W$*-bj-rtranclp-*$cdcl_W$ **by** *blast*
ultimately show *?case* **by** *auto*
qed (*simp-all add*: $cdcl_W$*-o.intros* $cdcl_W$*.intros r-into-rtranclp*)

lemma $cdcl_W$*-merge-restart-conflicting-true-or-no-step*:
assumes $cdcl_W$*-merge-restart S T*
shows *conflicting T = None* ∨ *no-step* $cdcl_W$ *T*
using *assms*
proof *induction*
case (*fw-r-conflict S T U*) **note** *confl* = *this*(*1*) **and** *n-s* = *this*(*2*)
{ fix *D V*
assume $cdcl_W$ *U V* **and** *conflicting U = Some D*
then have *False*
using *n-s* **unfolding** *full-def*
by (*induction rule*: $cdcl_W$*-all-rules-induct*)
(*auto dest*!: $cdcl_W$*-bj.intros elim*: *decideE propagateE conflictE forgetE restartE*)
}
then show *?case* **by** (*cases conflicting U*) *fastforce+*
qed (*auto simp add*: $cdcl_W$*-rf.simps elim*: *propagateE decideE restartE forgetE*)

inductive $cdcl_W$*-merge* :: *'st* ⇒ *'st* ⇒ *bool* **where**
fw-propagate: *propagate S S'* ⟹ $cdcl_W$*-merge S S'* |
fw-conflict: *conflict S T* ⟹ *full* $cdcl_W$*-bj T U* ⟹ $cdcl_W$*-merge S U* |
fw-decide: *decide S S'* ⟹ $cdcl_W$*-merge S S'*|
fw-forget: *forget S S'* ⟹ $cdcl_W$*-merge S S'*

lemma $cdcl_W$*-merge-*$cdcl_W$*-merge-restart*:
$cdcl_W$*-merge S T* ⟹ $cdcl_W$*-merge-restart S T*
by (*meson* $cdcl_W$*-merge.cases* $cdcl_W$*-merge-restart.simps forget*)

lemma *rtranclp-*$cdcl_W$*-merge-tranclp-*$cdcl_W$*-merge-restart*:
$cdcl_W$*-merge*** *S T* ⟹ $cdcl_W$*-merge-restart*** *S T*
using *rtranclp-mono*[*of* $cdcl_W$*-merge* $cdcl_W$*-merge-restart*] $cdcl_W$*-merge-*$cdcl_W$*-merge-restart* **by** *blast*

lemma $cdcl_W$*-merge-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge S T* ⟹ $cdcl_W{}^{**}$ *S T*
using $cdcl_W$*-merge-*$cdcl_W$*-merge-restart* $cdcl_W$*-merge-restart-*$cdcl_W$ **by** *blast*

lemma *rtranclp-*$cdcl_W$*-merge-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge*** *S T* ⟹ $cdcl_W{}^{**}$ *S T*
using *rtranclp-mono*[*of* $cdcl_W$*-merge* $cdcl_W{}^{**}$] $cdcl_W$*-merge-rtranclp-*$cdcl_W$ **by** *auto*

lemmas *rulesE* =
skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE

lemma $cdcl_W$*-all-struct-inv-tranclp-*$cdcl_W$*-merge-tranclp-*$cdcl_W$*-merge-*$cdcl_W$*-all-struct-inv*:
assumes
inv: $cdcl_W$*-all-struct-inv b*
$cdcl_W$*-merge*$^{++}$ *b a*
shows (λ*S T*. $cdcl_W$*-all-struct-inv S* ∧ $cdcl_W$*-merge S T*)$^{++}$ *b a*
using *assms*(*2*)
proof *induction*
case *base*

then show *?case* **using** *inv* **by** *auto*
next
case (*step c d*) **note** *st* = *this*(*1*) **and** *fw* = *this*(*2*) **and** *IH* = *this*(*3*)
have $cdcl_W$*-all-struct-inv c*
using *tranclp-into-rtranclp*[*OF st*] $cdcl_W$*-merge-rtranclp-*$cdcl_W$
assms(*1*) *rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-mono*[*of* $cdcl_W$*-merge* ${cdcl_W}^{**}$] **by** *fastforce*
then have ($\lambda S\ T.\ cdcl_W$*-all-struct-inv* $S \wedge cdcl_W$*-merge* $S\ T)^{++}\ c\ d$
using *fw* **by** *auto*
then show *?case* **using** *IH* **by** *auto*
qed

lemma *backtrack-is-full1-*$cdcl_W$*-bj*:
assumes *bt*: *backtrack S T* **and** *inv*: $cdcl_W$*-M-level-inv S*
shows *full1* $cdcl_W$*-bj S T*
using *bt inv backtrack-no-*$cdcl_W$*-bj* **unfolding** *full1-def* **by** *blast*

lemma *rtrancl-*$cdcl_W$*-conflicting-true-*$cdcl_W$*-merge-restart*:
assumes ${cdcl_W}^{**}\ S\ V$ **and** *inv*: $cdcl_W$*-M-level-inv S* **and** *conflicting S* = *None*
shows ($cdcl_W$*-merge-restart*** $S\ V \wedge$ *conflicting V* = *None*)
$\vee$ ($\exists\ T\ U.\ cdcl_W$*-merge-restart*** $S\ T \wedge$ *conflicting V* $\neq$ *None* $\wedge$ *conflict T U* $\wedge\ cdcl_W$*-bj*** $U\ V$)
using *assms*
proof *induction*
case *base*
then show *?case* **by** *simp*
next
case (*step U V*) **note** *st* = *this*(*1*) **and** $cdcl_W$ = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*−)] **and**
confl[*simp*] = *this*(*5*) **and** *inv* = *this*(*4*)
from $cdcl_W$
show *?case*
proof (*cases*)
case *propagate*
moreover then have *conflicting U* = *None* **and** *conflicting V* = *None*
by (*auto elim*: *propagateE*)
ultimately show *?thesis* **using** *IH* $cdcl_W$*-merge-restart.fw-r-propagate*[*of U V*] **by** *auto*
next
case *conflict*
moreover then have *conflicting U* = *None* **and** *conflicting V* $\neq$ *None*
by (*auto elim*!: *conflictE simp del*: *state-simp simp*: *state-eq-def*)
ultimately show *?thesis* **using** *IH* **by** *auto*
next
case *other*
then show *?thesis*
proof *cases*
case *decide*
then show *?thesis* **using** *IH* $cdcl_W$*-merge-restart.fw-r-decide*[*of U V*] **by** (*auto elim*: *decideE*)
next
case *bj*
moreover {
assume *skip-or-resolve U V*
have *f1*: $cdcl_W$*-bj*$^{++}$ $U\ V$
by (*simp add*: *local.bj tranclp.r-into-trancl*)
obtain $T\ T'$ :: *'st* **where**
f2: $cdcl_W$*-merge-restart*** $S\ U$
$\vee\ cdcl_W$*-merge-restart*** $S\ T \wedge$ *conflicting U* $\neq$ *None*
$\wedge$ *conflict T T'* $\wedge\ cdcl_W$*-bj*** $T'\ U$
using *IH confl* **by** *blast*

```
          have conflicting V ≠ None ∧ conflicting U ≠ None
            using ‹skip-or-resolve U V›
            by (auto simp: skip-or-resolve.simps state-eq-def elim!: skipE resolveE
              simp del: state-simp)
          then have ?thesis
            by (metis (full-types) IH f1 rtranclp-trans tranclp-into-rtranclp)
        }
        moreover {
          assume backtrack U V
          then have conflicting U ≠ None by (auto elim: backtrackE)
          then obtain T T' where
            cdcl_W-merge-restart** S T and
            conflicting U ≠ None and
            conflict T T' and
            cdcl_W-bj** T' U
            using IH confl by meson
          have invU: cdcl_W-M-level-inv U
            using inv rtranclp-cdcl_W-consistent-inv step.hyps(1) by blast
          then have conflicting V = None
            using ‹backtrack U V› inv by (auto elim: backtrackE
              simp: cdcl_W-M-level-inv-decomp)
          have full cdcl_W-bj T' V
            apply (rule rtranclp-fullI[of cdcl_W-bj T' U V])
              using ‹cdcl_W-bj** T' U› apply fast
            using ‹backtrack U V› backtrack-is-full1-cdcl_W-bj invU unfolding full1-def full-def
            by blast
          then have ?thesis
            using cdcl_W-merge-restart.fw-r-conflict[of T T' V] ‹conflict T T'›
            ‹cdcl_W-merge-restart** S T› ‹conflicting V = None› by auto
        }
        ultimately show ?thesis by (auto simp: cdcl_W-bj.simps)
      qed
    next
      case rf
      moreover then have conflicting U = None and conflicting V = None
        by (auto simp: cdcl_W-rf.simps elim: restartE forgetE)
      ultimately show ?thesis using IH cdcl_W-merge-restart.fw-r-rf[of U V] by auto
    qed
qed

lemma no-step-cdcl_W-no-step-cdcl_W-merge-restart: no-step cdcl_W S ⟹ no-step cdcl_W-merge-restart
S
  by (auto simp: cdcl_W.simps cdcl_W-merge-restart.simps cdcl_W-o.simps cdcl_W-bj.simps)

lemma no-step-cdcl_W-merge-restart-no-step-cdcl_W:
  assumes
    conflicting S = None and
    cdcl_W-M-level-inv S and
    no-step cdcl_W-merge-restart S
  shows no-step cdcl_W S
proof -
  { fix S'
    assume conflict S S'
    then have cdcl_W S S' using cdcl_W.conflict by auto
    then have cdcl_W-M-level-inv S'
      using assms(2) cdcl_W-consistent-inv by blast
```

then obtain S'' **where** *full* $cdcl_W$*-bj* S' S''
using $cdcl_W$*-bj-exists-normal-form*[*of* S'] **by** *auto*
then have *False*
using ‹*conflict* S S'› *assms*(*3*) *fw-r-conflict* **by** *blast*
}
then show *?thesis*
using *assms* **unfolding** $cdcl_W$*.simps* $cdcl_W$*-merge-restart.simps* $cdcl_W$*-o.simps* $cdcl_W$*-bj.simps*
by (*auto elim*: *skipE resolveE backtrackE conflictE decideE restartE*)
qed

lemma $cdcl_W$*-merge-restart-no-step-*$cdcl_W$*-bj*:
assumes
$cdcl_W$*-merge-restart* S T
shows *no-step* $cdcl_W$*-bj* T
using *assms*
by (*induction rule*: $cdcl_W$*-merge-restart.induct*)
(*force simp*: $cdcl_W$*-bj.simps* $cdcl_W$*-rf.simps* $cdcl_W$*-merge-restart.simps full-def*
elim!: *rulesE*)+

lemma *rtranclp-*$cdcl_W$*-merge-restart-no-step-*$cdcl_W$*-bj*:
assumes
$cdcl_W$*-merge-restart*** S T **and**
conflicting S = *None*
shows *no-step* $cdcl_W$*-bj* T
using *assms* **unfolding** *rtranclp-unfold*
apply (*elim disjE*)
apply (*force simp*: $cdcl_W$*-bj.simps* $cdcl_W$*-rf.simps elim*!: *rulesE*)
by (*auto simp*: *tranclp-unfold-end simp*: $cdcl_W$*-merge-restart-no-step-*$cdcl_W$*-bj*)

If *conflicting* $S \neq$ *None*, we cannot say anything.

Remark that this theorem does not say anything about well-foundedness: even if you know that one relation is well-founded, it only states that the normal forms are shared.

lemma *conflicting-true-full-*$cdcl_W$*-iff-full-*$cdcl_W$*-merge*:
assumes *confl*: *conflicting* S = *None* **and** *lev*: $cdcl_W$*-M-level-inv* S
shows *full* $cdcl_W$ S V $\longleftrightarrow$ *full* $cdcl_W$*-merge-restart* S V
proof
assume *full*: *full* $cdcl_W$*-merge-restart* S V
then have *st*: $cdcl_W$** S V
using *rtranclp-mono*[*of* $cdcl_W$*-merge-restart* $cdcl_W$**] $cdcl_W$*-merge-restart-*$cdcl_W$
unfolding *full-def* **by** *auto*

have *n-s*: *no-step* $cdcl_W$*-merge-restart* V
using *full* **unfolding** *full-def* **by** *auto*
have *n-s-bj*: *no-step* $cdcl_W$*-bj* V
using *rtranclp-*$cdcl_W$*-merge-restart-no-step-*$cdcl_W$*-bj confl full* **unfolding** *full-def* **by** *auto*
have $\bigwedge S'$. *conflict* V S' $\Longrightarrow$ $cdcl_W$*-M-level-inv* S'
using $cdcl_W$*.conflict* $cdcl_W$*-consistent-inv lev rtranclp-*$cdcl_W$*-consistent-inv st* **by** *blast*
then have $\bigwedge S'$. *conflict* V S' $\Longrightarrow$ *False*
using *n-s n-s-bj* $cdcl_W$*-bj-exists-normal-form* $cdcl_W$*-merge-restart.simps* **by** *meson*
then have *n-s-*$cdcl_W$: *no-step* $cdcl_W$ V
using *n-s n-s-bj* **by** (*auto simp*: $cdcl_W$*.simps* $cdcl_W$*-o.simps* $cdcl_W$*-merge-restart.simps*)
then show *full* $cdcl_W$ S V **using** *st* **unfolding** *full-def* **by** *auto*
next
assume *full*: *full* $cdcl_W$ S V
have *no-step* $cdcl_W$*-merge-restart* V

using *full no-step-cdcl*$_W$*-no-step-cdcl*$_W$*-merge-restart* **unfolding** *full-def* **by** *blast*
moreover
consider
(*fw*) *cdcl*$_W$*-merge-restart*** *S V* **and** *conflicting V = None*
| (*bj*) *T U* **where**
cdcl$_W$*-merge-restart*** *S T* **and**
conflicting V ≠ None **and**
conflict T U **and**
cdcl$_W$*-bj*** *U V*
using *full rtrancl-cdcl*$_W$*-conflicting-true-cdcl*$_W$*-merge-restart confl lev* **unfolding** *full-def*
by *meson*
then have *cdcl*$_W$*-merge-restart*** *S V*
proof *cases*
case *fw*
then show *?thesis* **by** *fast*
next
case (*bj T U*)
have *no-step cdcl*$_W$*-bj V*
using *full* **unfolding** *full-def* **by** (*meson cdcl*$_W$*-o.bj other*)
then have *full cdcl*$_W$*-bj U V*
using ‹ *cdcl*$_W$*-bj*** *U V*› **unfolding** *full-def* **by** *auto*
then have *cdcl*$_W$*-merge-restart T V*
using ‹*conflict T U*› *cdcl*$_W$*-merge-restart.fw-r-conflict* **by** *blast*
then show *?thesis* **using** ‹*cdcl*$_W$*-merge-restart*** *S T*› **by** *auto*
qed
ultimately show *full cdcl*$_W$*-merge-restart S V* **unfolding** *full-def* **by** *fast*
qed

lemma *init-state-true-full-cdcl*$_W$*-iff-full-cdcl*$_W$*-merge*:
shows *full cdcl*$_W$ (*init-state N*) *V* ⟷ *full cdcl*$_W$*-merge-restart* (*init-state N*) *V*
by (*rule conflicting-true-full-cdcl*$_W$*-iff-full-cdcl*$_W$*-merge*) *auto*

6.2.4 CDCL with Merge and Strategy

The intermediate step

inductive *cdcl*$_W$*-s'* :: *'st ⇒ 'st ⇒ bool* **where**
conflict': *full1 cdcl*$_W$*-cp S S'* ⟹ *cdcl*$_W$*-s' S S'* |
decide': *decide S S'* ⟹ *no-step cdcl*$_W$*-cp S* ⟹ *full cdcl*$_W$*-cp S' S''* ⟹ *cdcl*$_W$*-s' S S''* |
bj': *full1 cdcl*$_W$*-bj S S'* ⟹ *no-step cdcl*$_W$*-cp S* ⟹ *full cdcl*$_W$*-cp S' S''* ⟹ *cdcl*$_W$*-s' S S''*

inductive-cases *cdcl*$_W$*-s'E*: *cdcl*$_W$*-s' S T*

lemma *rtranclp-cdcl*$_W$*-bj-full1-cdclp-cdcl*$_W$*-stgy*:
cdcl$_W$*-bj*** *S S'* ⟹ *full cdcl*$_W$*-cp S' S''* ⟹ *cdcl*$_W$*-stgy*** *S S''*
proof (*induction rule*: *converse-rtranclp-induct*)
case *base*
then show *?case* **by** (*metis cdcl*$_W$*-stgy.conflict' full-unfold rtranclp.simps*)
next
case (*step T U*) **note** *st = this*(*2*) **and** *bj = this*(*1*) **and** *IH = this*(*3*)[*OF this*(*4*)]
have *no-step cdcl*$_W$*-cp T*
using *bj* **by** (*auto simp add*: *cdcl*$_W$*-bj.simps cdcl*$_W$*-cp.simps elim*!: *rulesE*)
consider
(*U*) *U = S'*
| (*U'*) *U'* **where** *cdcl*$_W$*-bj U U'* **and** *cdcl*$_W$*-bj*** *U' S'*
using *st* **by** (*metis converse-rtranclpE*)

then show *?case*
proof *cases*
case *U*
then show *?thesis*
using ⟨*no-step* $cdcl_W$*-cp T*⟩ $cdcl_W$*-o.bj local.bj other′ step.prems* **by** (*meson r-into-rtranclp*)
next
case *U′* **note** *U′* = *this*(*1*)
have *no-step* $cdcl_W$*-cp U*
using *U′* **by** (*fastforce simp*: $cdcl_W$*-cp.simps* $cdcl_W$*-bj.simps elim*: *rulesE*)
then have *full* $cdcl_W$*-cp U U*
by (*simp add*: *full-unfold*)
then have $cdcl_W$*-stgy T U*
using ⟨*no-step* $cdcl_W$*-cp T*⟩ $cdcl_W$*-stgy.simps local.bj* $cdcl_W$*-o.bj* **by** *meson*
then show *?thesis* **using** *IH* **by** *auto*
qed
qed

lemma $cdcl_W$*-s′-is-rtranclp-*$cdcl_W$*-stgy*:
$cdcl_W$*-s′ S T* ⟹ $cdcl_W$*-stgy*** *S T*
apply (*induction rule*: $cdcl_W$*-s′.induct*)
apply (*auto intro*: $cdcl_W$*-stgy.intros*)[]
apply (*meson decide other′ r-into-rtranclp*)
by (*metis full1-def rtranclp-*$cdcl_W$*-bj-full1-cdclp-*$cdcl_W$*-stgy tranclp-into-rtranclp*)

lemma $cdcl_W$*-cp-*$cdcl_W$*-bj-bissimulation*:
assumes
full $cdcl_W$*-cp T U* **and**
$cdcl_W$*-bj*** *T T′* **and**
$cdcl_W$*-all-struct-inv T* **and**
no-step $cdcl_W$*-bj T′*
shows *full* $cdcl_W$*-cp T′ U*
∨ (∃ *U′ U″*. *full* $cdcl_W$*-cp T′ U″* ∧ *full1* $cdcl_W$*-bj U U′* ∧ *full* $cdcl_W$*-cp U′ U″*
∧ $cdcl_W$*-s′*** *U U″*)
using *assms*(*2*,*1*,*3*,*4*)
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *blast*
next
case (*step T′ T″*) **note** *st* = *this*(*1*) **and** *bj* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*,*5*)] **and**
full = *this*(*4*) **and** *inv* = *this*(*5*)
have $cdcl_W$*-bj*** *T T″*
using *local.bj st* **by** *auto*
then have $cdcl_W$** *T T″*
using *rtranclp-*$cdcl_W$*-bj-rtranclp-*$cdcl_W$ **by** *blast*
then have *inv-T″*: $cdcl_W$*-all-struct-inv T″*
using *inv rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
have $cdcl_W$*-bj*$^{++}$ *T T″*
using *local.bj st* **by** *auto*
have *full1* $cdcl_W$*-bj T T″*
by (*metis* ⟨$cdcl_W$*-bj*$^{++}$ *T T″*⟩ *full1-def step.prems*(*3*))
then have *T* = *U*
proof −
obtain *Z* **where** $cdcl_W$*-bj T Z*
using ⟨$cdcl_W$*-bj*$^{++}$ *T T″*⟩ **by** (*blast dest*: *tranclpD*)
{ assume $cdcl_W$*-cp*$^{++}$ *T U*
then obtain *Z′* **where** $cdcl_W$*-cp T Z′*

      by (*meson tranclpD*)
    then have *False*
      using ‹$cdcl_W$*-bj T Z*› **by** (*fastforce simp*: $cdcl_W$*-bj.simps* $cdcl_W$*-cp.simps*
        elim: *rulesE*)
  }
  then show *?thesis*
    using *full* **unfolding** *full-def rtranclp-unfold* **by** *blast*
qed
obtain U'' **where** *full* $cdcl_W$*-cp* T'' U''
  using $cdcl_W$*-cp-normalized-element-all-inv inv-*T'' **by** *blast*
moreover then have $cdcl_W$*-stgy*** *U* U''
  by (*metis* ‹*T* = *U*› ‹$cdcl_W$*-bj*$^{++}$ *T* T''› *rtranclp-*$cdcl_W$*-bj-full1-cdclp-*$cdcl_W$*-stgy rtranclp-unfold*)
moreover have $cdcl_W$*-s*$'^{**}$ *U* U''
  proof −
    obtain *ss* :: *'st* ⇒ *'st* **where**
      f1: ∀ *x2*. (∃ *v3*. $cdcl_W$*-cp x2 v3*) = $cdcl_W$*-cp x2* (*ss x2*)
      by *moura*
    have ¬ $cdcl_W$*-cp U* (*ss U*)
      by (*meson full full-def*)
    then show *?thesis*
      using *f1* **by** (*metis* (*no-types*) ‹*T* = *U*› ‹*full1* $cdcl_W$*-bj T* T''› *bj'* *calculation*(*1*)
        r-into-rtranclp)
  qed
ultimately show *?case*
  using ‹*full1* $cdcl_W$*-bj T* T''› ‹*full* $cdcl_W$*-cp* T'' U''› **unfolding** ‹*T* = *U*› **by** *blast*
qed

lemma $cdcl_W$*-cp-*$cdcl_W$*-bj-bissimulation'*:
  assumes
    full $cdcl_W$*-cp T U* **and**
    $cdcl_W$*-bj*** *T* T' **and**
    $cdcl_W$*-all-struct-inv T* **and**
    no-step $cdcl_W$*-bj* T'
  shows *full* $cdcl_W$*-cp* T' *U*
    ∨ (∃ U'. *full1* $cdcl_W$*-bj U* U' ∧ (∀ U''. *full* $cdcl_W$*-cp* U' U'' ⟶ *full* $cdcl_W$*-cp* T' U''
      ∧ $cdcl_W$*-s*$'^{**}$ *U* U''))
  using *assms*(*2*,*1*,*3*,*4*)
proof (*induction rule*: *rtranclp-induct*)
  case *base*
  then show *?case* **by** *blast*
next
  case (*step* T' T'') **note** *st* = *this*(*1*) **and** *bj* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*,*5*)] **and**
    full = *this*(*4*) **and** *inv* = *this*(*5*)
  have $cdcl_W{}^{**}$ *T* T''
    by (*metis local.bj rtranclp.simps rtranclp-*$cdcl_W$*-bj-rtranclp-*$cdcl_W$ *st*)
  then have *inv-*T'': $cdcl_W$*-all-struct-inv* T''
    using *inv rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
  have $cdcl_W$*-bj*$^{++}$ *T* T''
    using *local.bj st* **by** *auto*
  have *full1* $cdcl_W$*-bj T* T''
    by (*metis* ‹$cdcl_W$*-bj*$^{++}$ *T* T''› *full1-def step.prems*(*3*))
  then have *T* = *U*
    proof −
      obtain *Z* **where** $cdcl_W$*-bj T Z*
        using ‹$cdcl_W$*-bj*$^{++}$ *T* T''› **by** (*blast dest*: *tranclpD*)
      { assume $cdcl_W$*-cp*$^{++}$ *T U*

then obtain Z' **where** $cdcl_W$*-cp* T Z'
by (*meson tranclpD*)
then have *False*
using ‹$cdcl_W$*-bj* T Z› **by** (*fastforce simp*: $cdcl_W$*-bj.simps* $cdcl_W$*-cp.simps elim*: *rulesE*)
}
then show *?thesis*
using *full* **unfolding** *full-def rtranclp-unfold* **by** *blast*
qed
{ fix U''
assume *full* $cdcl_W$*-cp* T'' U''
moreover then have $cdcl_W$*-stgy*** U U''
by (*metis* ‹$T = U$› ‹$cdcl_W$*-bj*$^{++}$ T T''› *rtranclp-*$cdcl_W$*-bj-full1-cdclp-*$cdcl_W$*-stgy rtranclp-unfold*)
moreover have $cdcl_W$*-s*$'^{**}$ U U''
proof −
obtain *ss* :: *'st* ⇒ *'st* **where**
f1: $\forall x2.\ (\exists v3.\ cdcl_W\text{-}cp\ x2\ v3) = cdcl_W\text{-}cp\ x2\ (ss\ x2)$
by *moura*
have ¬ $cdcl_W$*-cp* U (*ss* U)
by (*meson assms*(*1*) *full-def*)
then show *?thesis*
using *f1* **by** (*metis* (*no-types*) ‹$T = U$› ‹*full1* $cdcl_W$*-bj* T T''› *bj*$'$ *calculation*(*1*)
r-into-rtranclp)
qed
ultimately have *full1* $cdcl_W$*-bj* U T'' **and** $cdcl_W$*-s*$'^{**}$ T'' U''
using ‹*full1* $cdcl_W$*-bj* T T''› ‹*full* $cdcl_W$*-cp* T'' U''› **unfolding** ‹$T = U$›
apply *blast*
by (*metis* ‹*full* $cdcl_W$*-cp* T'' U''› $cdcl_W$*-s*$'$*.simps full-unfold rtranclp.simps*)
}
then show *?case*
using ‹*full1* $cdcl_W$*-bj* T T''› *full bj*$'$ **unfolding** ‹$T = U$› *full-def* **by** (*metis r-into-rtranclp*)
qed

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s*$'$*-connected*:
assumes $cdcl_W$*-stgy* S U **and** $cdcl_W$*-all-struct-inv* S
shows $cdcl_W$*-s*$'$ S U
∨ (∃ U'. *full1* $cdcl_W$*-bj* U U' ∧ (∀ U''. *full* $cdcl_W$*-cp* U' U'' ⟶ $cdcl_W$*-s*$'$ S U''))
using *assms*
proof (*induction rule*: $cdcl_W$*-stgy.induct*)
case (*conflict*$'$ T)
then have $cdcl_W$*-s*$'$ S T
using $cdcl_W$*-s*$'$*.conflict*$'$ **by** *blast*
then show *?case*
by *blast*
next
case (*other*$'$ T U) **note** $o = this(1)$ **and** $n\text{-}s = this(2)$ **and** $full = this(3)$ **and** $inv = this(4)$
show *?case*
using *o*
proof *cases*
case *decide*
then show *?thesis* **using** $cdcl_W$*-s*$'$*.simps full n-s* **by** *blast*
next
case *bj*
have *inv-T*: $cdcl_W$*-all-struct-inv* T
using $cdcl_W$*-all-struct-inv-inv o other other*$'$*.prems* **by** *blast*
consider
(*cp*) *full* $cdcl_W$*-cp* T U **and** *no-step* $cdcl_W$*-bj* T

| (*fbj*) T' **where** *full1* $cdcl_W$*-bj T* T'
apply (*cases no-step* $cdcl_W$*-bj T*)
using *full* **apply** *blast*
using $cdcl_W$*-bj-exists-normal-form*[*of T*] *inv-T* **unfolding** $cdcl_W$*-all-struct-inv-def*
by (*metis full-unfold*)
then show *?thesis*
proof *cases*
case *cp*
then show *?thesis*
proof −
obtain *ss* :: $'st \Rightarrow 'st$ **where**
f1: $\forall$ *s sa sb*. ($\neg$ *full1* $cdcl_W$*-bj s sa* $\vee$ $cdcl_W$*-cp s* (*ss s*) $\vee$ $\neg$ *full* $cdcl_W$*-cp sa sb*)
$\vee$ $cdcl_W$*-s'* *s sb*
using *bj'* **by** *moura*
have *full1* $cdcl_W$*-bj S T*
by (*simp add*: *cp*(*2*) *full1-def local.bj tranclp.r-into-trancl*)
then show *?thesis*
using *f1 full n-s* **by** *blast*
qed
next
case (*fbj* U')
then have *full1* $cdcl_W$*-bj S* U'
using *bj* **unfolding** *full1-def* **by** *auto*
moreover have *no-step* $cdcl_W$*-cp S*
using *n-s* **by** *blast*
moreover have $T = U$
using *full fbj* **unfolding** *full1-def full-def rtranclp-unfold*
by (*force dest*!: *tranclpD simp*:$cdcl_W$*-bj.simps elim*: *rulesE*)
ultimately show *?thesis* **using** $cdcl_W$*-s'.bj'*[*of S* U'] **using** *fbj* **by** *blast*
qed
qed
qed

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s'-connected'*:
assumes $cdcl_W$*-stgy S U* **and** $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-s' S U*
$\vee$ ($\exists$ U' U''. $cdcl_W$*-s' S* U'' $\wedge$ *full1* $cdcl_W$*-bj U* U' $\wedge$ *full* $cdcl_W$*-cp* U' U'')
using *assms*
proof (*induction rule*: $cdcl_W$*-stgy.induct*)
case (*conflict' T*)
then have $cdcl_W$*-s' S T*
using $cdcl_W$*-s'.conflict'* **by** *blast*
then show *?case*
by *blast*
next
case (*other' T U*) **note** *o* = *this*(*1*) **and** *n-s* = *this*(*2*) **and** *full* = *this*(*3*) **and** *inv* = *this*(*4*)
show *?case*
using *o*
proof *cases*
case *decide*
then show *?thesis* **using** $cdcl_W$*-s'.simps full n-s* **by** *blast*
next
case *bj*
have $cdcl_W$*-all-struct-inv T*
using $cdcl_W$*-all-struct-inv-inv o other other'.prems* **by** *blast*
then obtain T' **where** T': *full* $cdcl_W$*-bj T* T'

using $cdcl_W$*-bj-exists-normal-form* **unfolding** *full-def* $cdcl_W$*-all-struct-inv-def* **by** *metis*
then have *full* $cdcl_W$*-bj* S T'
proof −
have *f1*: $cdcl_W$*-bj*** T T' ∧ *no-step* $cdcl_W$*-bj* T'
by (*metis* (*no-types*) T' *full-def*)
then have $cdcl_W$*-bj*** S T'
by (*meson converse-rtranclp-into-rtranclp local.bj*)
then show *?thesis*
using *f1* **by** (*simp add*: *full-def*)
qed
have $cdcl_W$*-bj*** T T'
using T' **unfolding** *full-def* **by** *simp*
have $cdcl_W$*-all-struct-inv* T
using $cdcl_W$*-all-struct-inv-inv o other other'.prems* **by** *blast*
then consider
($T'U$) *full* $cdcl_W$*-cp* T' U
| (U) U' U'' **where**
full $cdcl_W$*-cp* T' U'' **and**
full1 $cdcl_W$*-bj* U U' **and**
full $cdcl_W$*-cp* U' U'' **and**
$cdcl_W$*-s'*** U U''
using $cdcl_W$*-cp-*$cdcl_W$*-bj-bissimulation*[*OF full* ‹$cdcl_W$*-bj*** T T'›] T' **unfolding** *full-def*
by *blast*
then show *?thesis* **by** (*metis* T' $cdcl_W$*-s'.simps full-fullI local.bj n-s*)
qed
qed

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s'-no-step*:
assumes $cdcl_W$*-stgy* S U **and** $cdcl_W$*-all-struct-inv* S **and** *no-step* $cdcl_W$*-bj* U
shows $cdcl_W$*-s'* S U
using $cdcl_W$*-stgy-*$cdcl_W$*-s'-connected*[*OF assms*(*1*,*2*)] *assms*(*3*)
by (*metis* (*no-types*, *lifting*) *full1-def tranclpD*)

lemma *rtranclp-*$cdcl_W$*-stgy-connected-to-rtranclp-*$cdcl_W$*-s'*:
assumes $cdcl_W$*-stgy*** S U **and** *inv*: $cdcl_W$*-M-level-inv* S
shows $cdcl_W$*-s'*** S U ∨ (∃ T. $cdcl_W$*-s'*** S T ∧ $cdcl_W$*-bj*$^{++}$ T U ∧ *conflicting* $U \neq$ *None*)
using *assms*(*1*)
proof *induction*
case *base*
then show *?case* **by** *simp*
next
case (*step* T V) **note** *st* = *this*(*1*) **and** *o* = *this*(*2*) **and** *IH* = *this*(*3*)
from *o* **show** *?case*
proof *cases*
case *conflict'*
then have *f2*: $cdcl_W$*-s'* T V
using $cdcl_W$*-s'.conflict'* **by** *blast*
obtain *ss* :: *'st* **where**
f3: $S = T$ ∨ $cdcl_W$*-stgy*** S *ss* ∧ $cdcl_W$*-stgy ss* T
by (*metis* (*full-types*) *rtranclp.simps st*)
obtain *ssa* :: *'st* **where**
ssa: $cdcl_W$*-cp* T *ssa*
using *conflict'* **by** (*metis* (*no-types*) *full1-def tranclpD*)
have ∀ *s*. ¬ *full* $cdcl_W$*-cp* *s* T
by (*meson ssa full-def*)
then have $S = T$

```
    by (metis (full-types) f3 ssa cdcl_W-stgy.cases full1-def)
  then show ?thesis
    using f2 by blast
next
  case (other' U) note o = this(1) and n-s = this(2) and full = this(3)
  then show ?thesis
    using o
    proof (cases rule: cdcl_W-o-rule-cases)
      case decide
      then have cdcl_W-s'** S T
        using IH by (auto elim: rulesE)
      then show ?thesis
        by (meson decide decide' full n-s rtranclp.rtrancl-into-rtrancl)
    next
      case backtrack
      consider
          (s') cdcl_W-s'** S T
        | (bj) S' where cdcl_W-s'** S S' and cdcl_W-bj++ S' T and conflicting T ≠ None
        using IH by blast
      then show ?thesis
        proof cases
          case s'
          moreover
            have cdcl_W-M-level-inv T
              using inv local.step(1) rtranclp-cdcl_W-stgy-consistent-inv by auto
            then have full1 cdcl_W-bj T U
              using backtrack-is-full1-cdcl_W-bj backtrack by blast
            then have cdcl_W-s' T V
             using full bj' n-s by blast
          ultimately show ?thesis by auto
        next
          case (bj S') note S-S' = this(1) and bj-T = this(2)
          have no-step cdcl_W-cp S'
            using bj-T by (fastforce simp: cdcl_W-cp.simps cdcl_W-bj.simps dest!: tranclpD
              elim: rulesE)
          moreover
            have cdcl_W-M-level-inv T
              using inv local.step(1) rtranclp-cdcl_W-stgy-consistent-inv by auto
            then have full1 cdcl_W-bj T U
              using backtrack-is-full1-cdcl_W-bj backtrack by blast
            then have full1 cdcl_W-bj S' U
              using bj-T unfolding full1-def by fastforce
          ultimately have cdcl_W-s' S' V using full by (simp add: bj')
          then show ?thesis using S-S' by auto
        qed
    next
      case skip
      then have [simp]: U = V
        using full converse-rtranclpE unfolding full-def by (fastforce elim: rulesE)
      then have confl-V: conflicting V ≠ None
        using skip by (auto elim!: rulesE simp del: state-simp simp: state-eq-def)
      consider
          (s') cdcl_W-s'** S T
        | (bj) S' where cdcl_W-s'** S S' and cdcl_W-bj++ S' T and conflicting T ≠ None
        using IH by blast
      then show ?thesis
```

proof *cases*
case s'
show *?thesis* **using** s' *confl-V skip* **by** *force*
next
case (*bj* S') **note** $S\text{-}S' = this(1)$ **and** $bj\text{-}T = this(2)$
have $cdcl_W\text{-}bj^{++}\ S'\ V$
using *skip bj-T* **by** (*metis* ‹$U = V$› $cdcl_W$*-bj.skip tranclp.simps*)
then show *?thesis* **using** $S\text{-}S'$ *confl-V* **by** *auto*
qed
next
case *resolve*
then have [*simp*]: $U = V$
using *full* **unfolding** *full-def rtranclp-unfold*
by (*auto elim*!: *rulesE dest*!: *tranclpD*
simp del: *state-simp simp*: *state-eq-def* $cdcl_W$*-cp.simps*)
have *confl-V*: *conflicting* $V \neq$ *None*
using *resolve* **by** (*auto elim*!: *rulesE simp del*: *state-simp simp*: *state-eq-def*)

consider
(s') $cdcl_W\text{-}s'^{**}\ S\ T$
| (*bj*) S' **where** $cdcl_W\text{-}s'^{**}\ S\ S'$ **and** $cdcl_W\text{-}bj^{++}\ S'\ T$ **and** *conflicting* $T \neq$ *None*
using *IH* **by** *blast*
then show *?thesis*
proof *cases*
case s'
have $cdcl_W\text{-}bj^{++}\ T\ V$
using *resolve* **by** *force*
then show *?thesis* **using** s' *confl-V* **by** *auto*
next
case (*bj* S') **note** $S\text{-}S' = this(1)$ **and** $bj\text{-}T = this(2)$
have $cdcl_W\text{-}bj^{++}\ S'\ V$
using *resolve bj-T* **by** (*metis* ‹$U = V$› $cdcl_W$*-bj.resolve tranclp.simps*)
then show *?thesis* **using** *confl-V* $S\text{-}S'$ **by** *auto*
qed
qed
qed
qed

lemma *n-step-*$cdcl_W$*-stgy-iff-no-step-*$cdcl_W$*-cl-*$cdcl_W$*-o*:
assumes *inv*: $cdcl_W$*-all-struct-inv S*
shows *no-step* $cdcl_W\text{-}s'\ S \longleftrightarrow$ *no-step* $cdcl_W\text{-}cp\ S \land$ *no-step* $cdcl_W\text{-}o\ S$ (**is** $?S'\ S \longleftrightarrow\ ?C\ S \land\ ?O\ S$)
proof
assume $?C\ S \land\ ?O\ S$
then show $?S'\ S$
by (*auto simp*: $cdcl_W\text{-}s'$*.simps full1-def tranclp-unfold-begin*)
next
assume *n-s*: $?S'\ S$
have $?C\ S$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then obtain S' **where** $cdcl_W\text{-}cp\ S\ S'$
by *auto*
then obtain T **where** *full1* $cdcl_W\text{-}cp\ S\ T$
using $cdcl_W$*-cp-normalized-element-all-inv inv* **by** (*metis* (*no-types*, *lifting*) *full-unfold*)
then show *False* **using** *n-s* $cdcl_W\text{-}s'.conflict'$ **by** *blast*
qed

moreover have *?O S*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain S' **where** $cdcl_W$*-o* S S'
by *auto*
then obtain T **where** *full1* $cdcl_W$*-cp* S' T
using $cdcl_W$*-cp-normalized-element-all-inv inv*
by (*meson* $cdcl_W$*-all-struct-inv-def n-s*
$cdcl_W$*-stgy-*$cdcl_W$*-s'-connected'* $cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step*)
then show *False* **using** *n-s* **by** (*meson* ‹$cdcl_W$*-o S S'*› $cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-stgy-*$cdcl_W$*-s'-connected'* $cdcl_W$*-then-exists-*$cdcl_W$*-stgy-step inv*)
qed
ultimately show *?C S* ∧ *?O S* **by** *auto*
qed

lemma $cdcl_W$*-s'-tranclp-*$cdcl_W$:
$cdcl_W$*-s'* S S' $\Longrightarrow$ $cdcl_W{}^{++}$ S S'
proof (*induct rule*: $cdcl_W$*-s'.induct*)
case *conflict'*
then show *?case*
by (*simp add*: *full1-def tranclp-*$cdcl_W$*-cp-tranclp-*$cdcl_W$)
next
case *decide'*
then show *?case*
using $cdcl_W$*-stgy.simps* $cdcl_W$*-stgy-tranclp-*$cdcl_W$ **by** (*meson* $cdcl_W$*-o.simps*)
next
case (*bj' Sa S'a S''*) **note** *a2* = *this*(*1*) **and** *a1* = *this*(*2*) **and** *n-s* = *this*(*3*)
obtain *ss* :: *'st* ⇒ *'st* ⇒ (*'st* ⇒ *'st* ⇒ *bool*) ⇒ *'st* **where**
∀ *x0 x1 x2*. (∃ *v3*. *x2 x1 v3* ∧ $x2^{**}$ *v3 x0*) = (*x2 x1* (*ss x0 x1 x2*) ∧ $x2^{**}$ (*ss x0 x1 x2*) *x0*)
by *moura*
then have *f3*: ∀ *p s sa*. ¬ p^{++} *s sa* ∨ *p s* (*ss sa s p*) ∧ p^{**} (*ss sa s p*) *sa*
by (*metis* (*full-types*) *tranclpD*)
have $cdcl_W$*-bj*$^{++}$ *Sa S'a* ∧ *no-step* $cdcl_W$*-bj S'a*
using *a2* **by** (*simp add*: *full1-def*)
then have $cdcl_W$*-bj Sa* (*ss S'a Sa* $cdcl_W$*-bj*) ∧ $cdcl_W$*-bj*** (*ss S'a Sa* $cdcl_W$*-bj*) *S'a*
using *f3* **by** *auto*
then show $cdcl_W{}^{++}$ *Sa S''*
using *a1 n-s* **by** (*meson bj other rtranclp-*$cdcl_W$*-bj-full1-cdclp-*$cdcl_W$*-stgy*
rtranclp-$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ *rtranclp-into-tranclp2*)
qed

lemma *tranclp-*$cdcl_W$*-s'-tranclp-*$cdcl_W$:
$cdcl_W$*-s'*$^{++}$ S S' $\Longrightarrow$ $cdcl_W{}^{++}$ S S'
apply (*induct rule*: *tranclp.induct*)
using $cdcl_W$*-s'-tranclp-*$cdcl_W$ **apply** *blast*
by (*meson* $cdcl_W$*-s'-tranclp-*$cdcl_W$ *tranclp-trans*)

lemma *rtranclp-*$cdcl_W$*-s'-rtranclp-*$cdcl_W$:
$cdcl_W$*-s'*** S S' $\Longrightarrow$ $cdcl_W{}^{**}$ S S'
using *rtranclp-unfold*[*of* $cdcl_W$*-s'* S S'] *tranclp-*$cdcl_W$*-s'-tranclp-*$cdcl_W$[*of* S S'] **by** *auto*

lemma *full-*$cdcl_W$*-stgy-iff-full-*$cdcl_W$*-s'*:
assumes *inv*: $cdcl_W$*-all-struct-inv* S
shows *full* $cdcl_W$*-stgy* S T ⟷ *full* $cdcl_W$*-s'* S T (**is** *?S* ⟷ *?S'*)
proof
assume *?S'*

then have $cdcl_W{}^{**}$ S T
using *rtranclp-cdcl*$_W$*-s'-rtranclp-cdcl*$_W$[*of* S T] **unfolding** *full-def* **by** *blast*
then have *inv'*: *cdcl*$_W$*-all-struct-inv* T
using *rtranclp-cdcl*$_W$*-all-struct-inv-inv inv* **by** *blast*
have *cdcl*$_W$*-stgy*** S T
using ‹*?S'*› **unfolding** *full-def*
using *cdcl*$_W$*-s'-is-rtranclp-cdcl*$_W$*-stgy rtranclp-mono*[*of cdcl*$_W$*-s' cdcl*$_W$*-stgy***] **by** *auto*
then show *?S*
using ‹*?S'*› *inv' cdcl*$_W$*-stgy-cdcl*$_W$*-s'-connected'* **unfolding** *full-def* **by** *blast*
next
assume *?S*
then have *inv-T*:*cdcl*$_W$*-all-struct-inv* T
by (*metis assms full-def rtranclp-cdcl*$_W$*-all-struct-inv-inv rtranclp-cdcl*$_W$*-stgy-rtranclp-cdcl*$_W$)

consider
(s') *cdcl*$_W$*-s'*** S T
| (*st*) S' **where** *cdcl*$_W$*-s'*** S S' **and** *cdcl*$_W$*-bj*$^{++}$ S' T **and** *conflicting* $T \neq$ *None*
using *rtranclp-cdcl*$_W$*-stgy-connected-to-rtranclp-cdcl*$_W$*-s'*[*of* S T] *inv* ‹*?S*›
unfolding *full-def cdcl*$_W$*-all-struct-inv-def*
by *blast*
then show *?S'*
proof *cases*
case s'
have *no-step cdcl*$_W$*-s'* T
using ‹*full cdcl*$_W$*-stgy* S T› **unfolding** *full-def*
by (*meson cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-s'E cdcl*$_W$*-stgy.conflict'*
cdcl$_W$*-then-exists-cdcl*$_W$*-stgy-step inv-T n-step-cdcl*$_W$*-stgy-iff-no-step-cdcl*$_W$*-cl-cdcl*$_W$*-o*)
then show *?thesis*
using s' **unfolding** *full-def* **by** *blast*
next
case (*st* S')
have *full cdcl*$_W$*-cp* T T
using *option-full-cdcl*$_W$*-cp st*(*3*) **by** *blast*
moreover
have *n-s*: *no-step cdcl*$_W$*-bj* T
by (*metis* ‹*full cdcl*$_W$*-stgy* S T› *bj inv-T cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-then-exists-cdcl*$_W$*-stgy-step full-def*)
then have *full1 cdcl*$_W$*-bj* S' T
using *st*(*2*) **unfolding** *full1-def* **by** *blast*
moreover have *no-step cdcl*$_W$*-cp* S'
using *st*(*2*) **by** (*fastforce dest*!: *tranclpD simp*: *cdcl*$_W$*-cp.simps cdcl*$_W$*-bj.simps*
elim: *rulesE*)
ultimately have *cdcl*$_W$*-s'* S' T
using *cdcl*$_W$*-s'.bj'*[*of* S' T T] **by** *blast*
then have *cdcl*$_W$*-s'*** S T
using *st*(*1*) **by** *auto*
moreover have *no-step cdcl*$_W$*-s'* T
using *inv-T* ‹*full cdcl*$_W$*-cp* T T› ‹*full cdcl*$_W$*-stgy* S T› **unfolding** *full-def*
by (*metis cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-then-exists-cdcl*$_W$*-stgy-step*
n-step-cdcl$_W$*-stgy-iff-no-step-cdcl*$_W$*-cl-cdcl*$_W$*-o*)
ultimately show *?thesis*
unfolding *full-def* **by** *blast*
qed
qed

lemma *conflict-step-cdcl*$_W$*-stgy-step*:

assumes
conflict S T
$cdcl_W$*-all-struct-inv S*
shows $\exists$ *T. $cdcl_W$-stgy S T*
proof –
obtain *U* **where** *full $cdcl_W$-cp S U*
using *$cdcl_W$-cp-normalized-element-all-inv assms* **by** *blast*
then have *full1 $cdcl_W$-cp S U*
by (*metis $cdcl_W$-cp.conflict' assms*(*1*) *full-unfold*)
then show *?thesis* **using** *$cdcl_W$-stgy.conflict'* **by** *blast*
qed

lemma *decide-step-$cdcl_W$-stgy-step*:
assumes
decide S T
$cdcl_W$-all-struct-inv S
shows $\exists$ *T. $cdcl_W$-stgy S T*
proof –
obtain *U* **where** *full $cdcl_W$-cp T U*
using *$cdcl_W$-cp-normalized-element-all-inv* **by** (*meson assms*(*1*) *assms*(*2*) *$cdcl_W$-all-struct-inv-inv $cdcl_W$-cp-normalized-element-all-inv decide other*)
then show *?thesis*
by (*metis assms $cdcl_W$-cp-normalized-element-all-inv $cdcl_W$-stgy.conflict' decide full-unfold other'*)
qed

lemma *rtranclp-$cdcl_W$-cp-conflicting-Some*:
$cdcl_W$-cp** S T $\Longrightarrow$ conflicting S = Some D $\Longrightarrow$ S = T
using *rtranclpD tranclpD* **by** *fastforce*

inductive *$cdcl_W$-merge-cp* :: *'st $\Rightarrow$ 'st $\Rightarrow$ bool* **for** *S* :: *'st* **where**
conflict': *conflict S T $\Longrightarrow$ full $cdcl_W$-bj T U $\Longrightarrow$ $cdcl_W$-merge-cp S U* |
propagate': *propagate^{++} S S' $\Longrightarrow$ $cdcl_W$-merge-cp S S'*

lemma *$cdcl_W$-merge-restart-cases*[*consumes 1*, *case-names conflict propagate*]:
assumes
$cdcl_W$-merge-cp S U **and**
$\bigwedge$*T. conflict S T $\Longrightarrow$ full $cdcl_W$-bj T U $\Longrightarrow$ P* **and**
propagate^{++} S U $\Longrightarrow$ P
shows *P*
using *assms* **unfolding** *$cdcl_W$-merge-cp.simps* **by** *auto*

lemma *$cdcl_W$-merge-cp-tranclp-$cdcl_W$-merge*:
$cdcl_W$-merge-cp S T $\Longrightarrow$ $cdcl_W$-merge^{++} S T
apply (*induction rule*: *$cdcl_W$-merge-cp.induct*)
using *$cdcl_W$-merge.simps* **apply** *auto*[*1*]
using *tranclp-mono*[*of propagate $cdcl_W$-merge*] *fw-propagate* **by** *blast*

lemma *rtranclp-$cdcl_W$-merge-cp-rtranclp-$cdcl_W$*:
$cdcl_W$-merge-cp** S T $\Longrightarrow$ $cdcl_W$** S T
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
unfolding *$cdcl_W$-merge-cp.simps* **by** (*meson $cdcl_W$-merge-restart-$cdcl_W$ fw-r-conflict rtranclp-propagate-is-rtranclp-$cdcl_W$ rtranclp-trans tranclp-into-rtranclp*)

lemma *full1-$cdcl_W$-bj-no-step-$cdcl_W$-bj*:

full1 cdcl$_W$-bj S T $\Longrightarrow$ *no-step cdcl$_W$-cp S*
unfolding *full1-def* **by** (*metis rtranclp-unfold cdcl$_W$-cp-conflicting-not-empty option.exhaust rtranclp-cdcl$_W$-merge-restart-no-step-cdcl$_W$-bj tranclpD*)

Full Transformation

inductive *cdcl$_W$-s'-without-decide* **where**
conflict'-without-decide[*intro*]: *full1 cdcl$_W$-cp S S'* $\Longrightarrow$ *cdcl$_W$-s'-without-decide S S'* |
bj'-without-decide[*intro*]: *full1 cdcl$_W$-bj S S'* $\Longrightarrow$ *no-step cdcl$_W$-cp S* $\Longrightarrow$ *full cdcl$_W$-cp S' S''* $\Longrightarrow$ *cdcl$_W$-s'-without-decide S S''*

lemma *rtranclp-cdcl$_W$-s'-without-decide-rtranclp-cdcl$_W$*:
cdcl$_W$-s'-without-decide** S T $\Longrightarrow$ *cdcl$_W$** S T*
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
by (*meson cdcl$_W$-s'.simps cdcl$_W$-s'-tranclp-cdcl$_W$ cdcl$_W$-s'-without-decide.simps rtranclp-tranclp-tranclp tranclp-into-rtranclp*)

lemma *rtranclp-cdcl$_W$-s'-without-decide-rtranclp-cdcl$_W$-s'*:
cdcl$_W$-s'-without-decide** S T $\Longrightarrow$ *cdcl$_W$-s'** S T*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step y z*) **note** *a2* = *this*(*2*) **and** *a1* = *this*(*3*)
have *cdcl$_W$-s' y z*
using *a2* **by** (*metis* (*no-types*) *bj' cdcl$_W$-s'.conflict' cdcl$_W$-s'-without-decide.cases*)
then show *cdcl$_W$-s'** S z*
using *a1* **by** (*meson r-into-rtranclp rtranclp-trans*)
qed

lemma *rtranclp-cdcl$_W$-merge-cp-is-rtranclp-cdcl$_W$-s'-without-decide*:
assumes
cdcl$_W$-merge-cp** S V
conflicting S = *None*
shows
(*cdcl$_W$-s'-without-decide** S V*)
$\vee$ ($\exists$ *T*. *cdcl$_W$-s'-without-decide** S T* $\wedge$ *propagate^{++} T V*)
$\vee$ ($\exists$ *T U*. *cdcl$_W$-s'-without-decide** S T* $\wedge$ *full1 cdcl$_W$-bj T U* $\wedge$ *propagate** U V*)
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step U V*) **note** *st* = *this*(*1*) **and** *cp* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*)]
from *cp* **show** *?case*
proof (*cases rule*: *cdcl$_W$-merge-restart-cases*)
case *propagate*
then show *?thesis* **using** *IH* **by** (*meson rtranclp-tranclp-tranclp tranclp-into-rtranclp*)
next
case (*conflict U'*) **note** *confl* = *this*(*1*) **and** *bj* = *this*(*2*)
have *full1-U-U'*: *full1 cdcl$_W$-cp U U'*
by (*simp add*: *conflict-is-full1-cdcl$_W$-cp local.conflict*(*1*))
consider
(*s'*) *cdcl$_W$-s'-without-decide** S U*
| (*propa*) *T'* **where** *cdcl$_W$-s'-without-decide** S T'* **and** *propagate^{++} T' U*

| (*bj-prop*) T' T'' **where**
cdcl$_W$*-s'-without-decide*** S T' **and**
full1 cdcl$_W$*-bj* T' T'' **and**
propagate** T'' U
using *IH* **by** *blast*
then show *?thesis*
proof *cases*
case *s'*
have *cdcl*$_W$*-s'-without-decide U U'*
using *full1-U-U' conflict'-without-decide* **by** *blast*
then have *cdcl*$_W$*-s'-without-decide*** *S U'*
using ‹*cdcl*$_W$*-s'-without-decide*** *S U*› **by** *auto*
moreover have $U' = V \lor$ *full1 cdcl*$_W$*-bj U' V*
using *bj* **by** (*meson full-unfold*)
ultimately show *?thesis* **by** *blast*
next
case *propa* **note** *s'* = *this*(*1*) **and** *T'-U* = *this*(*2*)
have *full1 cdcl*$_W$*-cp T' U'*
using *rtranclp-mono*[*of propagate cdcl*$_W$*-cp*] *T'-U cdcl*$_W$*-cp.propagate' full1-U-U'*
rtranclp-full1I[*of cdcl*$_W$*-cp T'*] **by** (*metis* (*full-types*) *predicate2D predicate2I*
tranclp-into-rtranclp)
have *cdcl*$_W$*-s'-without-decide*** *S U'*
using ‹*full1 cdcl*$_W$*-cp T' U'*› *conflict'-without-decide s'* **by** *force*
have *full1 cdcl*$_W$*-bj U' V* $\lor$ $V = U'$ **using** *bj* **unfolding** *full-unfold* **by** *blast*
then show *?thesis*
using ‹*cdcl*$_W$*-s'-without-decide*** *S U'*› **by** *blast*
next
case *bj-prop* **note** *s'* = *this*(*1*) **and** *bj-T'* = *this*(*2*) **and** *T''-U* = *this*(*3*)
have *no-step cdcl*$_W$*-cp T'*
using *bj-T' full1-cdcl*$_W$*-bj-no-step-cdcl*$_W$*-bj* **by** *blast*
moreover have *full1 cdcl*$_W$*-cp T'' U'*
using *rtranclp-mono*[*of propagate cdcl*$_W$*-cp*] *T''-U cdcl*$_W$*-cp.propagate' full1-U-U'*
rtranclp-full1I[*of cdcl*$_W$*-cp T''*] **by** *blast*
ultimately have *cdcl*$_W$*-s'-without-decide T' U'*
using *bj'-without-decide*[*of T' T'' U'*] *bj-T'* **by** (*simp add*: *full-unfold*)
then have *cdcl*$_W$*-s'-without-decide*** *S U'*
using *s' rtranclp.intros*(*2*)[*of - S T' U'*] **by** *blast*
then show *?thesis*
using *local.bj* **unfolding** *full-unfold* **by** *blast*
qed
qed
qed

lemma *rtranclp-cdcl*$_W$*-s'-without-decide-is-rtranclp-cdcl*$_W$*-merge-cp*:
assumes
cdcl$_W$*-s'-without-decide*** *S V* **and**
confl: *conflicting S* = *None*
shows
(*cdcl*$_W$*-merge-cp*** *S V* $\land$ *conflicting V* = *None*)
$\lor$ (*cdcl*$_W$*-merge-cp*** *S V* $\land$ *conflicting V* $\neq$ *None* $\land$ *no-step cdcl*$_W$*-cp V* $\land$ *no-step cdcl*$_W$*-bj V*)
$\lor$ ($\exists$ *T*. *cdcl*$_W$*-merge-cp*** *S T* $\land$ *conflict T V*)
using *assms*(*1*)
proof (*induction*)
case *base*
then show *?case* **using** *confl* **by** *auto*
next

```
case (step U V) note st = this(1) and s = this(2) and IH = this(3)
from s show ?case
  proof (cases rule: cdcl_W-s'-without-decide.cases)
    case conflict'-without-decide
    then have rt: cdcl_W-cp⁺⁺ U V unfolding full1-def by fast
    then have conflicting U = None
      using tranclp-cdcl_W-cp-propagate-with-conflict-or-not[of U V]
      conflict by (auto dest!: tranclpD simp: rtranclp-unfold elim: rulesE)
    then have cdcl_W-merge-cp** S U using IH by (auto elim: rulesE
      simp del: state-simp simp: state-eq-def)
    consider
        (propa) propagate⁺⁺ U V
      | (confl') conflict U V
      | (propa-confl') U' where propagate⁺⁺ U U' conflict U' V
      using tranclp-cdcl_W-cp-propagate-with-conflict-or-not[OF rt] unfolding rtranclp-unfold
      by fastforce
    then show ?thesis
      proof cases
        case propa
        then have cdcl_W-merge-cp U V
          by (auto intro: cdcl_W-merge-cp.intros)
        moreover have conflicting V = None
          using propa unfolding tranclp-unfold-end by (auto elim: rulesE)
        ultimately show ?thesis using ‹cdcl_W-merge-cp** S U› by (auto elim!: rulesE
          simp del: state-simp simp: state-eq-def)
      next
        case confl'
        then show ?thesis using ‹cdcl_W-merge-cp** S U› by auto
      next
        case propa-confl' note propa = this(1) and confl' = this(2)
        then have cdcl_W-merge-cp U U' by (auto intro: cdcl_W-merge-cp.intros)
        then have cdcl_W-merge-cp** S U' using ‹cdcl_W-merge-cp** S U› by auto
        then show ?thesis using ‹cdcl_W-merge-cp** S U› confl' by auto
      qed
  next
    case (bj'-without-decide U') note full-bj = this(1) and cp = this(3)
    then have conflicting U ≠ None
      using full-bj unfolding full1-def by (fastforce dest!: tranclpD simp: cdcl_W-bj.simps
        elim: rulesE)
    with IH obtain T where
      S-T: cdcl_W-merge-cp** S T and T-U: conflict T U
      using full-bj unfolding full1-def by (blast dest: tranclpD)
    then have cdcl_W-merge-cp T U'
      using cdcl_W-merge-cp.conflict'[of T U U'] full-bj by (simp add: full-unfold)
    then have S-U': cdcl_W-merge-cp** S U' using S-T by auto
    consider
        (n-s) U' = V
      | (propa) propagate⁺⁺ U' V
      | (confl') conflict U' V
      | (propa-confl') U'' where propagate⁺⁺ U' U'' conflict U'' V
      using tranclp-cdcl_W-cp-propagate-with-conflict-or-not cp
      unfolding rtranclp-unfold full-def by metis
    then show ?thesis
      proof cases
        case propa
        then have cdcl_W-merge-cp U' V by (blast intro: cdcl_W-merge-cp.intros)
```

moreover have *conflicting V = None*
using *propa* **unfolding** *tranclp-unfold-end* **by** (*auto elim*: *rulesE*)
ultimately show *?thesis* **using** *S-U'* **by** (*auto elim*: *rulesE*
simp del: *state-simp simp*: *state-eq-def*)
next
case *confl'*
then show *?thesis* **using** *S-U'* **by** *auto*
next
case *propa-confl'* **note** *propa = this(1)* **and** *confl = this(2)*
have $cdcl_W$*-merge-cp U' U''* **using** *propa* **by** (*blast intro*: $cdcl_W$*-merge-cp.intros*)
then show *?thesis* **using** *S-U' confl* **by** (*meson rtranclp.rtrancl-into-rtrancl*)
next
case *n-s*
then show *?thesis*
using *S-U'* **apply** (*cases conflicting V = None*)
using *full-bj* **apply** *simp*
by (*metis cp full-def full-unfold full-bj*)
qed
qed
qed

lemma *no-step-*$cdcl_W$*-s'-no-ste-*$cdcl_W$*-merge-cp*:
assumes
$cdcl_W$*-all-struct-inv S*
conflicting S = None
no-step $cdcl_W$*-s' S*
shows *no-step* $cdcl_W$*-merge-cp S*
using *assms* **apply** (*auto simp*: $cdcl_W$*-s'.simps* $cdcl_W$*-merge-cp.simps*)
using *conflict-is-full1-*$cdcl_W$*-cp* **apply** *blast*
using $cdcl_W$*-cp-normalized-element-all-inv* $cdcl_W$*-cp.propagate'* **by** (*metis* $cdcl_W$*-cp.propagate'*
full-unfold tranclpD)

The *no-step decide S* is needed, since $cdcl_W$*-merge-cp* is $cdcl_W$*-s'* without *decide*.

lemma *conflicting-true-no-step-*$cdcl_W$*-merge-cp-no-step-s'-without-decide*:
assumes
confl: *conflicting S = None* **and**
inv: $cdcl_W$*-M-level-inv S* **and**
n-s: *no-step* $cdcl_W$*-merge-cp S*
shows *no-step* $cdcl_W$*-s'-without-decide S*
proof (*rule ccontr*)
assume ¬ *no-step* $cdcl_W$*-s'-without-decide S*
then obtain *T* **where**
$cdcl_W$: $cdcl_W$*-s'-without-decide S T*
by *auto*
then have *inv-T*: $cdcl_W$*-M-level-inv T*
using *rtranclp-*$cdcl_W$*-s'-without-decide-rtranclp-*$cdcl_W$*[of S T]*
rtranclp-$cdcl_W$*-consistent-inv inv* **by** *blast*
from $cdcl_W$ **show** *False*
proof *cases*
case *conflict'-without-decide*
have *no-step propagate S*
using *n-s* **by** (*blast intro*: $cdcl_W$*-merge-cp.intros*)
then have *conflict S T*
using *local.conflict' tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not[of S T]*
local.conflict'-without-decide **unfolding** *full1-def rtranclp-unfold*
by (*metis tranclp-unfold-begin*)

moreover
then obtain T' **where** *full* $cdcl_W$*-bj* T T'
using $cdcl_W$*-bj-exists-normal-form inv-T* **by** *blast*
ultimately show *False* **using** $cdcl_W$*-merge-cp.conflict'* *n-s* **by** *meson*
next
case (*bj'-without-decide* S')
then show *?thesis*
using *confl* **unfolding** *full1-def* **by** (*fastforce simp*: $cdcl_W$*-bj.simps dest*: *tranclpD elim*: *rulesE*)
qed
qed

lemma *conflicting-true-no-step-s'-without-decide-no-step-*$cdcl_W$*-merge-cp*:
assumes
inv: $cdcl_W$*-all-struct-inv* S **and**
n-s: *no-step* $cdcl_W$*-s'-without-decide* S
shows *no-step* $cdcl_W$*-merge-cp* S
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain T **where** $cdcl_W$*-merge-cp* S T
by *auto*
then show *False*
proof *cases*
case (*conflict'* S')
then show *False* **using** *n-s conflict'-without-decide conflict-is-full1-*$cdcl_W$*-cp* **by** *blast*
next
case *propagate'*
moreover
have $cdcl_W$*-all-struct-inv* T
using *inv* **by** (*meson local.propagate' rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-propagate-is-rtranclp-*$cdcl_W$ *tranclp-into-rtranclp*)
then obtain U **where** *full* $cdcl_W$*-cp* T U
using $cdcl_W$*-cp-normalized-element-all-inv* **by** *auto*
ultimately have *full1* $cdcl_W$*-cp* S U
using *tranclp-full-full1I*[*of* $cdcl_W$*-cp* S T U] $cdcl_W$*-cp.propagate'*
tranclp-mono[*of propagate* $cdcl_W$*-cp*] **by** *blast*
then show *False* **using** *conflict'-without-decide n-s* **by** *blast*
qed
qed

lemma *no-step-*$cdcl_W$*-merge-cp-no-step-*$cdcl_W$*-cp*:
no-step $cdcl_W$*-merge-cp* $S \Longrightarrow$ $cdcl_W$*-M-level-inv* $S \Longrightarrow$ *no-step* $cdcl_W$*-cp* S
using $cdcl_W$*-bj-exists-normal-form* $cdcl_W$*-consistent-inv*[*OF* $cdcl_W$*.conflict*, *of* S]
by (*metis* $cdcl_W$*-cp.cases* $cdcl_W$*-merge-cp.simps tranclp.intros*(*1*))

lemma *conflicting-not-true-rtranclp-*$cdcl_W$*-merge-cp-no-step-*$cdcl_W$*-bj*:
assumes
conflicting S = *None* **and**
$cdcl_W$*-merge-cp*** S T
shows *no-step* $cdcl_W$*-bj* T
using *assms*(*2*,*1*) **by** (*induction*)
(*fastforce simp*: $cdcl_W$*-merge-cp.simps full-def tranclp-unfold-end* $cdcl_W$*-bj.simps elim*: *rulesE*)+

lemma *conflicting-true-full-*$cdcl_W$*-merge-cp-iff-full-*$cdcl_W$*-s'-without-decode*:
assumes

confl: *conflicting S = None* **and**
inv: *cdcl$_W$-all-struct-inv S*
shows
full cdcl$_W$-merge-cp S V $\longleftrightarrow$ *full cdcl$_W$-s'-without-decide S V* (**is** *?fw* $\longleftrightarrow$ *?s'*)
proof
assume *?fw*
then have *st*: *cdcl$_W$-merge-cp** S V* **and** *n-s*: *no-step cdcl$_W$-merge-cp V*
unfolding *full-def* **by** *blast+*
have *inv-V*: *cdcl$_W$-all-struct-inv V*
using *rtranclp-cdcl$_W$-merge-cp-rtranclp-cdcl$_W$[of S V]* ‹*?fw*› **unfolding** *full-def*
by (*simp add*: *inv rtranclp-cdcl$_W$-all-struct-inv-inv*)
consider
(*s'*) *cdcl$_W$-s'-without-decide** S V*
| (*propa*) *T* **where** *cdcl$_W$-s'-without-decide** S T* **and** *propagate^{++} T V*
| (*bj*) *T U* **where** *cdcl$_W$-s'-without-decide** S T* **and** *full1 cdcl$_W$-bj T U* **and** *propagate** U V*
using *rtranclp-cdcl$_W$-merge-cp-is-rtranclp-cdcl$_W$-s'-without-decide confl st n-s* **by** *metis*
then have *cdcl$_W$-s'-without-decide** S V*
proof *cases*
case *s'*
then show *?thesis* **.**
next
case *propa* **note** *s' = this(1)* **and** *propa = this(2)*
have *no-step cdcl$_W$-cp V*
using *no-step-cdcl$_W$-merge-cp-no-step-cdcl$_W$-cp n-s inv-V*
unfolding *cdcl$_W$-all-struct-inv-def* **by** *blast*
then have *full1 cdcl$_W$-cp T V*
using *propa tranclp-mono[of propagate cdcl$_W$-cp] cdcl$_W$-cp.propagate'* **unfolding** *full1-def*
by *blast*
then have *cdcl$_W$-s'-without-decide T V*
using *conflict'-without-decide* **by** *blast*
then show *?thesis* **using** *s'* **by** *auto*
next
case *bj* **note** *s' = this(1)* **and** *bj = this(2)* **and** *propa = this(3)*
have *no-step cdcl$_W$-cp V*
using *no-step-cdcl$_W$-merge-cp-no-step-cdcl$_W$-cp n-s inv-V*
unfolding *cdcl$_W$-all-struct-inv-def* **by** *blast*
then have *full cdcl$_W$-cp U V*
using *propa rtranclp-mono[of propagate cdcl$_W$-cp] cdcl$_W$-cp.propagate'* **unfolding** *full-def*
by *blast*
moreover have *no-step cdcl$_W$-cp T*
using *bj* **unfolding** *full1-def* **by** (*fastforce dest!*: *tranclpD simp*:*cdcl$_W$-bj.simps elim*: *rulesE*)
ultimately have *cdcl$_W$-s'-without-decide T V*
using *bj'-without-decide[of T U V] bj* **by** *blast*
then show *?thesis* **using** *s'* **by** *auto*
qed
moreover have *no-step cdcl$_W$-s'-without-decide V*
proof (*cases conflicting V = None*)
case *False*
{ fix *ss* :: *'st*
have *ff1*: $\forall$ *s sa*. $\neg$ *cdcl$_W$-s' s sa* $\vee$ *full1 cdcl$_W$-cp s sa*
$\vee$ ($\exists$ *sb*. *decide s sb* $\wedge$ *no-step cdcl$_W$-cp s* $\wedge$ *full cdcl$_W$-cp sb sa*)
$\vee$ ($\exists$ *sb*. *full1 cdcl$_W$-bj s sb* $\wedge$ *no-step cdcl$_W$-cp s* $\wedge$ *full cdcl$_W$-cp sb sa*)
by (*metis cdcl$_W$-s'.cases*)
have *ff2*: ($\forall$ *p s sa*. $\neg$ *full1 p* (*s*::*'st*) *sa* $\vee$ *p^{++} s sa* $\wedge$ *no-step p sa*)
$\wedge$ ($\forall$ *p s sa*. ($\neg$ *p^{++}* (*s*::*'st*) *sa* $\vee$ ($\exists$ *s*. *p sa s*)) $\vee$ *full1 p s sa*)
by (*meson full1-def*)

obtain *ssa* :: ($'st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st \Rightarrow 'st$ **where**
ff3: $\forall\, p\ s\ sa.\ \neg\ p^{++}\ s\ sa \vee p\ s\ (ssa\ p\ s\ sa) \wedge p^{**}\ (ssa\ p\ s\ sa)\ sa$
by (*metis* (*no-types*) *tranclpD*)
then have *a3*: $\neg\ cdcl_W\text{-}cp^{++}\ V\ ss$
using *False* **by** (*metis option-full-cdcl*$_W$*-cp full-def*)
have $\bigwedge s.\ \neg\ cdcl_W\text{-}bj^{++}\ V\ s$
using *ff3 False* **by** (*metis confl st*
conflicting-not-true-rtranclp-cdcl$_W$*-merge-cp-no-step-cdcl*$_W$*-bj*)
then have $\neg\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ V\ ss$
using *ff1 a3 ff2* **by** (*metis cdcl*$_W$*-s'-without-decide.cases*)
}
then show *?thesis*
by *fastforce*
next
case *True*
then show *?thesis*
using *conflicting-true-no-step-cdcl*$_W$*-merge-cp-no-step-s'-without-decide n-s inv-V*
unfolding *cdcl*$_W$*-all-struct-inv-def* **by** *simp*
qed
ultimately show *?s'* **unfolding** *full-def* **by** *blast*
next
assume s': *?s'*
then have *st*: $cdcl_W\text{-}s'\text{-}without\text{-}decide^{**}\ S\ V$ **and** *n-s*: *no-step cdcl*$_W$*-s'-without-decide V*
unfolding *full-def* **by** *auto*
then have $cdcl_W^{**}\ S\ V$
using *rtranclp-cdcl*$_W$*-s'-without-decide-rtranclp-cdcl*$_W$ *st* **by** *blast*
then have *inv-V*: *cdcl*$_W$*-all-struct-inv V* **using** *inv rtranclp-cdcl*$_W$*-all-struct-inv-inv* **by** *blast*
then have *n-s-cp-V*: *no-step cdcl*$_W$*-cp V*
using *cdcl*$_W$*-cp-normalized-element-all-inv*[*of V*] *full-fullI*[*of cdcl*$_W$*-cp V*] *n-s*
conflict'-without-decide conflicting-true-no-step-s'-without-decide-no-step-cdcl$_W$*-merge-cp*
no-step-cdcl$_W$*-merge-cp-no-step-cdcl*$_W$*-cp*
unfolding *cdcl*$_W$*-all-struct-inv-def* **by** *presburger*
have *n-s-bj*: *no-step cdcl*$_W$*-bj V*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then obtain *W* **where** *W*: *cdcl*$_W$*-bj V W* **by** *blast*
have *cdcl*$_W$*-all-struct-inv W*
using *W cdcl*$_W$*.simps cdcl*$_W$*-all-struct-inv-inv inv-V* **by** *blast*
then obtain W' **where** *full1 cdcl*$_W$*-bj V W'*
using *cdcl*$_W$*-bj-exists-normal-form*[*of W*] *full-fullI*[*of cdcl*$_W$*-bj V W*] *W*
unfolding *cdcl*$_W$*-all-struct-inv-def*
by *blast*
moreover
then have $cdcl_W^{++}\ V\ W'$
using *tranclp-mono*[*of cdcl*$_W$*-bj cdcl*$_W$] *cdcl*$_W$*.other cdcl*$_W$*-o.bj* **unfolding** *full1-def* **by** *blast*
then have *cdcl*$_W$*-all-struct-inv W'*
by (*meson inv-V rtranclp-cdcl*$_W$*-all-struct-inv-inv tranclp-into-rtranclp*)
then obtain *X* **where** *full cdcl*$_W$*-cp W' X*
using *cdcl*$_W$*-cp-normalized-element-all-inv* **by** *blast*
ultimately show *False*
using *bj'-without-decide n-s-cp-V n-s* **by** *blast*
qed
from s' **consider**
(*cp-true*) $cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V$ **and** *conflicting* $V = None$
| (*cp-false*) $cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V$ **and** *conflicting* $V \neq None$ **and** *no-step cdcl*$_W$*-cp V* **and**
no-step cdcl$_W$*-bj V*

| (*cp-confl*) T **where** $cdcl_W$*-merge-cp*** S T *conflict* T V
using *rtranclp-*$cdcl_W$*-s'-without-decide-is-rtranclp-*$cdcl_W$*-merge-cp*[*of* S V] *confl*
unfolding *full-def* **by** *meson*
then have $cdcl_W$*-merge-cp*** S V
proof *cases*
case *cp-confl* **note** S-T = *this*(1) **and** *conf*-V = *this*(2)
have *full* $cdcl_W$*-bj* V V
using *conf*-V *n-s-bj* **unfolding** *full-def* **by** *fast*
then have $cdcl_W$*-merge-cp* T V
using $cdcl_W$*-merge-cp.conflict'* *conf*-V **by** *auto*
then show *?thesis* **using** S-T **by** *auto*
qed *fast+*
moreover
then have $cdcl_W{}^{**}$ S V **using** *rtranclp-*$cdcl_W$*-merge-cp-rtranclp-*$cdcl_W$ **by** *blast*
then have $cdcl_W$*-all-struct-inv* V
using *inv* *rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
then have *no-step* $cdcl_W$*-merge-cp* V
using *conflicting-true-no-step-s'-without-decide-no-step-*$cdcl_W$*-merge-cp* *s'*
unfolding *full-def* **by** *blast*
ultimately show *?fw* **unfolding** *full-def* **by** *auto*
qed

lemma *conflicting-true-full1-*$cdcl_W$*-merge-cp-iff-full1-*$cdcl_W$*-s'-without-decode*:
assumes
confl: *conflicting* S = *None* **and**
inv: $cdcl_W$*-all-struct-inv* S
shows
full1 $cdcl_W$*-merge-cp* S V $\longleftrightarrow$ *full1* $cdcl_W$*-s'-without-decide* S V
proof −
have *full* $cdcl_W$*-merge-cp* S V = *full* $cdcl_W$*-s'-without-decide* S V
using *confl* *conflicting-true-full-*$cdcl_W$*-merge-cp-iff-full-*$cdcl_W$*-s'-without-decode* *inv*
by *simp*
then show *?thesis* **unfolding** *full-unfold* *full1-def* *tranclp-unfold-begin* **by** *blast*
qed

lemma *conflicting-true-full1-*$cdcl_W$*-merge-cp-imp-full1-*$cdcl_W$*-s'-without-decode*:
assumes
fw: *full1* $cdcl_W$*-merge-cp* S V **and**
inv: $cdcl_W$*-all-struct-inv* S
shows
full1 $cdcl_W$*-s'-without-decide* S V
proof −
have *conflicting* S = *None*
using *fw* **unfolding** *full1-def* **by** (*auto* *dest*!: *tranclpD* *simp*: $cdcl_W$*-merge-cp.simps* *elim*: *rulesE*)
then show *?thesis*
using *conflicting-true-full1-*$cdcl_W$*-merge-cp-iff-full1-*$cdcl_W$*-s'-without-decode* *fw* *inv* **by** *simp*
qed

inductive $cdcl_W$*-merge-stgy* **for** S :: *'st* **where**
fw-s-cp[*intro*]: *full1* $cdcl_W$*-merge-cp* S T $\Longrightarrow$ $cdcl_W$*-merge-stgy* S T |
fw-s-decide[*intro*]: *decide* S T $\Longrightarrow$ *no-step* $cdcl_W$*-merge-cp* S $\Longrightarrow$ *full* $cdcl_W$*-merge-cp* T U
$\Longrightarrow$ $cdcl_W$*-merge-stgy* S U

lemma $cdcl_W$*-merge-stgy-tranclp-*$cdcl_W$*-merge*:
assumes *fw*: $cdcl_W$*-merge-stgy* S T
shows $cdcl_W$*-merge*$^{++}$ S T

proof −
  { **fix** *S T*
    assume *full1* $cdcl_W$*-merge-cp S T*
    then have $cdcl_W$*-merge*$^{++}$ *S T*
      using *tranclp-mono*[*of* $cdcl_W$*-merge-cp* $cdcl_W$*-merge*$^{++}$] $cdcl_W$*-merge-cp-tranclp-*$cdcl_W$*-merge*
      unfolding *full1-def*
      by *auto*
  } **note** *full1-*$cdcl_W$*-merge-cp-*$cdcl_W$*-merge* = *this*
  show *?thesis*
    using *fw*
    apply (*induction rule*: $cdcl_W$*-merge-stgy.induct*)
      using *full1-*$cdcl_W$*-merge-cp-*$cdcl_W$*-merge* **apply** *simp*
    unfolding *full-unfold* **by** (*auto dest*!: *full1-*$cdcl_W$*-merge-cp-*$cdcl_W$*-merge fw-decide*)
qed

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$*-merge*:
  assumes *fw*: $cdcl_W$*-merge-stgy*** *S T*
  shows $cdcl_W$*-merge*** *S T*
  using *fw* $cdcl_W$*-merge-stgy-tranclp-*$cdcl_W$*-merge rtranclp-mono*[*of* $cdcl_W$*-merge-stgy* $cdcl_W$*-merge*$^{++}$]
  unfolding *tranclp-rtranclp-rtranclp* **by** *blast*

lemma $cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$:
  $cdcl_W$*-merge-stgy S T* $\Longrightarrow$ $cdcl_W{}^{**}$ *S T*
  apply (*induction rule*: $cdcl_W$*-merge-stgy.induct*)
    using *rtranclp-*$cdcl_W$*-merge-cp-rtranclp-*$cdcl_W$ **unfolding** *full1-def*
    apply (*simp add*: *tranclp-into-rtranclp*)
  using *rtranclp-*$cdcl_W$*-merge-cp-rtranclp-*$cdcl_W$ $cdcl_W$*-o.decide* $cdcl_W$*.other* **unfolding** *full-def*
  by (*meson r-into-rtranclp rtranclp-trans*)

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$:
  $cdcl_W$*-merge-stgy*** *S T* $\Longrightarrow$ $cdcl_W{}^{**}$ *S T*
  using *rtranclp-mono*[*of* $cdcl_W$*-merge-stgy* $cdcl_W{}^{**}$] $cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$ **by** *auto*

lemma $cdcl_W$*-merge-stgy-cases*[*consumes 1*, *case-names fw-s-cp fw-s-decide*]:
  assumes
    $cdcl_W$*-merge-stgy S U*
    full1 $cdcl_W$*-merge-cp S U* $\Longrightarrow$ *P*
    $\bigwedge T$. *decide S T* $\Longrightarrow$ *no-step* $cdcl_W$*-merge-cp S* $\Longrightarrow$ *full* $cdcl_W$*-merge-cp T U* $\Longrightarrow$ *P*
  shows *P*
  using *assms* **by** (*auto simp*: $cdcl_W$*-merge-stgy.simps*)

inductive $cdcl_W$*-s'-w* :: *'st* ⇒ *'st* ⇒ *bool* **where**
conflict': *full1* $cdcl_W$*-s'-without-decide S S'* $\Longrightarrow$ $cdcl_W$*-s'-w S S'* |
decide': *decide S S'* $\Longrightarrow$ *no-step* $cdcl_W$*-s'-without-decide S* $\Longrightarrow$ *full* $cdcl_W$*-s'-without-decide S' S''*
  $\Longrightarrow$ $cdcl_W$*-s'-w S S''*

lemma $cdcl_W$*-s'-w-rtranclp-*$cdcl_W$:
  $cdcl_W$*-s'-w S T* $\Longrightarrow$ $cdcl_W{}^{**}$ *S T*
  apply (*induction rule*: $cdcl_W$*-s'-w.induct*)
    using *rtranclp-*$cdcl_W$*-s'-without-decide-rtranclp-*$cdcl_W$ **unfolding** *full1-def*
    apply (*simp add*: *tranclp-into-rtranclp*)
  using *rtranclp-*$cdcl_W$*-s'-without-decide-rtranclp-*$cdcl_W$ **unfolding** *full-def*
  by (*meson decide other rtranclp-into-tranclp2 tranclp-into-rtranclp*)

lemma *rtranclp-*$cdcl_W$*-s'-w-rtranclp-*$cdcl_W$:
  $cdcl_W$*-s'-w*** *S T* $\Longrightarrow$ $cdcl_W{}^{**}$ *S T*

using *rtranclp-mono*[*of* $cdcl_W$*-s'-w* $cdcl_W^{**}$] $cdcl_W$*-s'-w-rtranclp-*$cdcl_W$ **by** *auto*

lemma *no-step-*$cdcl_W$*-cp-no-step-*$cdcl_W$*-s'-without-decide*:
assumes *no-step* $cdcl_W$*-cp S* **and** *conflicting S = None* **and** *inv*: $cdcl_W$*-M-level-inv S*
shows *no-step* $cdcl_W$*-s'-without-decide S*
by (*metis assms* $cdcl_W$*-cp.conflict'* $cdcl_W$*-cp.propagate'* $cdcl_W$*-merge-restart-cases tranclpD*
conflicting-true-no-step-$cdcl_W$*-merge-cp-no-step-s'-without-decide*)

lemma *no-step-*$cdcl_W$*-cp-no-step-*$cdcl_W$*-merge-restart*:
assumes *no-step* $cdcl_W$*-cp S* **and** *conflicting S = None*
shows *no-step* $cdcl_W$*-merge-cp S*
by (*metis assms*(*1*) $cdcl_W$*-cp.conflict'* $cdcl_W$*-cp.propagate'* $cdcl_W$*-merge-restart-cases tranclpD*)
lemma *after-*$cdcl_W$*-s'-without-decide-no-step-*$cdcl_W$*-cp*:
assumes $cdcl_W$*-s'-without-decide S T*
shows *no-step* $cdcl_W$*-cp T*
using *assms* **by** (*induction rule*: $cdcl_W$*-s'-without-decide.induct*) (*auto simp*: *full1-def full-def*)

lemma *no-step-*$cdcl_W$*-s'-without-decide-no-step-*$cdcl_W$*-cp*:
$cdcl_W$*-all-struct-inv S* $\Longrightarrow$ *no-step* $cdcl_W$*-s'-without-decide S* $\Longrightarrow$ *no-step* $cdcl_W$*-cp S*
by (*simp add*: *conflicting-true-no-step-s'-without-decide-no-step-*$cdcl_W$*-merge-cp*
no-step-$cdcl_W$*-merge-cp-no-step-*$cdcl_W$*-cp* $cdcl_W$*-all-struct-inv-def*)

lemma *after-*$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-cp*:
assumes $cdcl_W$*-s'-w S T* **and** $cdcl_W$*-all-struct-inv S*
shows *no-step* $cdcl_W$*-cp T*
using *assms*
proof (*induction rule*: $cdcl_W$*-s'-w.induct*)
case *conflict'*
then show *?case*
by (*auto simp*: *full1-def tranclp-unfold-end after-*$cdcl_W$*-s'-without-decide-no-step-*$cdcl_W$*-cp*)
next
case (*decide' S T U*)
moreover
then have $cdcl_W^{**}$ *S U*
using *rtranclp-*$cdcl_W$*-s'-without-decide-rtranclp-*$cdcl_W$[*of T U*] $cdcl_W$*.other*[*of S T*]
$cdcl_W$*-o.decide* **unfolding** *full-def* **by** *auto*
then have $cdcl_W$*-all-struct-inv U*
using *decide'.prems rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
ultimately show *?case*
using *no-step-*$cdcl_W$*-s'-without-decide-no-step-*$cdcl_W$*-cp* **unfolding** *full-def* **by** *blast*
qed

lemma *rtranclp-*$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-cp-or-eq*:
assumes $cdcl_W$*-s'-w*** *S T* **and** $cdcl_W$*-all-struct-inv S*
shows *S = T* $\vee$ *no-step* $cdcl_W$*-cp T*
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step T U*)
moreover have $cdcl_W$*-all-struct-inv T*
using *rtranclp-*$cdcl_W$*-s'-w-rtranclp-*$cdcl_W$[*of S U*] *assms*(*2*) *rtranclp-*$cdcl_W$*-all-struct-inv-inv*
rtranclp-$cdcl_W$*-s'-w-rtranclp-*$cdcl_W$ *step.hyps*(*1*) **by** *blast*
ultimately show *?case* **using** *after-*$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-cp* **by** *fast*
qed

lemma *rtranclp-cdcl*$_W$*-merge-stgy'-no-step-cdcl*$_W$*-cp-or-eq*:
  assumes *cdcl*$_W$*-merge-stgy*** *S T* **and** *inv*: *cdcl*$_W$*-all-struct-inv S*
  shows $S = T \lor$ *no-step cdcl*$_W$*-cp T*
  using *assms*
proof (*induction rule*: *rtranclp-induct*)
  case *base*
  then show *?case* **by** *simp*
next
  case (*step T U*)
  moreover have *cdcl*$_W$*-all-struct-inv T*
    using *rtranclp-cdcl*$_W$*-merge-stgy-rtranclp-cdcl*$_W$[*of S U*] *assms*(*2*) *rtranclp-cdcl*$_W$*-all-struct-inv-inv*
    rtranclp-cdcl$_W$*-s'-w-rtranclp-cdcl*$_W$ *step.hyps*(*1*)
    by (*meson rtranclp-cdcl*$_W$*-merge-stgy-rtranclp-cdcl*$_W$)
  ultimately show *?case*
    using *after-cdcl*$_W$*-s'-w-no-step-cdcl*$_W$*-cp inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def*
    by (*metis cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-merge-stgy.simps full1-def full-def*
      no-step-cdcl$_W$*-merge-cp-no-step-cdcl*$_W$*-cp rtranclp-cdcl*$_W$*-all-struct-inv-inv*
      rtranclp-cdcl$_W$*-merge-stgy-rtranclp-cdcl*$_W$ *tranclp.intros*(*1*) *tranclp-into-rtranclp*)
qed

lemma *no-step-cdcl*$_W$*-s'-without-decide-no-step-cdcl*$_W$*-bj*:
  assumes *no-step cdcl*$_W$*-s'-without-decide S* **and** *inv*: *cdcl*$_W$*-all-struct-inv S*
  shows *no-step cdcl*$_W$*-bj S*
proof (*rule ccontr*)
  assume ¬ *?thesis*
  then obtain *T* **where** *S-T*: *cdcl*$_W$*-bj S T*
    by *auto*
  have *cdcl*$_W$*-all-struct-inv T*
    using *S-T cdcl*$_W$*-all-struct-inv-inv inv other* **by** *blast*
  then obtain *T'* **where** *full1 cdcl*$_W$*-bj S T'*
    using *cdcl*$_W$*-bj-exists-normal-form*[*of T*] *full-fullI S-T* **unfolding** *cdcl*$_W$*-all-struct-inv-def*
    by *metis*
  moreover
    then have *cdcl*$_W$** *S T'*
      using *rtranclp-mono*[*of cdcl*$_W$*-bj cdcl*$_W$] *cdcl*$_W$*.other cdcl*$_W$*-o.bj tranclp-into-rtranclp*[*of cdcl*$_W$*-bj*]
      unfolding *full1-def* **by** *blast*
    then have *cdcl*$_W$*-all-struct-inv T'*
      using *inv rtranclp-cdcl*$_W$*-all-struct-inv-inv* **by** *blast*
    then obtain *U* **where** *full cdcl*$_W$*-cp T' U*
      using *cdcl*$_W$*-cp-normalized-element-all-inv* **by** *blast*
  moreover have *no-step cdcl*$_W$*-cp S*
    using *S-T* **by** (*auto simp*: *cdcl*$_W$*-bj.simps elim*: *rulesE*)
  ultimately show *False*
  using *assms cdcl*$_W$*-s'-without-decide.intros*(*2*)[*of S T' U*] **by** *fast*
qed

lemma *cdcl*$_W$*-s'-w-no-step-cdcl*$_W$*-bj*:
  assumes *cdcl*$_W$*-s'-w S T* **and** *cdcl*$_W$*-all-struct-inv S*
  shows *no-step cdcl*$_W$*-bj T*
  using *assms* **apply** *induction*
    using *rtranclp-cdcl*$_W$*-s'-without-decide-rtranclp-cdcl*$_W$ *rtranclp-cdcl*$_W$*-all-struct-inv-inv*
    no-step-cdcl$_W$*-s'-without-decide-no-step-cdcl*$_W$*-bj* **unfolding** *full1-def*
    apply (*meson tranclp-into-rtranclp*)
  using *rtranclp-cdcl*$_W$*-s'-without-decide-rtranclp-cdcl*$_W$ *rtranclp-cdcl*$_W$*-all-struct-inv-inv*
    no-step-cdcl$_W$*-s'-without-decide-no-step-cdcl*$_W$*-bj* **unfolding** *full-def*

by (*meson* $cdcl_W$*-merge-restart-*$cdcl_W$ *fw-r-decide*)

lemma *rtranclp-*$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-bj-or-eq*:
assumes $cdcl_W$*-s'-w*** *S T* **and** $cdcl_W$*-all-struct-inv S*
shows $S = T \lor$ *no-step* $cdcl_W$*-bj T*
using *assms* **apply** *induction*
apply *simp*
using *rtranclp-*$cdcl_W$*-s'-w-rtranclp-*$cdcl_W$ *rtranclp-*$cdcl_W$*-all-struct-inv-inv*
$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-bj* **by** *meson*

lemma *rtranclp-*$cdcl_W$*-s'-no-step-*$cdcl_W$*-s'-without-decide-decomp-into-*$cdcl_W$*-merge*:
assumes
$cdcl_W$*-s'*** *R V* **and**
conflicting $R = None$ **and**
inv: $cdcl_W$*-all-struct-inv R*
shows ($cdcl_W$*-merge-stgy*** *R V* $\land$ *conflicting* $V = None$)
$\lor$ ($cdcl_W$*-merge-stgy*** *R V* $\land$ *conflicting* $V \neq None$ $\land$ *no-step* $cdcl_W$*-bj V*)
$\lor$ ($\exists$ *S T U*. $cdcl_W$*-merge-stgy*** *R S* $\land$ *no-step* $cdcl_W$*-merge-cp S* $\land$ *decide S T*
$\land$ $cdcl_W$*-merge-cp*** *T U* $\land$ *conflict U V*)
$\lor$ ($\exists$ *S T*. $cdcl_W$*-merge-stgy*** *R S* $\land$ *no-step* $cdcl_W$*-merge-cp S* $\land$ *decide S T*
$\land$ $cdcl_W$*-merge-cp*** *T V*
$\land$ *conflicting* $V = None$)
$\lor$ ($cdcl_W$*-merge-cp*** *R V* $\land$ *conflicting* $V = None$)
$\lor$ ($\exists$ *U*. $cdcl_W$*-merge-cp*** *R U* $\land$ *conflict U V*)
using *assms*(*1*,*2*)
proof *induction*
case *base*
then show *?case* **by** *simp*
next
case (*step V W*) **note** *st* = *this*(*1*) **and** *s'* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*)] **and**
n-s-R = *this*(*4*)
from *s'*
show *?case*
proof *cases*
case *conflict'*
consider
(*s'*) $cdcl_W$*-merge-stgy*** *R V*
| (*dec-confl*) *S T U* **where** $cdcl_W$*-merge-stgy*** *R S* **and** *no-step* $cdcl_W$*-merge-cp S* **and**
decide S T **and** $cdcl_W$*-merge-cp*** *T U* **and** *conflict U V*
| (*dec*) *S T* **where** $cdcl_W$*-merge-stgy*** *R S* **and** *no-step* $cdcl_W$*-merge-cp S* **and** *decide S T*
and $cdcl_W$*-merge-cp*** *T V* **and** *conflicting* $V = None$
| (*cp*) $cdcl_W$*-merge-cp*** *R V*
| (*cp-confl*) *U* **where** $cdcl_W$*-merge-cp*** *R U* **and** *conflict U V*
using *IH* **by** *meson*
then show *?thesis*
proof *cases*
case *s'*
then have $R = V$ **using** *inv local.conflict'* **unfolding** *full1-def*
by (*metis tranclp-unfold-begin*
rtranclp-$cdcl_W$*-merge-stgy'-no-step-*$cdcl_W$*-cp-or-eq*)
consider
(*V-W*) $V = W$
| (*propa*) *propagate*$^{++}$ *V W* **and** *conflicting* $W = None$
| (*propa-confl*) *V'* **where** *propagate*** *V V'* **and** *conflict V' W*
using *tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not*[*of V W*] *conflict'*
unfolding *full-unfold full1-def* **by** *meson*

then show *?thesis*
proof *cases*
case *V-W*
then show *?thesis* **using** ‹$R = V$› *n-s-R* **by** *simp*
next
case *propa*
then show *?thesis* **using** ‹$R = V$› **by** (*auto intro*: $cdcl_W$*-merge-cp.intros*)
next
case *propa-confl*
moreover
then have $cdcl_W$*-merge-cp*** V V'
by (*metis rtranclp-unfold* $cdcl_W$*-merge-cp.propagate*$'$ *r-into-rtranclp*)
ultimately show *?thesis* **using** s' ‹$R = V$› **by** *blast*
qed
next
case *dec-confl* **note** *-* = *this*(*5*)
then have *False* **using** *conflict*$'$ **unfolding** *full1-def* **by** (*auto dest*!: *tranclpD elim*: *rulesE*)
then show *?thesis* **by** *fast*
next
case *dec* **note** *T-V* = *this*(*4*)
consider
(*propa*) *propagate*$^{++}$ V W **and** *conflicting* W = *None*
| (*propa-confl*) V' **where** *propagate*** V V' **and** *conflict* V' W
using *tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not*[*of* V W] *conflict*$'$
unfolding *full1-def* **by** *meson*
then show *?thesis*
proof *cases*
case *propa*
then show *?thesis*
by (*meson T-V* $cdcl_W$*-merge-cp.propagate*$'$ *dec rtranclp.rtrancl-into-rtrancl*)
next
case *propa-confl*
then have $cdcl_W$*-merge-cp*** T V'
using *T-V* **by** (*metis rtranclp-unfold* $cdcl_W$*-merge-cp.propagate*$'$ *rtranclp.simps*)
then show *?thesis* **using** *dec propa-confl*(*2*) **by** *metis*
qed
next
case *cp*
consider
(*propa*) *propagate*$^{++}$ V W **and** *conflicting* W = *None*
| (*propa-confl*) V' **where** *propagate*** V V' **and** *conflict* V' W
using *tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not*[*of* V W] *conflict*$'$
unfolding *full1-def* **by** *meson*
then show *?thesis*
proof *cases*
case *propa*
then show *?thesis* **by** (*meson* $cdcl_W$*-merge-cp.propagate*$'$ *cp*
rtranclp.rtrancl-into-rtrancl)
next
case *propa-confl*
then show *?thesis*
using *propa-confl*(*2*) *cp*
by (*metis* (*full-types*) $cdcl_W$*-merge-cp.propagate*$'$ *rtranclp.rtrancl-into-rtrancl*
rtranclp-unfold)
qed
next

case *cp-confl*
then show *?thesis* **using** *conflict'* **unfolding** *full1-def* **by** (*fastforce dest*!: *tranclpD* *elim*!: *rulesE*)
qed
next
case (*decide' V'*)
then have *conf-V*: *conflicting V = None*
by (*auto elim*: *rulesE*)
consider
(*s'*) $cdcl_W$*-merge-stgy*** *R V*
| (*dec-confl*) *S T U* **where** $cdcl_W$*-merge-stgy*** *R S* **and** *no-step* $cdcl_W$*-merge-cp S* **and** *decide S T* **and** $cdcl_W$*-merge-cp*** *T U* **and** *conflict U V*
| (*dec*) *S T* **where** $cdcl_W$*-merge-stgy*** *R S* **and** *no-step* $cdcl_W$*-merge-cp S* **and** *decide S T* **and** $cdcl_W$*-merge-cp*** *T V* **and** *conflicting V = None*
| (*cp*) $cdcl_W$*-merge-cp*** *R V*
| (*cp-confl*) *U* **where** $cdcl_W$*-merge-cp*** *R U* **and** *conflict U V*
using *IH* **by** *meson*
then show *?thesis*
proof *cases*
case *s'*
have *confl-V'*: *conflicting V' = None* **using** *decide'*(*1*) **by** (*auto elim*: *rulesE*)
have *full*: *full1* $cdcl_W$*-cp V' W* ∨ (*V' = W* ∧ *no-step* $cdcl_W$*-cp W*)
using *decide'*(*3*) **unfolding** *full-unfold* **by** *blast*
consider
(*V'-W*) *V' = W*
| (*propa*) *propagate*++ *V' W* **and** *conflicting W = None*
| (*propa-confl*) *V''* **where** *propagate*** *V' V''* **and** *conflict V'' W*
using *tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not*[*of V W*] *decide'*
⟨*full1* $cdcl_W$*-cp V' W* ∨ *V' = W* ∧ *no-step* $cdcl_W$*-cp W*⟩ **unfolding** *full1-def*
by (*metis tranclp-*$cdcl_W$*-cp-propagate-with-conflict-or-not*)
then show *?thesis*
proof *cases*
case *V'-W*
then show *?thesis*
using *confl-V' local.decide'*(*1*,*2*) *s' conf-V*
no-step-$cdcl_W$*-cp-no-step-*$cdcl_W$*-merge-restart*[*of V*]
by *auto*
next
case *propa*
then show *?thesis* **using** *local.decide'*(*1*,*2*) *s'* **by** (*metis* $cdcl_W$*-merge-cp.simps conf-V* *no-step-*$cdcl_W$*-cp-no-step-*$cdcl_W$*-merge-restart r-into-rtranclp*)
next
case *propa-confl*
then have $cdcl_W$*-merge-cp*** *V' V''*
by (*metis rtranclp-unfold* $cdcl_W$*-merge-cp.propagate' r-into-rtranclp*)
then show *?thesis*
using *local.decide'*(*1*,*2*) *propa-confl*(*2*) *s' conf-V*
no-step-$cdcl_W$*-cp-no-step-*$cdcl_W$*-merge-restart*
by *metis*
qed
next
case (*dec*) **note** *s' = this*(*1*) **and** *dec = this*(*2*) **and** *cp = this*(*3*) **and** *ns-cp-T = this*(*4*)
have *full* $cdcl_W$*-merge-cp T V*
unfolding *full-def* **by** (*simp add*: *conf-V local.decide'*(*2*)
no-step-$cdcl_W$*-cp-no-step-*$cdcl_W$*-merge-restart ns-cp-T*)
moreover have *no-step* $cdcl_W$*-merge-cp V*

by (*simp add*: *conf-V local.decide*′(*2*) *no-step-cdcl*$_W$*-cp-no-step-cdcl*$_W$*-merge-restart*)
moreover have *no-step cdcl*$_W$*-merge-cp S*
by (*metis dec*)
ultimately have *cdcl*$_W$*-merge-stgy S V*
using *cp* **by** *blast*
then have *cdcl*$_W$*-merge-stgy*** *R V* **using** *s*′ **by** *auto*
consider
(*V*′*-W*) *V*′ = *W*
| (*propa*) *propagate*$^{++}$ *V*′ *W* **and** *conflicting W* = *None*
| (*propa-confl*) *V*″ **where** *propagate*** *V*′ *V*″ **and** *conflict V*″ *W*
using *tranclp-cdcl*$_W$*-cp-propagate-with-conflict-or-not*[*of V*′ *W*] *decide*′
unfolding *full-unfold full1-def* **by** *meson*
then show *?thesis*
proof *cases*
case *V*′*-W*
moreover have *conflicting V*′ = *None*
using *decide*′(*1*) **by** (*auto elim*: *rulesE*)
ultimately show *?thesis*
using ‹*cdcl*$_W$*-merge-stgy*** *R V*› *decide*′ ‹*no-step cdcl*$_W$*-merge-cp V*› **by** *blast*
next
case *propa*
moreover then have *cdcl*$_W$*-merge-cp V*′ *W* **by** (*blast intro*: *cdcl*$_W$*-merge-cp.intros*)
ultimately show *?thesis*
using ‹*cdcl*$_W$*-merge-stgy*** *R V*› *decide*′ ‹*no-step cdcl*$_W$*-merge-cp V*›
by (*meson r-into-rtranclp*)
next
case *propa-confl*
moreover then have *cdcl*$_W$*-merge-cp*** *V*′ *V*″
by (*metis cdcl*$_W$*-merge-cp.propagate*′ *rtranclp-unfold tranclp-unfold-end*)
ultimately show *?thesis* **using** ‹*cdcl*$_W$*-merge-stgy*** *R V*› *decide*′
‹*no-step cdcl*$_W$*-merge-cp V*› **by** (*meson r-into-rtranclp*)
qed
next
case *cp*
have *no-step cdcl*$_W$*-merge-cp V*
using *conf-V local.decide*′(*2*) *no-step-cdcl*$_W$*-cp-no-step-cdcl*$_W$*-merge-restart* **by** *auto*
then have *full cdcl*$_W$*-merge-cp R V*
unfolding *full-def* **using** *cp* **by** *fast*
then have *cdcl*$_W$*-merge-stgy*** *R V*
unfolding *full-unfold* **by** *auto*
have *full1 cdcl*$_W$*-cp V*′ *W* ∨ (*V*′ = *W* ∧ *no-step cdcl*$_W$*-cp W*)
using *decide*′(*3*) **unfolding** *full-unfold* **by** *blast*

consider
(*V*′*-W*) *V*′ = *W*
| (*propa*) *propagate*$^{++}$ *V*′ *W* **and** *conflicting W* = *None*
| (*propa-confl*) *V*″ **where** *propagate*** *V*′ *V*″ **and** *conflict V*″ *W*
using *tranclp-cdcl*$_W$*-cp-propagate-with-conflict-or-not*[*of V*′ *W*] *decide*′
unfolding *full-unfold full1-def* **by** *meson*
then show *?thesis*

proof *cases*
case *V*′*-W*
moreover have *conflicting V*′ = *None*
using *decide*′(*1*) **by** (*auto elim*: *rulesE*)
ultimately show *?thesis*

using ‹$cdcl_W$*-merge-stgy*** R V› *decide'* ‹*no-step* $cdcl_W$*-merge-cp* V› **by** *blast*
next
case *propa*
moreover then have $cdcl_W$*-merge-cp* V' W
by (*blast intro*: $cdcl_W$*-merge-cp.intros*)
ultimately show *?thesis* **using** ‹$cdcl_W$*-merge-stgy*** R V› *decide'*
‹*no-step* $cdcl_W$*-merge-cp* V› **by** (*meson r-into-rtranclp*)
next
case *propa-confl*
moreover then have $cdcl_W$*-merge-cp*** V' V''
by (*metis* $cdcl_W$*-merge-cp.propagate'* *rtranclp-unfold tranclp-unfold-end*)
ultimately show *?thesis* **using** ‹$cdcl_W$*-merge-stgy*** R V› *decide'*
‹*no-step* $cdcl_W$*-merge-cp* V› **by** (*meson r-into-rtranclp*)
qed
next
case (*dec-confl*)
show *?thesis* **using** *conf-V dec-confl*(*5*) **by** (*auto elim*!: *rulesE*
simp del: *state-simp simp*: *state-eq-def*)
next
case *cp-confl*
then show *?thesis* **using** *decide'* **apply** − **by** (*intro HOL.disjI2*) (*fastforce elim*: *rulesE*
simp del: *state-simp simp*: *state-eq-def*)
qed
next
case (*bj'* V')
then have ¬*no-step* $cdcl_W$*-bj* V
by (*auto dest*: *tranclpD simp*: *full1-def*)
then consider
(s') $cdcl_W$*-merge-stgy*** R V **and** *conflicting* V = *None*
| (*dec-confl*) S T U **where** $cdcl_W$*-merge-stgy*** R S **and** *no-step* $cdcl_W$*-merge-cp* S **and**
decide S T **and** $cdcl_W$*-merge-cp*** T U **and** *conflict* U V
| (*dec*) S T **where** $cdcl_W$*-merge-stgy*** R S **and** *no-step* $cdcl_W$*-merge-cp* S **and** *decide* S T
and $cdcl_W$*-merge-cp*** T V **and** *conflicting* V = *None*
| (*cp*) $cdcl_W$*-merge-cp*** R V **and** *conflicting* V = *None*
| (*cp-confl*) U **where** $cdcl_W$*-merge-cp*** R U **and** *conflict* U V
using *IH* **by** *meson*
then show *?thesis*
proof *cases*
case s' **note** - = *this*(*2*)
then have *False*
using *bj'*(*1*) **unfolding** *full1-def* **by** (*force dest*!: *tranclpD simp*: $cdcl_W$*-bj.simps*
elim: *rulesE*)
then show *?thesis* **by** *fast*
next
case *dec* **note** - = *this*(*5*)
then have *False*
using *bj'*(*1*) **unfolding** *full1-def* **by** (*force dest*!: *tranclpD simp*: $cdcl_W$*-bj.simps*
elim: *rulesE*)
then show *?thesis* **by** *fast*
next
case *dec-confl*
then have $cdcl_W$*-merge-cp* U V'
using *bj'* $cdcl_W$*-merge-cp.intros*(*1*)[*of* U V V'] **by** (*simp add*: *full-unfold*)
then have $cdcl_W$*-merge-cp*** T V'
using *dec-confl*(*4*) **by** *simp*
consider

```
      (V'-W) V' = W
    | (propa) propagate⁺⁺ V' W and conflicting W = None
    | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
    using tranclp-cdcl_W-cp-propagate-with-conflict-or-not[of V' W] bj'(3)
    unfolding full-unfold full1-def by meson
  then show ?thesis
    proof cases
      case V'-W
      then have no-step cdcl_W-cp V'
        using bj'(3) unfolding full-def by auto
      then have no-step cdcl_W-merge-cp V'
        by (metis cdcl_W-cp.propagate' cdcl_W-merge-cp.cases tranclpD
          no-step-cdcl_W-cp-no-conflict-no-propagate(1) )
      then have full1 cdcl_W-merge-cp T V'
        unfolding full1-def using ‹cdcl_W-merge-cp U V'› dec-confl(4) by auto
      then have full cdcl_W-merge-cp T V'
        by (simp add: full-unfold)
      then have cdcl_W-merge-stgy S V'
        using dec-confl(3) cdcl_W-merge-stgy.fw-s-decide ‹no-step cdcl_W-merge-cp S› by blast
      then have cdcl_W-merge-stgy** R V'
        using ‹cdcl_W-merge-stgy** R S› by auto
      show ?thesis
        proof cases
          assume conflicting W = None
          then show ?thesis using ‹cdcl_W-merge-stgy** R V'› ‹V' = W› by auto
        next
          assume conflicting W ≠ None
          then show ?thesis
            using ‹cdcl_W-merge-stgy** R V'› ‹V' = W› by (metis ‹cdcl_W-merge-cp U V'›
              conflictE conflicting-not-true-rtranclp-cdcl_W-merge-cp-no-step-cdcl_W-bj
              dec-confl(5) r-into-rtranclp)
        qed
    next
      case propa
      moreover then have cdcl_W-merge-cp V' W by (blast intro: cdcl_W-merge-cp.intros)
     ultimately show ?thesis using decide' by (meson ‹cdcl_W-merge-cp** T V'› dec-confl(1−3)
        rtranclp.rtrancl-into-rtrancl)
    next
      case propa-confl
      moreover then have cdcl_W-merge-cp** V' V''
        by (metis cdcl_W-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
    ultimately show ?thesis by (meson ‹cdcl_W-merge-cp** T V'› dec-confl(1−3) rtranclp-trans)
    qed
next
  case cp note - = this(2)
  then show ?thesis using bj'(1) ‹¬ no-step cdcl_W-bj V›
    conflicting-not-true-rtranclp-cdcl_W-merge-cp-no-step-cdcl_W-bj by auto
next
  case cp-confl
  then have cdcl_W-merge-cp U V' by (simp add: cdcl_W-merge-cp.conflict' full-unfold
    local.bj'(1))
  consider
      (V'-W) V' = W
    | (propa) propagate⁺⁺ V' W and conflicting W = None
    | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
    using tranclp-cdcl_W-cp-propagate-with-conflict-or-not[of V' W] bj'
```

```
          unfolding full-unfold full1-def by meson
        then show ?thesis

          proof cases
            case V'-W
            show ?thesis
              proof cases
                assume conflicting V' = None
                then show ?thesis
                  using V'-W ‹cdclW-merge-cp U V'› cp-confl(1) by force
              next
                assume confl: conflicting V' ≠ None
                then have no-step cdclW-merge-stgy V'
                  by (fastforce simp: cdclW-merge-stgy.simps full1-def full-def
                    cdclW-merge-cp.simps dest!: tranclpD elim: rulesE)
                have no-step cdclW-merge-cp V'
                  using confl by (auto simp: full1-def full-def cdclW-merge-cp.simps
                  dest!: tranclpD elim: rulesE)
                moreover have cdclW-merge-cp U W
                  using V'-W ‹cdclW-merge-cp U V'› by blast
                ultimately have full1 cdclW-merge-cp R V'
                  using cp-confl(1) V'-W unfolding full1-def by auto
                then have cdclW-merge-stgy R V'
                  by auto
                moreover have no-step cdclW-merge-stgy V'
                  using confl ‹no-step cdclW-merge-cp V'› by (auto simp: cdclW-merge-stgy.simps
                    full1-def dest!: tranclpD elim: rulesE)
                ultimately have cdclW-merge-stgy** R V' by auto
                { fix ss :: 'st
                  have cdclW-merge-cp U W
                    using V'-W ‹cdclW-merge-cp U V'› by blast
                  then have ¬ cdclW-bj W ss
                    by (meson conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj
                      cp-confl(1) rtranclp.rtrancl-into-rtrancl step.prems)
                  then have cdclW-merge-stgy** R W ∧ conflicting W = None ∨
                    cdclW-merge-stgy** R W ∧ ¬ cdclW-bj W ss
                    using V'-W ‹cdclW-merge-stgy** R V'› by presburger }
                then show ?thesis
                  by presburger
              qed
          next
            case propa
            moreover then have cdclW-merge-cp V' W
              by (blast intro: cdclW-merge-cp.intros)
            ultimately show ?thesis using ‹cdclW-merge-cp U V'› cp-confl(1) by force
          next
            case propa-confl
            moreover then have cdclW-merge-cp** V' V''
              by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
            ultimately show ?thesis
              using ‹cdclW-merge-cp U V'› cp-confl(1) by (metis rtranclp.rtrancl-into-rtrancl
                rtranclp-trans)
          qed
      qed
    qed
qed
```

lemma *decide-rtranclp-cdcl$_W$-s$'$-rtranclp-cdcl$_W$-s$'$*:
  assumes
    dec: *decide S T* **and**
    cdcl$_W$-s$'^{**}$ T U **and**
    n-s-S: *no-step cdcl$_W$-cp S* **and**
    no-step cdcl$_W$-cp U
  shows *cdcl$_W$-s$'^{**}$ S U*
  using *assms(2,4)*
proof *induction*
  case (*step U V*) **note** *st* = *this*(*1*) **and** *s$'$* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *n-s* = *this*(*4*)
  consider
      (*TU*) *T* = *U*
    | (*s$'$-st*) *T$'$* **where** *cdcl$_W$-s$'$ T T$'$* **and** *cdcl$_W$-s$'^{**}$ T$'$ U*
    using *st*[*unfolded rtranclp-unfold*] **by** (*auto dest*!: *tranclpD*)
  then show *?case*
    proof *cases*
      case *TU*
      then show *?thesis*
        proof −
          assume *a1*: *T* = *U*
          then have *f2*: *cdcl$_W$-s$'$ T V*
            using *s$'$* **by** *force*
          obtain *ss* :: *$'$st* **where**
            ss: *cdcl$_W$-s$'^{**}$ S T* ∨ *cdcl$_W$-cp T ss*
            using *a1 step.IH* **by** *blast*−
          obtain *ssa* :: *$'$st* ⇒ *$'$st* **where**
            f3: ∀ *s sa sb*. (¬ *decide s sa* ∨ *cdcl$_W$-cp s* (*ssa s*) ∨ ¬ *full cdcl$_W$-cp sa sb*)
              ∨ *cdcl$_W$-s$'$ s sb*
            using *cdcl$_W$-s$'$.decide$'$* **by** *moura*
          have ∀ *s sa*. ¬ *cdcl$_W$-s$'$ s sa* ∨ *full1 cdcl$_W$-cp s sa* ∨
            (∃ *sb*. *decide s sb* ∧ *no-step cdcl$_W$-cp s* ∧ *full cdcl$_W$-cp sb sa*) ∨
            (∃ *sb*. *full1 cdcl$_W$-bj s sb* ∧ *no-step cdcl$_W$-cp s* ∧ *full cdcl$_W$-cp sb sa*)
            by (*metis cdcl$_W$-s$'$E*)
          then have ∃ *s*. *cdcl$_W$-s$'^{**}$ S s* ∧ *cdcl$_W$-s$'$ s V*
            using *f3 ss f2* **by** (*metis dec full1-is-full n-s-S rtranclp-unfold*)
          then show *?thesis*
            by *force*
        qed
    next
      case (*s$'$-st T$'$*) **note** *s$'$-T$'$* = *this*(*1*) **and** *st* = *this*(*2*)
      have *cdcl$_W$-s$'^{**}$ S T$'$*
        using *s$'$-T$'$*
        proof *cases*
          case *conflict$'$*
          then have *cdcl$_W$-s$'$ S T$'$*
             using *dec cdcl$_W$-s$'$.decide$'$ n-s-S* **by** (*simp add*: *full-unfold*)
          then show *?thesis*
             using *st* **by** *auto*
        next
          case (*decide$'$ T$''$*)
          then have *cdcl$_W$-s$'$ S T*
             using *dec cdcl$_W$-s$'$.decide$'$ n-s-S* **by** (*simp add*: *full-unfold*)
          then show *?thesis* **using** *decide$'$ s$'$-T$'$* **by** *auto*
        next
          case *bj$'$*

    then have *False*
      using *dec* **unfolding** *full1-def* **by** (*fastforce dest*!: *tranclpD simp*: $cdcl_W$*-bj.simps*
        elim: *rulesE*)
    then show *?thesis* **by** *fast*
  qed
  then show *?thesis* **using** *s′ st* **by** *auto*
qed
next
  case *base*
  then have *full* $cdcl_W$*-cp T T*
    by (*simp add*: *full-unfold*)
  then show *?case*
    using $cdcl_W$*-s′.simps dec n-s-S* **by** *auto*
qed

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$*-s′*:
  assumes
    $cdcl_W$*-merge-stgy*** *R V* **and**
    inv: $cdcl_W$*-all-struct-inv R*
  shows $cdcl_W$*-s′*** *R V*
  using *assms*(*1*)
proof *induction*
  case *base*
  then show *?case* **by** *simp*
next
  case (*step S T*) **note** *st* = *this*(*1*) **and** *fw* = *this*(*2*) **and** *IH* = *this*(*3*)
  have $cdcl_W$*-all-struct-inv S*
    using *inv rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$ *st* **by** *blast*
  from *fw* **show** *?case*
    proof (*cases rule*: $cdcl_W$*-merge-stgy-cases*)
      case *fw-s-cp*
      have $\bigwedge$*s*. ¬ *full* $cdcl_W$*-merge-cp s S*
        using *fw-s-cp* **unfolding** *full-def full1-def* **by** (*metis tranclp-unfold-begin*)
      then have *S* = *R*
        using *fw-s-cp* **unfolding** *full1-def* **by** (*metis* $cdcl_W$*-cp.conflict′* $cdcl_W$*-cp.propagate′*
          $cdcl_W$*-merge-cp.cases tranclp-unfold-begin inv st*
          rtranclp-$cdcl_W$*-merge-stgy′-no-step-*$cdcl_W$*-cp-or-eq*)
      then have *full1* $cdcl_W$*-s′-without-decide R T*
        using *inv local.fw-s-cp*
        by (*blast intro*: *conflicting-true-full1-*$cdcl_W$*-merge-cp-imp-full1-*$cdcl_W$*-s′-without-decode*)
      then show *?thesis* **unfolding** *full1-def*
        by (*metis* (*no-types*) *rtranclp-*$cdcl_W$*-s′-without-decide-rtranclp-*$cdcl_W$*-s′ rtranclp-unfold*)
    next
      case (*fw-s-decide S′*) **note** *dec* = *this*(*1*) **and** *n-S* = *this*(*2*) **and** *full* = *this*(*3*)
      moreover then have *conflicting S′* = *None*
        by (*auto elim*: *rulesE*)
      ultimately have *full* $cdcl_W$*-s′-without-decide S′ T*
        by (*meson* ‹$cdcl_W$*-all-struct-inv S*› $cdcl_W$*-merge-restart-*$cdcl_W$ *fw-r-decide*
          rtranclp-$cdcl_W$*-all-struct-inv-inv*
          conflicting-true-full-$cdcl_W$*-merge-cp-iff-full-*$cdcl_W$*-s′-without-decode*)
      then have *a1*: $cdcl_W$*-s′*** *S′ T*
        unfolding *full-def* **by** (*metis* (*full-types*) *rtranclp-*$cdcl_W$*-s′-without-decide-rtranclp-*$cdcl_W$*-s′*)
      have $cdcl_W$*-merge-stgy*** *S T*
        using *fw* **by** *blast*
      then have $cdcl_W$*-s′*** *S T*
        using *decide-rtranclp-*$cdcl_W$*-s′-rtranclp-*$cdcl_W$*-s′ a1* **by** (*metis* ‹$cdcl_W$*-all-struct-inv S*› *dec*

n-S no-step-cdcl$_W$-merge-cp-no-step-cdcl$_W$-cp cdcl$_W$-all-struct-inv-def
rtranclp-cdcl$_W$-merge-stgy$'$-no-step-cdcl$_W$-cp-or-eq)
then show *?thesis* **using** *IH* **by** *auto*
qed
qed

lemma *rtranclp-cdcl$_W$-merge-stgy-distinct-mset-clauses*:
assumes *invR*: *cdcl$_W$-all-struct-inv R* **and**
st: *cdcl$_W$-merge-stgy** R S* **and**
dist: *distinct-mset* (*clauses R*) **and**
R: *trail R* = []
shows *distinct-mset* (*clauses S*)
using *rtranclp-cdcl$_W$-stgy-distinct-mset-clauses*[*OF invR - dist R*]
invR st rtranclp-mono[*of cdcl$_W$-s$'$ cdcl$_W$-stgy***] *cdcl$_W$-s$'$-is-rtranclp-cdcl$_W$-stgy*
by (*auto dest*!: *cdcl$_W$-s$'$-is-rtranclp-cdcl$_W$-stgy rtranclp-cdcl$_W$-merge-stgy-rtranclp-cdcl$_W$-s$'$*)

lemma *no-step-cdcl$_W$-s$'$-no-step-cdcl$_W$-merge-stgy*:
assumes
inv: *cdcl$_W$-all-struct-inv R* **and** *s$'$*: *no-step cdcl$_W$-s$'$ R*
shows *no-step cdcl$_W$-merge-stgy R*
proof −
{ fix *ss* :: *$'$st*
obtain *ssa* :: *$'$st ⇒ $'$st ⇒ $'$st* **where**
ff1: $\bigwedge$*s sa*. ¬ *cdcl$_W$-merge-stgy s sa* ∨ *full1 cdcl$_W$-merge-cp s sa* ∨ *decide s* (*ssa s sa*)
using *cdcl$_W$-merge-stgy.cases* **by** *moura*
obtain *ssb* :: (*$'$st ⇒ $'$st ⇒ bool*) ⇒ *$'$st ⇒ $'$st ⇒ $'$st* **where**
ff2: $\bigwedge$*p s sa*. ¬ *p^{++} s sa* ∨ *p s* (*ssb p s sa*)
by (*meson tranclp-unfold-begin*)
obtain *ssc* :: *$'$st ⇒ $'$st* **where**
ff3: $\bigwedge$*s sa sb*. (¬ *cdcl$_W$-all-struct-inv s* ∨ ¬ *cdcl$_W$-cp s sa* ∨ *cdcl$_W$-s$'$ s* (*ssc s*))
∧ (¬ *cdcl$_W$-all-struct-inv s* ∨ ¬ *cdcl$_W$-o s sb* ∨ *cdcl$_W$-s$'$ s* (*ssc s*))
using *n-step-cdcl$_W$-stgy-iff-no-step-cdcl$_W$-cl-cdcl$_W$-o* **by** *moura*
then have *ff4*: $\bigwedge$*s*. ¬ *cdcl$_W$-o R s*
using *s$'$ inv* **by** *blast*
have *ff5*: $\bigwedge$*s*. ¬ *cdcl$_W$-cp^{++} R s*
using *ff3 ff2 s$'$* **by** (*metis inv*)
have $\bigwedge$*s*. ¬ *cdcl$_W$-bj^{++} R s*
using *ff4 ff2* **by** (*metis bj*)
then have $\bigwedge$*s*. ¬ *cdcl$_W$-s$'$-without-decide R s*
using *ff5* **by** (*simp add*: *cdcl$_W$-s$'$-without-decide.simps full1-def*)
then have ¬ *cdcl$_W$-s$'$-without-decide^{++} R ss*
using *ff2* **by** *blast*
then have ¬ *full1 cdcl$_W$-s$'$-without-decide R ss*
by (*simp add*: *full1-def*)
then have ¬ *cdcl$_W$-merge-stgy R ss*
using *ff4 ff1 conflicting-true-full1-cdcl$_W$-merge-cp-imp-full1-cdcl$_W$-s$'$-without-decode inv*
by *blast* **}**
then show *?thesis*
by *fastforce*
qed
end

Termination and full Equivalence

We will discharge the assumption later using NOT's proof of termination.

locale *conflict-driven-clause-learning*$_W$*-termination* =
  conflict-driven-clause-learning$_W$ +
  assumes *wf-cdcl*$_W$*-merge-inv*: *wf* {(*T*, *S*). *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge S T*}
begin

lemma *wf-tranclp-cdcl*$_W$*-merge*: *wf* {(*T*, *S*). *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge*$^{++}$ *S T*}
  using *wf-trancl*[*OF wf-cdcl*$_W$*-merge-inv*]
  apply (*rule wf-subset*)
  by (*auto simp*: *trancl-set-tranclp*
    cdcl$_W$*-all-struct-inv-tranclp-cdcl*$_W$*-merge-tranclp-cdcl*$_W$*-merge-cdcl*$_W$*-all-struct-inv*)

lemma *wf-cdcl*$_W$*-merge-cp*:
  wf{(*T*, *S*). *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge-cp S T*}
  using *wf-tranclp-cdcl*$_W$*-merge* **by** (*rule wf-subset*) (*auto simp*: *cdcl*$_W$*-merge-cp-tranclp-cdcl*$_W$*-merge*)

lemma *wf-cdcl*$_W$*-merge-stgy*:
  wf{(*T*, *S*). *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge-stgy S T*}
  using *wf-tranclp-cdcl*$_W$*-merge* **by** (*rule wf-subset*)
  (*auto simp add*: *cdcl*$_W$*-merge-stgy-tranclp-cdcl*$_W$*-merge*)

lemma *cdcl*$_W$*-merge-cp-obtain-normal-form*:
  assumes *inv*: *cdcl*$_W$*-all-struct-inv R*
  obtains *S* **where** *full cdcl*$_W$*-merge-cp R S*
proof −
  obtain *S* **where** *full* (λ*S T*. *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge-cp S T*) *R S*
    using *wf-exists-normal-form-full*[*OF wf-cdcl*$_W$*-merge-cp*] **by** *blast*
  then have
    st: (λ*S T*. *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge-cp S T*)** *R S* **and**
    n-s: *no-step* (λ*S T*. *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge-cp S T*) *S*
    unfolding *full-def* **by** *blast+*
  have *cdcl*$_W$*-merge-cp*** *R S*
    using *st* **by** *induction auto*
  moreover
    have *cdcl*$_W$*-all-struct-inv S*
      using *st inv*
      apply (*induction rule*: *rtranclp-induct*)
        apply *simp*
      by (*meson r-into-rtranclp rtranclp-cdcl*$_W$*-all-struct-inv-inv*
        rtranclp-cdcl$_W$*-merge-cp-rtranclp-cdcl*$_W$)
    then have *no-step cdcl*$_W$*-merge-cp S*
      using *n-s* **by** *auto*
  ultimately show *?thesis*
    using *that* **unfolding** *full-def* **by** *blast*
qed

lemma *no-step-cdcl*$_W$*-merge-stgy-no-step-cdcl*$_W$*-s′*:
  assumes
    inv: *cdcl*$_W$*-all-struct-inv R* **and**
    confl: *conflicting R* = *None* **and**
    n-s: *no-step cdcl*$_W$*-merge-stgy R*
  shows *no-step cdcl*$_W$*-s′ R*
proof (*rule ccontr*)
  assume ¬ *?thesis*
  then obtain *S* **where** *cdcl*$_W$*-s′ R S* **by** *auto*
  then show *False*
    proof *cases*

case *conflict'*
then obtain S' **where** *full1 cdcl$_W$-merge-cp R S'*
proof −
obtain R' **where**
cdcl$_W$-merge-cp R R'
using *inv* **unfolding** *cdcl$_W$-all-struct-inv-def* **by** (*meson confl*
cdcl$_W$-s'-without-decide.simps conflict'
conflicting-true-no-step-cdcl$_W$-merge-cp-no-step-s'-without-decide)
then show *?thesis*
using *that* **by** (*metis cdcl$_W$-merge-cp-obtain-normal-form full-unfold inv*)
qed
then show *False* **using** *n-s* **by** *blast*
next
case (*decide' R'*)
then have *cdcl$_W$-all-struct-inv R'*
using *inv cdcl$_W$-all-struct-inv-inv cdcl$_W$.other cdcl$_W$-o.decide* **by** *meson*
then obtain R'' **where** *full cdcl$_W$-merge-cp R' R''*
using *cdcl$_W$-merge-cp-obtain-normal-form* **by** *blast*
moreover have *no-step cdcl$_W$-merge-cp R*
by (*simp add*: *confl local.decide'*(*2*) *no-step-cdcl$_W$-cp-no-step-cdcl$_W$-merge-restart*)
ultimately show *False* **using** *n-s cdcl$_W$-merge-stgy.intros local.decide'*(*1*) **by** *blast*
next
case (*bj' R'*)
then show *False*
using *confl no-step-cdcl$_W$-cp-no-step-cdcl$_W$-s'-without-decide inv*
unfolding *cdcl$_W$-all-struct-inv-def* **by** *auto*
qed
qed

lemma *rtranclp-cdcl$_W$-merge-cp-no-step-cdcl$_W$-bj*:
assumes *conflicting R = None* **and** *cdcl$_W$-merge-cp** R S*
shows *no-step cdcl$_W$-bj S*
using *assms conflicting-not-true-rtranclp-cdcl$_W$-merge-cp-no-step-cdcl$_W$-bj* **by** *auto*

lemma *rtranclp-cdcl$_W$-merge-stgy-no-step-cdcl$_W$-bj*:
assumes *confl*: *conflicting R = None* **and** *cdcl$_W$-merge-stgy** R S*
shows *no-step cdcl$_W$-bj S*
using *assms*(*2*)
proof *induction*
case *base*
then show *?case*
using *confl* **by** (*auto simp*: *cdcl$_W$-bj.simps elim*: *rulesE*)
next
case (*step S T*) **note** *st = this*(*1*) **and** *fw = this*(*2*) **and** *IH = this*(*3*)
have *confl-S*: *conflicting S = None*
using *fw* **apply** *cases*
by (*auto simp*: *full1-def cdcl$_W$-merge-cp.simps dest*!: *tranclpD elim*: *rulesE*)
from *fw* **show** *?case*
proof *cases*
case *fw-s-cp*
then show *?thesis*
using *rtranclp-cdcl$_W$-merge-cp-no-step-cdcl$_W$-bj confl-S*
by (*simp add*: *full1-def tranclp-into-rtranclp*)
next
case (*fw-s-decide S'*)
moreover then have *conflicting S' = None* **by** (*auto elim*: *rulesE*)

ultimately show *?thesis*
using *conflicting-not-true-rtranclp-cdcl*$_W$*-merge-cp-no-step-cdcl*$_W$*-bj*
unfolding *full-def* **by** *meson*
qed
qed

end

end
theory *CDCL-WNOT*
imports *CDCL-NOT CDCL-W-Termination CDCL-W-Merge*
begin

6.3 Link between Weidenbach's and NOT's CDCL

6.3.1 Inclusion of the states

declare *upt.simps*(*2*)[*simp del*]

fun *convert-ann-lit-from-W* **where**
convert-ann-lit-from-W (*Propagated L -*) = *Propagated L* () |
convert-ann-lit-from-W (*Decided L*) = *Decided L*

abbreviation *convert-trail-from-W* ::
('*v*, '*mark*) *ann-lits*
⇒ ('*v*, *unit*) *ann-lits* **where**
convert-trail-from-W ≡ *map convert-ann-lit-from-W*

lemma *lits-of-l-convert-trail-from-W*[*simp*]:
lits-of-l (*convert-trail-from-W M*) = *lits-of-l M*
by (*induction rule*: *ann-lit-list-induct*) *simp-all*

lemma *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-ann-lit-from-W L*) = *lit-of L*
by (*cases L*) *auto*

lemma *no-dup-convert-from-W*[*simp*]:
no-dup (*convert-trail-from-W M*) ⟷ *no-dup M*
by (*auto simp*: *comp-def*)

lemma *convert-trail-from-W-true-annots*[*simp*]:
convert-trail-from-W M ⊨*as C* ⟷ *M* ⊨*as C*
by (*auto simp*: *true-annots-true-cls image-image lits-of-def*)

lemma *defined-lit-convert-trail-from-W*[*simp*]:
defined-lit (*convert-trail-from-W S*) *L* ⟷ *defined-lit S L*
by (*auto simp*: *defined-lit-map image-comp*)

The values *0* and {#} are dummy values.

consts *dummy-cls* :: '*cls*
fun *convert-ann-lit-from-NOT*
:: ('*v*, '*mark*) *ann-lit* ⇒ ('*v*, '*cls*) *ann-lit* **where**
convert-ann-lit-from-NOT (*Propagated L -*) = *Propagated L dummy-cls* |
convert-ann-lit-from-NOT (*Decided L*) = *Decided L*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT $\equiv$ *map convert-ann-lit-from-NOT*

lemma *undefined-lit-convert-trail-from-NOT*[*simp*]:
undefined-lit (*convert-trail-from-NOT F*) *L* $\longleftrightarrow$ *undefined-lit F L*
by (*induction F rule*: *ann-lit-list-induct*) (*auto simp*: *defined-lit-map*)

lemma *lits-of-l-convert-trail-from-NOT*:
lits-of-l (*convert-trail-from-NOT F*) = *lits-of-l F*
by (*induction F rule*: *ann-lit-list-induct*) *auto*

lemma *convert-trail-from-W-from-NOT*[*simp*]:
convert-trail-from-W (*convert-trail-from-NOT M*) = *M*
by (*induction rule*: *ann-lit-list-induct*) *auto*

lemma *convert-trail-from-W-convert-lit-from-NOT*[*simp*]:
convert-ann-lit-from-W (*convert-ann-lit-from-NOT L*) = *L*
by (*cases L*) *auto*

abbreviation $trail_{NOT}$ **where**
$trail_{NOT}$ *S* $\equiv$ *convert-trail-from-W* (*fst S*)

lemma *undefined-lit-convert-trail-from-W*[*iff*]:
undefined-lit (*convert-trail-from-W M*) *L* $\longleftrightarrow$ *undefined-lit M L*
by (*auto simp*: *defined-lit-map image-comp*)

lemma *lit-of-convert-ann-lit-from-NOT*[*iff*]:
lit-of (*convert-ann-lit-from-NOT L*) = *lit-of L*
by (*cases L*) *auto*

sublocale $state_W$ $\subseteq$ *dpll-state-ops*
λS. *convert-trail-from-W* (*trail S*)
clauses
λL *S*. *cons-trail* (*convert-ann-lit-from-NOT L*) *S*
λS. *tl-trail S*
λC *S*. *add-learned-cls C S*
λC *S*. *remove-cls C S*
by *unfold-locales*

sublocale $state_W$ $\subseteq$ *dpll-state*
λS. *convert-trail-from-W* (*trail S*)
clauses
λL *S*. *cons-trail* (*convert-ann-lit-from-NOT L*) *S*
λS. *tl-trail S*
λC *S*. *add-learned-cls C S*
λC *S*. *remove-cls C S*
by *unfold-locales* (*auto simp*: *map-tl o-def*)

context $state_W$
begin
declare $state\text{-}simp_{NOT}$[*simp del*]
end

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W$ $\subseteq$ $cdcl_{NOT}$*-merge-bj-learn-ops*
λS. *convert-trail-from-W* (*trail S*)
clauses

$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
λ- -. *True*
λ- $S.$ *conflicting* S = *None*
$\lambda C\ C'\ L'\ S\ T.$ *backjump-l-cond* $C\ C'\ L'\ S\ T$
$\wedge$ *distinct-mset* $(C' + \{\#L'\#\}) \wedge \neg$*tautology* $(C' + \{\#L'\#\})$
by *unfold-locales*

thm $cdcl_{NOT}$*-merge-bj-learn-proxy.axioms*
sublocale *conflict-driven-clause-learning*$_W$ $\subseteq$ $cdcl_{NOT}$*-merge-bj-learn-proxy*
$\lambda S.$ *convert-trail-from-W* (*trail* S)
clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$

λ- -. *True*
λ- $S.$ *conflicting* S = *None*
backjump-l-cond
inv_{NOT}
proof (*unfold-locales*, *goal-cases*)
case *2*
then show *?case* **using** $cdcl_{NOT}$*-merged-bj-learn-no-dup-inv* **by** (*auto simp*: *comp-def*)
next
case (*1* C' S C F' K F L)
moreover
let *?C'* = *remdups-mset* C'
have $L \notin\# C'$
using ‹$F \models$*as CNot* C'› ‹*undefined-lit* F L› *Decided-Propagated-in-iff-in-lits-of-l*
in-CNot-implies-uminus(*2*) **by** *fast*
then have *distinct-mset* (*?C'* + $\{\#L\#\}$)
by (*simp add*: *distinct-mset-single-add*)
moreover
have *no-dup* F
using ‹inv_{NOT} S› ‹*convert-trail-from-W* (*trail* S) = F' @ *Decided* K # F›
unfolding inv_{NOT}*-def*
by (*smt comp-apply distinct.simps*(*2*) *distinct-append list.simps*(*9*) *map-append*
no-dup-convert-from-W)
then have *consistent-interp* (*lits-of-l* F)
using *distinct-consistent-interp* **by** *blast*
then have $\neg$ *tautology* C'
using ‹$F \models$*as CNot* C'› *consistent-CNot-not-tautology true-annots-true-cls* **by** *blast*
then have $\neg$ *tautology* (*?C'* + $\{\#L\#\}$)
using ‹$F \models$*as CNot* C'› ‹*undefined-lit* F L› **by** (*metis CNot-remdups-mset*
Decided-Propagated-in-iff-in-lits-of-l add.commute in-CNot-uminus tautology-add-single
tautology-remdups-mset true-annot-singleton true-annots-def)
show *?case*
proof –
have *f2*: *no-dup* (*convert-trail-from-W* (*trail* S))
using ‹inv_{NOT} S› **unfolding** inv_{NOT}*-def* **by** (*simp add*: *o-def*)
have *f3*: *atm-of* $L \in$ *atms-of-mm* (*clauses* S)
$\cup$ *atm-of* ‘ *lits-of-l* (*convert-trail-from-W* (*trail* S))
using ‹*convert-trail-from-W* (*trail* S) = F' @ *Decided* K # F›

‹atm-of L ∈ atms-of-mm (clauses S) ∪ atm-of ' lits-of-l (F' @ Decided K # F)› **by** *auto*
have *f4*: *clauses S* $\models$*pm remdups-mset C' + {#L#}*
by (*metis* (*no-types*) ‹*L* ∉# *C'*› ‹*clauses S* $\models$*pm C' + {#L#}*› *remdups-mset-singleton-sum*(*2*)
true-clss-cls-remdups-mset union-commute)
have *F* $\models$*as CNot* (*remdups-mset C'*)
by (*simp add*: ‹*F* $\models$*as CNot C'*›)
have *Ex* (*backjump-l S*)
apply *standard*
apply (*rule backjump-l.intros*[*OF - f2*, *of - - -*])
using *f4 f3 f2* ‹¬ *tautology* (*remdups-mset C' + {#L#}*)›
calculation(*2−5*,*9*) ‹*F* $\models$*as CNot* (*remdups-mset C'*)›
state-eq$_{NOT}$*-ref* **unfolding** *backjump-l-cond-def* **by** *blast+*
then show *?thesis*
by *blast*
qed
qed

sublocale *conflict-driven-clause-learning*$_W$ ⊆ *cdcl*$_{NOT}$*-merge-bj-learn-proxy2*
λS. convert-trail-from-W (*trail S*)
clauses
λL S. cons-trail (*convert-ann-lit-from-NOT L*) *S*
λS. tl-trail S
λC S. add-learned-cls C S
λC S. remove-cls C S
λ- -. True
λ- S. conflicting S = None backjump-l-cond inv$_{NOT}$
by *unfold-locales*

sublocale *conflict-driven-clause-learning*$_W$ ⊆ *cdcl*$_{NOT}$*-merge-bj-learn*
λS. convert-trail-from-W (*trail S*)
clauses
λL S. cons-trail (*convert-ann-lit-from-NOT L*) *S*
λS. tl-trail S
λC S. add-learned-cls C S
λC S. remove-cls C S
backjump-l-cond
λ- -. True
λ- S. conflicting S = None inv$_{NOT}$
apply *unfold-locales*
using *dpll-bj-no-dup* **apply** (*simp add*: *comp-def*)
using *cdcl*$_{NOT}$*.simps cdcl*$_{NOT}$*-no-dup no-dup-convert-from-W* **unfolding** *inv*$_{NOT}$*-def* **by** *blast*

context *conflict-driven-clause-learning*$_W$
begin

Notations are lost while proving locale inclusion:

notation *state-eq*$_{NOT}$ (**infix** $\sim_{NOT}$ *50*)

6.3.2 Additional Lemmas between NOT and W states

lemma *trail*$_W$*-eq-reduce-trail-to*$_{NOT}$*-eq*:
trail S = trail T ⟹ *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = *trail* (*reduce-trail-to*$_{NOT}$ *F T*)
proof (*induction F S arbitrary*: *T rule*: *reduce-trail-to*$_{NOT}$*.induct*)
case (*1 F S T*) **note** *IH* = *this*(*1*) **and** *tr* = *this*(*2*)
then have [] = *convert-trail-from-W* (*trail S*)
∨ *length F = length* (*convert-trail-from-W* (*trail S*))

$\vee$ *trail* (*reduce-trail-to*$_{NOT}$ *F* (*tl-trail S*)) = *trail* (*reduce-trail-to*$_{NOT}$ *F* (*tl-trail T*))
using *IH* **by** (*metis* (*no-types*) *trail-tl-trail*)
then show *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = *trail* (*reduce-trail-to*$_{NOT}$ *F T*)
using *tr* **by** (*metis* (*no-types*) *reduce-trail-to*$_{NOT}$*.elims*)
qed

lemma *trail-reduce-trail-to*$_{NOT}$*-add-learned-cls*:
no-dup (*trail S*) $\Longrightarrow$
trail (*reduce-trail-to*$_{NOT}$ *M* (*add-learned-cls D S*)) = *trail* (*reduce-trail-to*$_{NOT}$ *M S*)
by (*rule trail*$_W$*-eq-reduce-trail-to*$_{NOT}$*-eq*) *simp*

lemma *reduce-trail-to*$_{NOT}$*-reduce-trail-convert*:
reduce-trail-to$_{NOT}$ *C S* = *reduce-trail-to* (*convert-trail-from-NOT C*) *S*
apply (*induction C S rule*: *reduce-trail-to*$_{NOT}$*.induct*)
apply (*subst reduce-trail-to*$_{NOT}$*.simps*, *subst reduce-trail-to.simps*)
by *auto*

lemma *reduce-trail-to-map*[*simp*]:
reduce-trail-to (*map f M*) *S* = *reduce-trail-to M S*
by (*rule reduce-trail-to-length*) *simp*

lemma *reduce-trail-to*$_{NOT}$*-map*[*simp*]:
reduce-trail-to$_{NOT}$ (*map f M*) *S* = *reduce-trail-to*$_{NOT}$ *M S*
by (*rule reduce-trail-to*$_{NOT}$*-length*) *simp*

lemma *skip-or-resolve-state-change*:
assumes *skip-or-resolve*** *S T*
shows
$\exists$ *M*. *trail S* = *M* @ *trail T* $\wedge$ ($\forall$ *m* $\in$ *set M*. $\neg$*is-decided m*)
clauses S = *clauses T*
backtrack-lvl S = *backtrack-lvl T*
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
case *1* **show** *?case* **by** *simp*
case *2* **show** *?case* **by** *simp*
case *3* **show** *?case* **by** *simp*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *s-o-r* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *IH'* = *this*(*3*−*5*)

case *2* **show** *?case* **using** *IH' s-o-r* **by** (*auto elim*!: *rulesE simp*: *skip-or-resolve.simps*)
case *3* **show** *?case* **using** *IH' s-o-r* **by** (*auto elim*!: *rulesE simp*: *skip-or-resolve.simps*)
case *1* **show** *?case*
using *s-o-r*
proof *cases*
case *s-or-r-skip*
then show *?thesis* **using** *IH* **by** (*auto elim*!: *rulesE simp*: *skip-or-resolve.simps*)
next
case *s-or-r-resolve*
then show *?thesis*
using *IH* **by** (*cases trail T*) (*auto elim*!: *rulesE simp*: *skip-or-resolve.simps*)
qed
qed

6.3.3 Inclusion of Weidenbach's CDCL in NOT's CDCL

This lemma shows the inclusion of Weidenbach's CDCL *$cdcl_W$-merge* (with merging) in NOT's *$cdcl_{NOT}$-merged-bj-learn*.

```
lemma cdcl_W-merge-is-cdcl_NOT-merged-bj-learn:
  assumes
    inv: cdcl_W-all-struct-inv S and
    cdcl_W: cdcl_W-merge S T
  shows cdcl_NOT-merged-bj-learn S T
    ∨ (no-step cdcl_W-merge T ∧ conflicting T ≠ None)
  using cdcl_W inv
proof induction
  case (fw-propagate S T) note propa = this(1)
  then obtain M N U k L C where
    H: state S = (M, N, U, k, None) and
    CL: C + {#L#} ∈# clauses S and
    M-C: M ⊨as CNot C and
    undef: undefined-lit (trail S) L and
    T: state T = (Propagated L (C + {#L#}) # M, N, U, k, None)
    by (auto elim: propagate-high-levelE)
  have propagate_NOT S T
    using H CL T undef M-C by (auto simp: state-eq_NOT-def state-eq-def clauses-def
      simp del: state-simp)
  then show ?case
    using cdcl_NOT-merged-bj-learn.intros(2) by blast
next
  case (fw-decide S T) note dec = this(1) and inv = this(2)
  then obtain L where
    undef-L: undefined-lit (trail S) L and
    atm-L: atm-of L ∈ atms-of-mm (init-clss S) and
    T: T ∼ cons-trail (Decided L)
      (update-backtrack-lvl (Suc (backtrack-lvl S)) S)
    by (auto elim: decideE)
  have decide_NOT S T
    apply (rule decide_NOT.decide_NOT)
       using undef-L apply simp
     using atm-L inv unfolding cdcl_W-all-struct-inv-def no-strange-atm-def clauses-def
       apply auto[]
    using T undef-L unfolding state-eq-def state-eq_NOT-def by (auto simp: clauses-def)
  then show ?case using cdcl_NOT-merged-bj-learn-decide_NOT by blast
next
  case (fw-forget S T) note rf = this(1) and inv = this(2)
  then obtain C where
     S: conflicting S = None and
     C-le: C ∈# learned-clss S and
     ¬(trail S) ⊨asm clauses S and
     C ∉ set (get-all-mark-of-propagated (trail S)) and
     C-init: C ∉# init-clss S and
     T: T ∼ remove-cls C S
    by (auto elim: forgetE)
  have init-clss S ⊨pm C
    using inv C-le unfolding cdcl_W-all-struct-inv-def cdcl_W-learned-clause-def clauses-def
    by (meson true-clss-clss-in-imp-true-clss-cls)
  then have S-C: removeAll-mset C (clauses S) ⊨pm C
    using C-init C-le unfolding clauses-def by (auto simp add: Un-Diff ac-simps)
```

have *forget*$_{NOT}$ *S T*
apply (*rule forget*$_{NOT}$*.forget*$_{NOT}$)
using *S-C* **apply** *blast*
using *S* **apply** *simp*
using *C-init C-le* **apply** (*simp add*: *clauses-def*)
using *T C-le C-init* **by** (*auto*
simp: *state-eq-def Un-Diff state-eq*$_{NOT}$*-def clauses-def ac-simps*
simp del: *state-simp*)
then show *?case* **using** *cdcl*$_{NOT}$*-merged-bj-learn-forget*$_{NOT}$ **by** *blast*
next
case (*fw-conflict S T U*) **note** *confl* = *this*(*1*) **and** *bj* = *this*(*2*) **and** *inv* = *this*(*3*)
obtain C_S *CT* **where**
confl-T: *conflicting T* = *Some CT* **and**
CT: *CT* = C_S **and**
C_S: C_S ∈# *clauses S* **and**
tr-S-C_S: *trail S* ⊨*as CNot* C_S
using *confl* **by** (*elim conflictE*) (*auto simp del*: *state-simp simp*: *state-eq-def*)
have *cdcl*$_W$*-all-struct-inv T*
using *cdcl*$_W$*.simps cdcl*$_W$*-all-struct-inv-inv confl inv* **by** *blast*
then have *cdcl*$_W$*-M-level-inv T*
unfolding *cdcl*$_W$*-all-struct-inv-def* **by** *auto*
then consider
(*no-bt*) *skip-or-resolve*** *T U*
| (*bt*) *T'* **where** *skip-or-resolve*** *T T'* **and** *backtrack T' U*
using *bj rtranclp-cdcl*$_W$*-bj-skip-or-resolve-backtrack* **unfolding** *full-def* **by** *meson*
then show *?case*
proof *cases*
case *no-bt*
then have *conflicting U* ≠ *None*
using *confl* **by** (*induction rule*: *rtranclp-induct*)
(*auto simp del*: *state-simp simp*: *skip-or-resolve.simps state-eq-def elim*!: *rulesE*)
moreover then have *no-step cdcl*$_W$*-merge U*
by (*auto simp*: *cdcl*$_W$*-merge.simps elim*: *rulesE*)
ultimately show *?thesis* **by** *blast*
next
case *bt* **note** *s-or-r* = *this*(*1*) **and** *bt* = *this*(*2*)
have *cdcl*$_W$** *T T'*
using *s-or-r mono-rtranclp*[*of skip-or-resolve cdcl*$_W$] *rtranclp-skip-or-resolve-rtranclp-cdcl*$_W$
by *blast*
then have *cdcl*$_W$*-M-level-inv T'*
using *rtranclp-cdcl*$_W$*-consistent-inv* ‹*cdcl*$_W$*-M-level-inv T*› **by** *blast*
then obtain *M1 M2 i D L K* **where**
confl-T': *conflicting T'* = *Some D* **and**
LD: *L* ∈# *D* **and**
M1-M2:(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail T'*)) **and**
get-level (*trail T'*) *K* = *i*+*1*
get-level (*trail T'*) *L* = *backtrack-lvl T'* **and**
get-level (*trail T'*) *L* = *get-maximum-level* (*trail T'*) *D* **and**
get-maximum-level (*trail T'*) (*remove1-mset L D*) = *i* **and**
U: *U* ∼ *cons-trail* (*Propagated L D*)
(*reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None T'*))))
using *bt* **by** (*auto elim*: *backtrackE*)
have [*simp*]: *clauses S* = *clauses T*

using *confl* **by** (*auto elim*: *rulesE*)
have [*simp*]: *clauses T* = *clauses T'*
using *s-or-r*
proof (*induction*)
case *base*
then show *?case* **by** *simp*
next
case (*step U V*) **note** *st* = *this*(*1*) **and** *s-o-r* = *this*(*2*) **and** *IH* = *this*(*3*)
have *clauses U* = *clauses V*
using *s-o-r* **by** (*auto simp*: *skip-or-resolve.simps elim*: *rulesE*)
then show *?case* **using** *IH* **by** *auto*
qed
have *inv-T*: *cdcl*$_W$*-all-struct-inv T*
by (*meson cdcl*$_W$*-cp.simps confl inv r-into-rtranclp rtranclp-cdcl*$_W$*-all-struct-inv-inv*
rtranclp-cdcl$_W$*-cp-rtranclp-cdcl*$_W$)
have *cdcl*$_W$** *T T'*
using *rtranclp-skip-or-resolve-rtranclp-cdcl*$_W$ *s-or-r* **by** *blast*
have *inv-T'*: *cdcl*$_W$*-all-struct-inv T'*
using ‹*cdcl*$_W$** *T T'*› *inv-T rtranclp-cdcl*$_W$*-all-struct-inv-inv* **by** *blast*
have *inv-U*: *cdcl*$_W$*-all-struct-inv U*
using *cdcl*$_W$*-merge-restart-cdcl*$_W$ *confl fw-r-conflict inv local.bj*
rtranclp-cdcl$_W$*-all-struct-inv-inv* **by** *blast*

have [*simp*]: *init-clss S* = *init-clss T'*
using ‹*cdcl*$_W$** *T T'*› *cdcl*$_W$*-init-clss confl cdcl*$_W$*-all-struct-inv-def conflict inv*
by (*metis* ‹*cdcl*$_W$*-M-level-inv T*› *rtranclp-cdcl*$_W$*-init-clss*)
then have *atm-L*: *atm-of L* ∈ *atms-of-mm* (*clauses S*)
using *inv-T' confl-T' LD* **unfolding** *cdcl*$_W$*-all-struct-inv-def no-strange-atm-def*
clauses-def
by (*simp add*: *atms-of-def image-subset-iff*)
obtain *M* **where** *tr-T*: *trail T* = *M* @ *trail T'*
using *s-or-r skip-or-resolve-state-change* **by** *meson*
obtain *M'* **where**
tr-T': *trail T'* = *M'* @ *Decided K* # *tl* (*trail U*) **and**
tr-U: *trail U* = *Propagated L D* # *tl* (*trail U*)
using *U M1-M2 inv-T'* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def*
by *fastforce*
def *M''* ≡ *M* @ *M'*
have *tr-T*: *trail S* = *M''* @ *Decided K* # *tl* (*trail U*)
using *tr-T tr-T' confl* **unfolding** *M''-def* **by** (*auto elim*: *rulesE*)
have *init-clss T'* + *learned-clss S* ⊨*pm D*
using *inv-T' confl-T'* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-learned-clause-def*
clauses-def **by** *simp*
have *reduce-trail-to* (*convert-trail-from-NOT* (*convert-trail-from-W M1*)) *S* =
reduce-trail-to M1 S
by (*rule reduce-trail-to-length*) *simp*
moreover have *trail* (*reduce-trail-to M1 S*) = *M1*
apply (*rule reduce-trail-to-skip-beginning*[*of - M* @ *-* @ *M2* @ [*Decided K*]])
using *confl M1-M2* ‹*trail T* = *M* @ *trail T'*›
apply (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
elim!: *conflictE*)
by (*rule sym*) *auto*
ultimately have [*simp*]: *trail* (*reduce-trail-to*$_{NOT}$ *M1 S*) = *M1*
using *M1-M2 confl* **by** (*subst reduce-trail-to*$_{NOT}$*-reduce-trail-convert*)
(*auto simp*: *comp-def elim*: *rulesE*)
have *every-mark-is-a-conflict U*

using *inv-U* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def* **by** *simp*
then have *U-D*: *tl* (*trail U*) $\models$*as CNot* (*remove1-mset L D*)
by (*metis append-self-conv2 tr-U*)
have *undef-L*: *undefined-lit* (*tl* (*trail U*)) *L*
using *U M1-M2 inv-U* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def*
by (*auto simp*: *lits-of-def defined-lit-map*)
have *backjump-l S U*
apply (*rule backjump-l*[*of - - - - - L D - remove1-mset L D*])
using *tr-T* **apply** *simp*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def*
apply (*simp add*: *comp-def*)
using *U M1-M2 confl M1-M2 inv-T′ inv* **unfolding** $cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-M-level-inv-def* **apply** (*auto simp*: *state-eq*$_{NOT}$*-def*
trail-reduce-trail-to$_{NOT}$*-add-learned-cls*)[]
using C_S **apply** *auto*[]
using *tr-S-*C_S **apply** *simp*

using *undef-L* **apply** *auto*[]
using *atm-L* **apply** (*simp add*: *trail-reduce-trail-to*$_{NOT}$*-add-learned-cls*)
using ‹*init-clss T′* + *learned-clss S* $\models$*pm D*› *LD* **unfolding** *clauses-def*
apply *simp*
using *LD* **apply** *simp*
apply (*metis U-D convert-trail-from-W-true-annots*)
using *inv-T′ inv-U U confl-T′ undef-L M1-M2 LD* **unfolding** $cdcl_W$*-all-struct-inv-def*
distinct-$cdcl_W$*-state-def* **by** (*simp add*: $cdcl_W$*-M-level-inv-decomp backjump-l-cond-def*)
then show *?thesis* **using** $cdcl_{NOT}$*-merged-bj-learn-backjump-l* **by** *fast*
qed
qed

abbreviation $cdcl_{NOT}$*-restart* **where**
$cdcl_{NOT}$*-restart* $\equiv$ *restart-ops.*$cdcl_{NOT}$*-raw-restart* $cdcl_{NOT}$ *restart*

lemma $cdcl_W$*-merge-restart-is-*$cdcl_{NOT}$*-merged-bj-learn-restart-no-step*:
assumes
inv: $cdcl_W$*-all-struct-inv S* **and**
$cdcl_W$:$cdcl_W$*-merge-restart S T*
shows $cdcl_{NOT}$*-restart*** *S T* $\vee$ (*no-step* $cdcl_W$*-merge T* $\wedge$ *conflicting T* $\neq$ *None*)
proof −
consider
(*fw*) $cdcl_W$*-merge S T*
| (*fw-r*) *restart S T*
using $cdcl_W$ **by** (*meson* $cdcl_W$*-merge-restart.simps* $cdcl_W$*-rf.cases fw-conflict fw-decide fw-forget*
fw-propagate)
then show *?thesis*
proof *cases*
case *fw*
then have *IH*: $cdcl_{NOT}$*-merged-bj-learn S T* $\vee$ (*no-step* $cdcl_W$*-merge T* $\wedge$ *conflicting T* $\neq$ *None*)
using *inv* $cdcl_W$*-merge-is-*$cdcl_{NOT}$*-merged-bj-learn* **by** *blast*
have *invS*: *inv*$_{NOT}$ *S*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
have *ff2*: $cdcl_{NOT}$$^{++}$ *S T* $\longrightarrow$ $cdcl_{NOT}$** *S T*
by (*meson tranclp-into-rtranclp*)
have *ff3*: *no-dup* (*convert-trail-from-W* (*trail S*))
using *invS* **by** (*simp add*: *comp-def*)
have $cdcl_{NOT}$ $\leq$ $cdcl_{NOT}$*-restart*
by (*auto simp*: *restart-ops.*$cdcl_{NOT}$*-raw-restart.simps*)

then show *?thesis*
  using *ff3 ff2 IH* $cdcl_{NOT}$*-merged-bj-learn-is-tranclp-*$cdcl_{NOT}$
  rtranclp-mono[*of* $cdcl_{NOT}$ $cdcl_{NOT}$*-restart*] *invS predicate2D* **by** *blast*
next
  case *fw-r*
  then show *?thesis* **by** (*blast intro*: *restart-ops.*$cdcl_{NOT}$*-raw-restart.intros*)
qed
qed

abbreviation μ_{FW} :: *'st* $\Rightarrow$ *nat* **where**
μ_{FW} $S \equiv$ (*if no-step* $cdcl_W$*-merge* S *then 0 else 1+*μ_{CDCL}*'-merged* (*set-mset* (*init-clss* S)) S)

lemma $cdcl_W$*-merge-*μ_{FW}*-decreasing*:
assumes
inv: $cdcl_W$*-all-struct-inv* S **and**
fw: $cdcl_W$*-merge* S T
shows μ_{FW} $T < \mu_{FW}$ S
proof −
let *?A* = *init-clss* S
have *atm-clauses*: *atms-of-mm* (*clauses* S) $\subseteq$ *atms-of-mm ?A*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def no-strange-atm-def clauses-def* **by** *auto*
have *atm-trail*: *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-mm ?A*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def no-strange-atm-def clauses-def* **by** *auto*
have *n-d*: *no-dup* (*trail* S)
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
have [*simp*]: ¬ *no-step* $cdcl_W$*-merge* S
using *fw* **by** *auto*
have [*simp*]: *init-clss* S = *init-clss* T
using $cdcl_W$*-merge-restart-*$cdcl_W$[*of* S T] *inv rtranclp-*$cdcl_W$*-init-clss*
unfolding $cdcl_W$*-all-struct-inv-def*
by (*meson* $cdcl_W$*-merge.simps* $cdcl_W$*-merge-restart.simps* $cdcl_W$*-rf.simps fw*)
consider
(*merged*) $cdcl_{NOT}$*-merged-bj-learn* S T
| (*n-s*) *no-step* $cdcl_W$*-merge* T
using $cdcl_W$*-merge-is-*$cdcl_{NOT}$*-merged-bj-learn inv fw* **by** *blast*
then show *?thesis*
proof *cases*
case *merged*
then show *?thesis*
using $cdcl_{NOT}$*-decreasing-measure'*[*OF - - atm-clauses*, *of* T] *atm-trail n-d*
by (*auto split*: *if-split simp*: *comp-def image-image lits-of-def*)
next
case *n-s*
then show *?thesis* **by** *simp*
qed
qed

lemma *wf-*$cdcl_W$*-merge*: *wf* {(T, S). $cdcl_W$*-all-struct-inv* S $\wedge$ $cdcl_W$*-merge* S T}
apply (*rule wfP-if-measure*[*of - -* μ_{FW}])
using $cdcl_W$*-merge-*μ_{FW}*-decreasing* **by** *blast*

sublocale *conflict-driven-clause-learning*$_W$*-termination*
by *unfold-locales* (*simp add*: *wf-*$cdcl_W$*-merge*)

6.3.4 Correctness of $cdcl_W$-*merge-stgy*

lemma *full*-$cdcl_W$-s'-*full*-$cdcl_W$-*merge-restart*:
assumes
conflicting R = None **and**
inv: $cdcl_W$-*all-struct-inv R*
shows *full* $cdcl_W$-s' *R V* $\longleftrightarrow$ *full* $cdcl_W$-*merge-stgy R V* (**is** *?s'* $\longleftrightarrow$ *?fw*)
proof
assume *?s'*
then have $cdcl_W$-s'^{**} *R V* **unfolding** *full-def* **by** *blast*
have $cdcl_W$-*all-struct-inv V*
using ‹$cdcl_W$-s'^{**} *R V*› *inv rtranclp*-$cdcl_W$-*all-struct-inv-inv rtranclp*-$cdcl_W$-s'-*rtranclp*-$cdcl_W$
by *blast*
then have *n-s*: *no-step* $cdcl_W$-*merge-stgy V*
using *no-step*-$cdcl_W$-s'-*no-step*-$cdcl_W$-*merge-stgy* **by** (*meson* ‹*full* $cdcl_W$-s' *R V*› *full-def*)
have *n-s-bj*: *no-step* $cdcl_W$-*bj V*
by (*metis* ‹$cdcl_W$-*all-struct-inv V*› ‹*full* $cdcl_W$-s' *R V*› *bj full-def*
n-step-$cdcl_W$-*stgy-iff-no-step*-$cdcl_W$-*cl*-$cdcl_W$-*o*)
have *n-s-cp*: *no-step* $cdcl_W$-*merge-cp V*
proof −
{ fix *ss* :: *'st*
obtain *ssa* :: *'st* ⇒ *'st* **where**
ff1: ∀ *s*. ¬ $cdcl_W$-*all-struct-inv s* ∨ $cdcl_W$-s'-*without-decide s* (*ssa s*)
∨ *no-step* $cdcl_W$-*merge-cp s*
using *conflicting-true-no-step-s'-without-decide-no-step*-$cdcl_W$-*merge-cp* **by** *moura*
have ∀ *p s sa*. ¬ *full p* (*s*::*'st*) *sa* ∨ p^{**} *s sa* ∧ *no-step p sa* **and**
∀ *p s sa*. (¬ p^{**} (*s*::*'st*) *sa* ∨ (∃ *s*. *p sa s*)) ∨ *full p s sa*
by (*meson full-def*)+
then have ¬ $cdcl_W$-*merge-cp V ss*
using *ff1* **by** (*metis* (*no-types*) ‹$cdcl_W$-*all-struct-inv V*› ‹*full* $cdcl_W$-s' *R V*› $cdcl_W$-s'.*simps*
$cdcl_W$-s'-*without-decide.cases*) **}**
then show *?thesis*
by *blast*
qed
consider
(*fw-no-confl*) $cdcl_W$-*merge-stgy*** *R V* **and** *conflicting V = None*
| (*fw-confl*) $cdcl_W$-*merge-stgy*** *R V* **and** *conflicting V* ≠ *None* **and** *no-step* $cdcl_W$-*bj V*
| (*fw-dec-confl*) *S T U* **where** $cdcl_W$-*merge-stgy*** *R S* **and** *no-step* $cdcl_W$-*merge-cp S* **and**
decide S T **and** $cdcl_W$-*merge-cp*** *T U* **and** *conflict U V*
| (*fw-dec-no-confl*) *S T* **where** $cdcl_W$-*merge-stgy*** *R S* **and** *no-step* $cdcl_W$-*merge-cp S* **and**
decide S T **and** $cdcl_W$-*merge-cp*** *T V* **and** *conflicting V = None*
| (*cp-no-confl*) $cdcl_W$-*merge-cp*** *R V* **and** *conflicting V = None*
| (*cp-confl*) *U* **where** $cdcl_W$-*merge-cp*** *R U* **and** *conflict U V*
using *rtranclp*-$cdcl_W$-s'-*no-step*-$cdcl_W$-s'-*without-decide-decomp-into*-$cdcl_W$-*merge*[*OF*
‹$cdcl_W$-s'^{**} *R V*› *assms*] **by** *auto*
then show *?fw*
proof *cases*
case *fw-no-confl*
then show *?thesis* **using** *n-s* **unfolding** *full-def* **by** *blast*
next
case *fw-confl*
then show *?thesis* **using** *n-s* **unfolding** *full-def* **by** *blast*
next
case *fw-dec-confl*
have $cdcl_W$-*merge-cp U V*
using *n-s-bj* **by** (*metis* $cdcl_W$-*merge-cp.simps full-unfold fw-dec-confl*(*5*))

then have *full1* $cdcl_W$*-merge-cp* T V
unfolding *full1-def* **by** (*metis fw-dec-confl*(*4*) *n-s-cp tranclp-unfold-end*)
then have $cdcl_W$*-merge-stgy* S V **using** ⟨*decide* S T⟩ ⟨*no-step* $cdcl_W$*-merge-cp* S⟩ **by** *auto*
then show *?thesis* **using** *n-s* ⟨ $cdcl_W$*-merge-stgy*** R S⟩ **unfolding** *full-def* **by** *auto*
next
case *fw-dec-no-confl*
then have *full* $cdcl_W$*-merge-cp* T V
using *n-s-cp* **unfolding** *full-def* **by** *blast*
then have $cdcl_W$*-merge-stgy* S V **using** ⟨*decide* S T⟩ ⟨*no-step* $cdcl_W$*-merge-cp* S⟩ **by** *auto*
then show *?thesis* **using** *n-s* ⟨ $cdcl_W$*-merge-stgy*** R S⟩ **unfolding** *full-def* **by** *auto*
next
case *cp-no-confl*
then have *full* $cdcl_W$*-merge-cp* R V
by (*simp add*: *full-def n-s-cp*)
then have $R = V \vee$ $cdcl_W$*-merge-stgy*$^{++}$ R V
using *fw-s-cp* **unfolding** *full-unfold fw-s-cp*
by (*metis* (*no-types*) *rtranclp-unfold tranclp-unfold-end*)
then show *?thesis*
by (*simp add*: *full-def n-s rtranclp-unfold*)
next
case *cp-confl*
have *full* $cdcl_W$*-bj* V V
using *n-s-bj* **unfolding** *full-def* **by** *blast*
then have *full1* $cdcl_W$*-merge-cp* R V
unfolding *full1-def* **by** (*meson* $cdcl_W$*-merge-cp.conflict*$'$ *cp-confl*(*1*,*2*) *n-s-cp*
rtranclp-into-tranclp1)
then show *?thesis* **using** *n-s* **unfolding** *full-def* **by** *auto*
qed
next
assume *?fw*
then have $cdcl_W{}^{**}$ R V **using** *rtranclp-mono*[*of* $cdcl_W$*-merge-stgy* $cdcl_W{}^{**}$]
$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$ **unfolding** *full-def* **by** *auto*
then have *inv*$'$: $cdcl_W$*-all-struct-inv* V **using** *inv rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
have $cdcl_W$*-s*$'^{**}$ R V
using ⟨*?fw*⟩ **by** (*simp add*: *full-def inv rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$*-s*$'$)
moreover have *no-step* $cdcl_W$*-s*$'$ V
proof *cases*
assume *conflicting* V = *None*
then show *?thesis*
by (*metis inv*$'$ ⟨*full* $cdcl_W$*-merge-stgy* R V⟩ *full-def*
no-step-$cdcl_W$*-merge-stgy-no-step-*$cdcl_W$*-s*$'$)
next
assume *confl-V*: *conflicting* $V \neq$ *None*
then have *no-step* $cdcl_W$*-bj* V
using *rtranclp-*$cdcl_W$*-merge-stgy-no-step-*$cdcl_W$*-bj* **by** (*meson* ⟨*full* $cdcl_W$*-merge-stgy* R V⟩
assms(*1*) *full-def*)
then show *?thesis* **using** *confl-V* **by** (*fastforce simp*: $cdcl_W$*-s*$'$*.simps full1-def* $cdcl_W$*-cp.simps*
dest!: *tranclpD elim*: *rulesE*)
qed
ultimately show *?s*$'$ **unfolding** *full-def* **by** *blast*
qed

lemma *full-*$cdcl_W$*-stgy-full-*$cdcl_W$*-merge*:
assumes
conflicting R = *None* **and**
$cdcl_W$*-all-struct-inv* R

shows *full* $cdcl_W$*-stgy R V* $\longleftrightarrow$ *full* $cdcl_W$*-merge-stgy R V*
by (*simp add*: *assms full*-$cdcl_W$-s'-*full*-$cdcl_W$-*merge-restart full*-$cdcl_W$-*stgy-iff-full*-$cdcl_W$-s')

lemma *full*-$cdcl_W$-*merge-stgy-final-state-conclusive*$'$:
fixes S' :: $'st$
assumes
full: *full* $cdcl_W$-*merge-stgy* (*init-state N*) S' **and**
no-d: *distinct-mset-mset N*
shows (*conflicting* S' = *Some* {#} ∧ *unsatisfiable* (*set-mset N*))
∨ (*conflicting* S' = *None* ∧ *trail* S' ⊨*asm N* ∧ *satisfiable* (*set-mset N*))
proof −
have $cdcl_W$-*all-struct-inv* (*init-state N*)
using *no-d* **unfolding** $cdcl_W$-*all-struct-inv-def* **by** *auto*
moreover have *conflicting* (*init-state N*) = *None*
by *auto*
ultimately show *?thesis*
using *full full*-$cdcl_W$-*stgy-final-state-conclusive-from-init-state*
full-$cdcl_W$-*stgy-full*-$cdcl_W$-*merge no-d* **by** *presburger*
qed
end

end
theory *CDCL-W-Restart*
imports *CDCL-W-Merge*
begin

6.3.5 Adding Restarts

locale $cdcl_W$-*restart* =
conflict-driven-clause-learning$_W$
— functions for the state:
— access functions:
trail init-clss learned-clss backtrack-lvl conflicting
— changing state:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state
for
trail :: $'st \Rightarrow ('v, 'v\ clause)$ *ann-lits* **and**
init-clss :: $'st \Rightarrow 'v$ *clauses* **and**
learned-clss :: $'st \Rightarrow 'v$ *clauses* **and**
backtrack-lvl :: $'st \Rightarrow nat$ **and**
conflicting :: $'st \Rightarrow 'v$ *clause option* **and**

cons-trail :: $('v, 'v\ clause)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-learned-cls :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls :: $'v$ *clause* $\Rightarrow 'st \Rightarrow 'st$ **and**
update-backtrack-lvl :: $nat \Rightarrow 'st \Rightarrow 'st$ **and**
update-conflicting :: $'v$ *clause option* $\Rightarrow 'st \Rightarrow 'st$ **and**

init-state :: $'v$ *clauses* $\Rightarrow 'st$ +
fixes f :: $nat \Rightarrow nat$
assumes f: *unbounded f*

begin

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive *cdcl*$_W$*-merge-with-restart* **where**
restart-step:
$(cdcl_W\text{-}merge\text{-}stgy^{\frown\frown}(card\ (set\text{-}mset\ (learned\text{-}clss\ T)) - card\ (set\text{-}mset\ (learned\text{-}clss\ S))))\ S\ T$
$\Longrightarrow card\ (set\text{-}mset\ (learned\text{-}clss\ T)) - card\ (set\text{-}mset\ (learned\text{-}clss\ S)) > f\ n$
$\Longrightarrow restart\ T\ U \Longrightarrow cdcl_W\text{-}merge\text{-}with\text{-}restart\ (S, n)\ (U, Suc\ n)$ |
restart-full: $full1\ cdcl_W\text{-}merge\text{-}stgy\ S\ T \Longrightarrow cdcl_W\text{-}merge\text{-}with\text{-}restart\ (S, n)\ (T, Suc\ n)$

lemma $cdcl_W\text{-}merge\text{-}with\text{-}restart\ S\ T \Longrightarrow cdcl_W\text{-}merge\text{-}restart^{**}\ (fst\ S)\ (fst\ T)$
by (*induction rule*: *cdcl*$_W$*-merge-with-restart.induct*)
(*auto dest*!: *relpowp-imp-rtranclp cdcl*$_W$*-merge-stgy-tranclp-cdcl*$_W$*-merge tranclp-into-rtranclp rtranclp-cdcl*$_W$*-merge-stgy-rtranclp-cdcl*$_W$*-merge rtranclp-cdcl*$_W$*-merge-tranclp-cdcl*$_W$*-merge-restart fw-r-rf cdcl*$_W$*-rf.restart*
simp: *full1-def*)

lemma *cdcl*$_W$*-merge-with-restart-rtranclp-cdcl*$_W$:
$cdcl_W\text{-}merge\text{-}with\text{-}restart\ S\ T \Longrightarrow cdcl_W^{**}\ (fst\ S)\ (fst\ T)$
by (*induction rule*: *cdcl*$_W$*-merge-with-restart.induct*)
(*auto dest*!: *relpowp-imp-rtranclp rtranclp-cdcl*$_W$*-merge-stgy-rtranclp-cdcl*$_W$ *cdcl*$_W$*.rf*
cdcl$_W$*-rf.restart tranclp-into-rtranclp simp*: *full1-def*)

lemma *cdcl*$_W$*-merge-with-restart-increasing-number*:
$cdcl_W\text{-}merge\text{-}with\text{-}restart\ S\ T \Longrightarrow snd\ T = 1 + snd\ S$
by (*induction rule*: *cdcl*$_W$*-merge-with-restart.induct*) *auto*

lemma $full1\ cdcl_W\text{-}merge\text{-}stgy\ S\ T \Longrightarrow cdcl_W\text{-}merge\text{-}with\text{-}restart\ (S, n)\ (T, Suc\ n)$
using *restart-full* **by** *blast*

lemma *cdcl*$_W$*-all-struct-inv-learned-clss-bound*:
assumes *inv*: *cdcl*$_W$*-all-struct-inv S*
shows $set\text{-}mset\ (learned\text{-}clss\ S) \subseteq simple\text{-}clss\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S))$
proof
fix *C*
assume *C*: $C \in set\text{-}mset\ (learned\text{-}clss\ S)$
have *distinct-mset C*
using *C inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def distinct-cdcl*$_W$*-state-def distinct-mset-set-def*
by *auto*
moreover have $\neg tautology\ C$
using *C inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-learned-clause-def* **by** *auto*
moreover
have $atms\text{-}of\ C \subseteq atms\text{-}of\text{-}mm\ (learned\text{-}clss\ S)$
using *C* **by** *auto*
then have $atms\text{-}of\ C \subseteq atms\text{-}of\text{-}mm\ (init\text{-}clss\ S)$
using *inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def no-strange-atm-def* **by** *force*
moreover have *finite* (*atms-of-mm* (*init-clss S*))
using *inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def* **by** *auto*
ultimately show $C \in simple\text{-}clss\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S))$
using *distinct-mset-not-tautology-implies-in-simple-clss simple-clss-mono*
by *blast*
qed

lemma cdcl$_W$-merge-with-restart-init-clss:
  cdcl$_W$-merge-with-restart S T $\Longrightarrow$ cdcl$_W$-M-level-inv (fst S) $\Longrightarrow$
  init-clss (fst S) = init-clss (fst T)
  using cdcl$_W$-merge-with-restart-rtranclp-cdcl$_W$ rtranclp-cdcl$_W$-init-clss **by** blast

lemma
  wf {(T, S). cdcl$_W$-all-struct-inv (fst S) $\wedge$ cdcl$_W$-merge-with-restart S T}
proof (rule ccontr)
  assume $\neg$?thesis
    then obtain g **where**
    g: $\bigwedge$i. cdcl$_W$-merge-with-restart (g i) (g (Suc i)) **and**
    inv: $\bigwedge$i. cdcl$_W$-all-struct-inv (fst (g i))
    unfolding wf-iff-no-infinite-down-chain **by** fast
  { fix i
    have init-clss (fst (g i)) = init-clss (fst (g 0))
      apply (induction i)
        apply simp
      using g inv **unfolding** cdcl$_W$-all-struct-inv-def **by** (metis cdcl$_W$-merge-with-restart-init-clss)
    } note init-g = this
  let ?S = g 0
  have finite (atms-of-mm (init-clss (fst ?S)))
    using inv **unfolding** cdcl$_W$-all-struct-inv-def **by** auto
  have snd-g: $\bigwedge$i. snd (g i) = i + snd (g 0)
    apply (induct-tac i)
      apply simp
    by (metis Suc-eq-plus1-left add-Suc cdcl$_W$-merge-with-restart-increasing-number g)
  then have snd-g-0: $\bigwedge$i. i > 0 $\Longrightarrow$ snd (g i) = i + snd (g 0)
    by blast
  have unbounded-f-g: unbounded (λi. f (snd (g i)))
    using f **unfolding** bounded-def **by** (metis add.commute f less-or-eq-imp-le snd-g
      not-bounded-nat-exists-larger not-le le-iff-add)

  obtain k **where**
    f-g-k: f (snd (g k)) > card (simple-clss (atms-of-mm (init-clss (fst ?S)))) **and**
    k > card (simple-clss (atms-of-mm (init-clss (fst ?S))))
    using not-bounded-nat-exists-larger[OF unbounded-f-g] **by** blast

The following does not hold anymore with the non-strict version of cardinality in the definition.

  { fix i
    assume no-step cdcl$_W$-merge-stgy (fst (g i))
    with g[of i]
    have False
      proof (induction rule: cdcl$_W$-merge-with-restart.induct)
        case (restart-step T S n) **note** H = this(1) **and** c = this(2) **and** n-s = this(4)
        obtain S' **where** cdcl$_W$-merge-stgy S S'
          using H c **by** (metis gr-implies-not0 relpowp-E2)
        then show False **using** n-s **by** auto
      next
        case (restart-full S T)
        then show False **unfolding** full1-def **by** (auto dest: tranclpD)
      qed
    } note H = this
  obtain m T **where**
    m: m = card (set-mset (learned-clss T)) − card (set-mset (learned-clss (fst (g k)))) **and**
    m > f (snd (g k)) **and**
    restart T (fst (g (k+1))) **and**

$cdcl_W$*-merge-stgy*: ($cdcl_W$*-merge-stgy* ^^ *m*) (*fst* (*g k*)) *T*
using *g*[*of k*] *H*[*of Suc k*] **by** (*force simp*: $cdcl_W$*-merge-with-restart.simps full1-def*)
have $cdcl_W$*-merge-stgy*** (*fst* (*g k*)) *T*
using $cdcl_W$*-merge-stgy relpowp-imp-rtranclp* **by** *metis*
then have $cdcl_W$*-all-struct-inv T*
using *inv*[*of k*] *rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$
by *blast*
moreover have *card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss* (*fst* (*g k*))))
> *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*fst ?S*))))
unfolding *m*[*symmetric*] **using** ‹*m* > *f* (*snd* (*g k*))› *f-g-k* **by** *linarith*
then have *card* (*set-mset* (*learned-clss T*))
> *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*fst ?S*))))
by *linarith*
moreover
have *init-clss* (*fst* (*g k*)) = *init-clss T*
using ‹$cdcl_W$*-merge-stgy*** (*fst* (*g k*)) *T*› *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$
rtranclp-$cdcl_W$*-init-clss inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *blast*
then have *init-clss* (*fst ?S*) = *init-clss T*
using *init-g*[*of k*] **by** *auto*
ultimately show *False*
using $cdcl_W$*-all-struct-inv-learned-clss-bound*
by (*simp add*: ‹*finite* (*atms-of-mm* (*init-clss* (*fst* (*g 0*))))› *simple-clss-finite*
card-mono leD)
qed

lemma $cdcl_W$*-merge-with-restart-distinct-mset-clauses*:
assumes *invR*: $cdcl_W$*-all-struct-inv* (*fst R*) **and**
st: $cdcl_W$*-merge-with-restart R S* **and**
dist: *distinct-mset* (*clauses* (*fst R*)) **and**
R: *trail* (*fst R*) = []
shows *distinct-mset* (*clauses* (*fst S*))
using *assms*(*2*,*1*,*3*,*4*)
proof (*induction*)
case (*restart-full S T*)
then show *?case* **using** *rtranclp-*$cdcl_W$*-merge-stgy-distinct-mset-clauses*[*of S T*] **unfolding** *full1-def*
by (*auto dest*: *tranclp-into-rtranclp*)
next
case (*restart-step T S n U*)
then have *distinct-mset* (*clauses T*)
using *rtranclp-*$cdcl_W$*-merge-stgy-distinct-mset-clauses*[*of S T*] **unfolding** *full1-def*
by (*auto dest*: *relpowp-imp-rtranclp*)
then show *?case* **using** ‹*restart T U*› **unfolding** *clauses-def*
by (*metis distinct-mset-union fstI restartE subset-mset.le-iff-add union-assoc*)
qed

inductive $cdcl_W$*-with-restart* **where**
restart-step:
($cdcl_W$*-stgy*^^(*card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)))) *S T* ⟹
card (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)) > *f n* ⟹
restart T U ⟹
$cdcl_W$*-with-restart* (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1* $cdcl_W$*-stgy S T* ⟹ $cdcl_W$*-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma $cdcl_W$*-with-restart-rtranclp-*$cdcl_W$:
$cdcl_W$*-with-restart S T* ⟹ $cdcl_W$** (*fst S*) (*fst T*)
apply (*induction rule*: $cdcl_W$*-with-restart.induct*)

by (*auto dest*!: *relpowp-imp-rtranclp* *tranclp-into-rtranclp fw-r-rf*
$cdcl_W$*-rf.restart rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ $cdcl_W$*-merge-restart-*$cdcl_W$
simp: *full1-def*)

lemma $cdcl_W$*-with-restart-increasing-number*:
$cdcl_W$*-with-restart S T* $\Longrightarrow$ *snd T* = *1* + *snd S*
by (*induction rule*: $cdcl_W$*-with-restart.induct*) *auto*

lemma *full1* $cdcl_W$*-stgy S T* $\Longrightarrow$ $cdcl_W$*-with-restart* (*S*, *n*) (*T*, *Suc n*)
using *restart-full* **by** *blast*

lemma $cdcl_W$*-with-restart-init-clss*:
$cdcl_W$*-with-restart S T* $\Longrightarrow$ $cdcl_W$*-M-level-inv* (*fst S*) $\Longrightarrow$ *init-clss* (*fst S*) = *init-clss* (*fst T*)
using $cdcl_W$*-with-restart-rtranclp-*$cdcl_W$ *rtranclp-*$cdcl_W$*-init-clss* **by** *blast*

lemma
wf {(*T*, *S*). $cdcl_W$*-all-struct-inv* (*fst S*) ∧ $cdcl_W$*-with-restart S T*}
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *g* **where**
g: $\bigwedge$*i*. $cdcl_W$*-with-restart* (*g i*) (*g* (*Suc i*)) **and**
inv: $\bigwedge$*i*. $cdcl_W$*-all-struct-inv* (*fst* (*g i*))
unfolding *wf-iff-no-infinite-down-chain* **by** *fast*
{ fix *i*
have *init-clss* (*fst* (*g i*)) = *init-clss* (*fst* (*g 0*))
apply (*induction i*)
apply *simp*
using *g inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** (*metis* $cdcl_W$*-with-restart-init-clss*)
} note *init-g* = *this*
let *?S* = *g 0*
have *finite* (*atms-of-mm* (*init-clss* (*fst ?S*)))
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *auto*
have *snd-g*: $\bigwedge$*i*. *snd* (*g i*) = *i* + *snd* (*g 0*)
apply (*induct-tac i*)
apply *simp*
by (*metis Suc-eq-plus1-left add-Suc* $cdcl_W$*-with-restart-increasing-number g*)
then have *snd-g-0*: $\bigwedge$*i*. *i* > *0* $\Longrightarrow$ *snd* (*g i*) = *i* + *snd* (*g 0*)
by *blast*
have *unbounded-f-g*: *unbounded* (λ*i*. *f* (*snd* (*g i*)))
using *f* **unfolding** *bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*
not-bounded-nat-exists-larger not-le le-iff-add)

obtain *k* **where**
f-g-k: *f* (*snd* (*g k*)) > *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*fst ?S*)))) **and**
k > *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*fst ?S*))))
using *not-bounded-nat-exists-larger*[*OF unbounded-f-g*] **by** *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

{ fix *i*
assume *no-step* $cdcl_W$*-stgy* (*fst* (*g i*))
with *g*[*of i*]
have *False*
proof (*induction rule*: $cdcl_W$*-with-restart.induct*)
case (*restart-step T S n*) **note** *H* = *this*(*1*) **and** *c* = *this*(*2*) **and** *n-s* = *this*(*4*)
obtain *S′* **where** $cdcl_W$*-stgy S S′*
using *H c* **by** (*metis gr-implies-not0 relpowp-E2*)

then show *False* **using** *n-s* **by** *auto*
next
case (*restart-full S T*)
then show *False* **unfolding** *full1-def* **by** (*auto dest*: *tranclpD*)
qed
} **note** *H* = *this*
obtain *m T* **where**
m: *m* = *card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss* (*fst* (*g k*)))) **and**
m > *f* (*snd* (*g k*)) **and**
restart T (*fst* (*g* (*k+1*))) **and**
$cdcl_W$*-merge-stgy*: ($cdcl_W$*-stgy* ^^ *m*) (*fst* (*g k*)) *T*
using *g*[*of k*] *H*[*of Suc k*] **by** (*force simp*: $cdcl_W$*-with-restart.simps full1-def*)
have $cdcl_W$*-stgy*** (*fst* (*g k*)) *T*
using $cdcl_W$*-merge-stgy relpowp-imp-rtranclp* **by** *metis*
then have $cdcl_W$*-all-struct-inv T*
using *inv*[*of k*] *rtranclp-*$cdcl_W$*-all-struct-inv-inv rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ **by** *blast*
moreover have *card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss* (*fst* (*g k*))))
> *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*fst ?S*))))
unfolding *m*[*symmetric*] **using** ‹*m* > *f* (*snd* (*g k*))› *f-g-k* **by** *linarith*
then have *card* (*set-mset* (*learned-clss T*))
> *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*fst ?S*))))
by *linarith*
moreover
have *init-clss* (*fst* (*g k*)) = *init-clss T*
using ‹$cdcl_W$*-stgy*** (*fst* (*g k*)) *T*› *rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ *rtranclp-*$cdcl_W$*-init-clss*
inv **unfolding** $cdcl_W$*-all-struct-inv-def*
by *blast*
then have *init-clss* (*fst ?S*) = *init-clss T*
using *init-g*[*of k*] **by** *auto*
ultimately show *False*
using $cdcl_W$*-all-struct-inv-learned-clss-bound*
by (*simp add*: ‹*finite* (*atms-of-mm* (*init-clss* (*fst* (*g 0*))))› *simple-clss-finite*
card-mono leD)
qed

lemma $cdcl_W$*-with-restart-distinct-mset-clauses*:
assumes *invR*: $cdcl_W$*-all-struct-inv* (*fst R*) **and**
st: $cdcl_W$*-with-restart R S* **and**
dist: *distinct-mset* (*clauses* (*fst R*)) **and**
R: *trail* (*fst R*) = []
shows *distinct-mset* (*clauses* (*fst S*))
using *assms*(*2*,*1*,*3*,*4*)
proof (*induction*)
case (*restart-full S T*)
then show *?case* **using** *rtranclp-*$cdcl_W$*-stgy-distinct-mset-clauses*[*of S T*] **unfolding** *full1-def*
by (*auto dest*: *tranclp-into-rtranclp*)
next
case (*restart-step T S n U*)
then have *distinct-mset* (*clauses T*) **using** *rtranclp-*$cdcl_W$*-stgy-distinct-mset-clauses*[*of S T*]
unfolding *full1-def* **by** (*auto dest*: *relpowp-imp-rtranclp*)
then show *?case* **using** ‹*restart T U*› **unfolding** *clauses-def*
by (*metis distinct-mset-union fstI restartE subset-mset.le-iff-add union-assoc*)
qed
end

locale *luby-sequence* =

fixes *ur* :: *nat*
assumes *ur* > *0*
begin

lemma *exists-luby-decomp*:
fixes *i* ::*nat*
shows ∃*k*::*nat*. (*2* ^ (*k* − *1*) ≤ *i* ∧ *i* < *2* ^ *k* − *1*) ∨ *i* = *2* ^ *k* − *1*
proof (*induction i*)
case *0*
then show *?case*
by (*rule exI*[*of* - *0*], *simp*)
next
case (*Suc n*)
then obtain *k* **where** *2* ^ (*k* − *1*) ≤ *n* ∧ *n* < *2* ^ *k* − *1* ∨ *n* = *2* ^ *k* − *1*
by *blast*
then consider
(*st-interv*) *2* ^ (*k* − *1*) ≤ *n* **and** *n* ≤ *2* ^ *k* − *2*
| (*end-interv*) *2* ^ (*k* − *1*) ≤ *n* **and** *n* = *2* ^ *k* − *2*
| (*pow2*) *n* = *2* ^ *k* − *1*
by *linarith*
then show *?case*
proof *cases*
case *st-interv*
then show *?thesis* **apply** − **apply** (*rule exI*[*of* - *k*])
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-diff-Suc Suc-lessI*
‹*2* ^ (*k* − *1*) ≤ *n* ∧ *n* < *2* ^ *k* − *1* ∨ *n* = *2* ^ *k* − *1*› *diff-self-eq-0*
dual-order.trans le-SucI le-imp-less-Suc numeral-2-eq-2 one-le-numeral
one-le-power zero-less-numeral zero-less-power)
next
case *end-interv*
then show *?thesis* **apply** − **apply** (*rule exI*[*of* - *k*]) **by** *auto*
next
case *pow2*
then show *?thesis* **apply** − **apply** (*rule exI*[*of* - *k*+*1*]) **by** *auto*
qed
qed

Luby sequences are defined by:

- $2^k - 1$, if $i = (2::'a)^k - (1::'a)$
- *luby-sequence-core* $(i - 2^{k - 1} + 1)$, if $(2::'a)^{k - 1} \leq i$ and $i \leq (2::'a)^k - (1::'a)$

Then the sequence is then scaled by a constant unit run (called *ur* here), strictly positive.

function *luby-sequence-core* :: *nat* ⇒ *nat* **where**
luby-sequence-core i =
(*if* ∃*k*. *i* = *2*^*k* − *1*
then 2^((*SOME k*. *i* = *2*^*k* − *1*) − *1*)
else luby-sequence-core (*i* − *2*^((*SOME k*. *2*^(*k*−*1*)≤ *i* ∧ *i* < *2*^*k* − *1*) − *1*) + *1*))
by *auto*
termination
proof (*relation less-than*, *goal-cases*)
case *1*
then show *?case* **by** *auto*
next

case (*2 i*)
let *?k = SOME k. 2 ^ (k − 1) ≤ i ∧ i < 2 ^ k − 1*
have *2 ^ (?k − 1) ≤ i ∧ i < 2 ^ ?k − 1*
apply (*rule someI-ex*)
using *2 exists-luby-decomp* **by** *blast*
then show *?case*

proof −
have ∀ *n na.* ¬ (*1::nat*) ≤ *n* ∨ *1* ≤ *n ^ na*
by (*meson one-le-power*)
then have *f1*: (*1::nat*) ≤ *2 ^ (?k − 1)*
using *one-le-numeral* **by** *blast*
have *f2*: *i − 2 ^ (?k − 1) + 2 ^ (?k − 1) = i*
using ‹*2 ^ (?k − 1) ≤ i ∧ i < 2 ^ ?k − 1*› *le-add-diff-inverse2* **by** *blast*
have *f3*: *2 ^ ?k − 1 ≠ Suc 0*
using *f1* ‹*2 ^ (?k − 1) ≤ i ∧ i < 2 ^ ?k − 1*› **by** *linarith*
have *2 ^ ?k −* (*1::nat*) ≠ *0*
using ‹*2 ^ (?k − 1) ≤ i ∧ i < 2 ^ ?k − 1*› *gr-implies-not0* **by** *blast*
then have *f4*: *2 ^ ?k* ≠ (*1::nat*)
by *linarith*
have *f5*: ∀ *n na. if na = 0 then* (*n::nat*) *^ na = 1 else n ^ na = n ∗ n ^* (*na − 1*)
by (*simp add*: *power-eq-if*)
then have *?k ≠ 0*
using *f4* **by** *meson*
then have *2 ^ (?k − 1) ≠ Suc 0*
using *f5 f3* **by** *presburger*
then have *Suc 0 < 2 ^ (?k − 1)*
using *f1* **by** *linarith*
then show *?thesis*
using *f2 less-than-iff* **by** *presburger*
qed
qed

function *natlog2* :: *nat* ⇒ *nat* **where**
natlog2 n = (*if n = 0 then 0 else 1 + natlog2* (*n div 2*))
using *not0-implies-Suc* **by** *auto*
termination by (*relation measure* (λ*n. n*)) *auto*

declare *natlog2.simps*[*simp del*]

declare *luby-sequence-core.simps*[*simp del*]

lemma *two-pover-n-eq-two-power-n′-eq*:
assumes *H*: (*2::nat*) ^ (*k::nat*) − *1* = *2 ^ k′ − 1*
shows *k′ = k*
proof −
have (*2::nat*) ^ (*k::nat*) = *2 ^ k′*
using *H* **by** (*metis One-nat-def Suc-pred zero-less-numeral zero-less-power*)
then show *?thesis* **by** *simp*
qed

lemma *luby-sequence-core-two-power-minus-one*:
luby-sequence-core (*2^k − 1*) = *2^*(*k−1*) (**is** *?L = ?K*)
proof −
have *decomp*: ∃ *ka. 2 ^ k − 1 = 2 ^ ka − 1*
by *auto*

have *?L* = *2*^((*SOME* k'. (*2*::*nat*)^*k* − *1* = *2*^k' − *1*) − *1*)
  apply (*subst luby-sequence-core.simps*, *subst decomp*)
  by *simp*
moreover have (*SOME* k'. (*2*::*nat*)^*k* − *1* = *2*^k' − *1*) = *k*
  apply (*rule some-equality*)
    apply *simp*
    using *two-pover-n-eq-two-power-n'-eq* **by** *blast*
ultimately show *?thesis* **by** *presburger*
qed

lemma *different-luby-decomposition-false*:
  assumes
    H: *2* ^ (*k* − *Suc 0*) ≤ *i* **and**
    k': *i* < *2* ^ k' − *Suc 0* **and**
    k-k': *k* > k'
  shows *False*
proof −
  have *2* ^ k' − *Suc 0* < *2* ^ (*k* − *Suc 0*)
    using *k-k'* *less-eq-Suc-le* **by** *auto*
  then show *?thesis*
    using *H* k' **by** *linarith*
qed

lemma *luby-sequence-core-not-two-power-minus-one*:
  assumes
    k-i: *2* ^ (*k* − *1*) ≤ *i* **and**
    i-k: *i* < *2*^*k* − *1*
  shows *luby-sequence-core i* = *luby-sequence-core* (*i* − *2* ^ (*k* − *1*) + *1*)
proof −
  have *H*: ¬ (∃*ka*. *i* = *2* ^ *ka* − *1*)
    proof (*rule ccontr*)
      assume ¬ *?thesis*
      then obtain k'::*nat* **where** k': *i* = *2* ^ k' − *1* **by** *blast*
      have (*2*::*nat*) ^ k' − *1* < *2* ^ *k* − *1*
        using *i-k* **unfolding** k' **.**
      then have (*2*::*nat*) ^ k' < *2* ^ *k*
        by *linarith*
      then have k' < *k*
        by *simp*
      have *2* ^ (*k* − *1*) ≤ *2* ^ k' − (*1*::*nat*)
        using *k-i* **unfolding** k' **.**
      then have (*2*::*nat*) ^ (*k*−*1*) < *2* ^ k'
        by (*metis Suc-diff-1 not-le not-less-eq zero-less-numeral zero-less-power*)
      then have *k*−*1* < k'
        by *simp*

      show *False* **using** ‹k' < *k*› ‹*k*−*1* < k'› **by** *linarith*
    qed
  have ⋀*k* k'. *2* ^ (*k* − *Suc 0*) ≤ *i* ⟹ *i* < *2* ^ *k* − *Suc 0* ⟹ *2* ^ (k' − *Suc 0*) ≤ *i* ⟹
    i < *2* ^ k' − *Suc 0* ⟹ *k* = k'
    by (*meson different-luby-decomposition-false linorder-neqE-nat*)
  then have *k*: (*SOME k*. *2* ^ (*k* − *Suc 0*) ≤ *i* ∧ *i* < *2* ^ *k* − *Suc 0*) = *k*
    using *k-i i-k* **by** *auto*
  show *?thesis*
    apply (*subst luby-sequence-core.simps*[*of i*], *subst H*)
    by (*simp add*: *k*)

qed

lemma *unbounded-luby-sequence-core*: *unbounded luby-sequence-core*
  unfolding *bounded-def*
proof
  assume $\exists\, b.\ \forall\, n.$ *luby-sequence-core* $n \leq b$
  then obtain b **where** b: $\bigwedge n.$ *luby-sequence-core* $n \leq b$
    by *metis*
  have *luby-sequence-core* $(2\hat{}(b+1) - 1) = 2\hat{}b$
    using *luby-sequence-core-two-power-minus-one*[*of* $b+1$] **by** *simp*
  moreover have $(2::nat)\hat{}b > b$
    by (*induction b*) *auto*
  ultimately show *False* **using** b[*of* $2\hat{}(b+1) - 1$] **by** *linarith*
qed

abbreviation *luby-sequence* :: *nat* $\Rightarrow$ *nat* **where**
luby-sequence $n \equiv$ *ur* $*$ *luby-sequence-core* n

lemma *bounded-luby-sequence*: *unbounded luby-sequence*
  using *bounded-const-product*[*of ur*] *luby-sequence-axioms*
  luby-sequence-def unbounded-luby-sequence-core **by** *blast*

lemma *luby-sequence-core-0*: *luby-sequence-core* $0 = 1$
proof −
  have *0*: $(0::nat) = 2\hat{}0-1$
    by *auto*
  show *?thesis*
    by (*subst 0*, *subst luby-sequence-core-two-power-minus-one*) *simp*
qed

lemma *luby-sequence-core* $n \geq 1$
proof (*induction n rule*: *nat-less-induct-case*)
  case *0*
  then show *?case* **by** (*simp add*: *luby-sequence-core-0*)
next
  case (*Suc n*) **note** $IH = this$

  consider
      (*interv*) k **where** $2 \hat{}\ (k - 1) \leq Suc\ n$ **and** $Suc\ n < 2 \hat{}\ k - 1$
    | (*pow2*) k **where** $Suc\ n = 2 \hat{}\ k - Suc\ 0$
    using *exists-luby-decomp*[*of Suc n*] **by** *auto*

  then show *?case*
     proof *cases*
       case *pow2*
       show *?thesis*
         using *luby-sequence-core-two-power-minus-one pow2* **by** *auto*
     next
       case *interv*
       have *n*: $Suc\ n - 2 \hat{}\ (k - 1) + 1 < Suc\ n$
         by (*metis Suc-1 Suc-eq-plus1 add.commute add-diff-cancel-left′ add-less-mono1 gr0I*
           interv(*1*) *interv*(*2*) *le-add-diff-inverse2 less-Suc-eq not-le power-0 power-one-right*
           power-strict-increasing-iff)
       show *?thesis*
         apply (*subst luby-sequence-core-not-two-power-minus-one*[*OF interv*])
         using *IH n* **by** *auto*

qed
qed
end

locale *luby-sequence-restart* =
luby-sequence ur +
conflict-driven-clause-learning$_W$
— functions for the state:
— access functions:
trail init-clss learned-clss backtrack-lvl conflicting
— changing state:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state
for
ur :: *nat* **and**
trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lit* **and**
init-clss :: *'st* ⇒ *'v clauses* **and**
learned-clss :: *'st* ⇒ *'v clauses* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
conflicting :: *'st* ⇒ *'v clause option* **and**

cons-trail :: (*'v*, *'v clause*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-learned-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'v clause option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'v clauses* ⇒ *'st*
begin

sublocale *cdcl*$_W$*-restart* - - - - - - - - - - - - *luby-sequence*
apply *unfold-locales*
using *bounded-luby-sequence* **by** *blast*

end
end
theory *CDCL-W-Incremental*
imports *CDCL-W-Termination*
begin

6.4 Incremental SAT solving

locale *state*$_W$*-adding-init-clause* =
state$_W$
— functions about the state:
— getter:
trail init-clss learned-clss backtrack-lvl conflicting
— setter:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— Some specific states:
init-state
for
trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
init-clss :: *'st* ⇒ *'v clauses* **and**
learned-clss :: *'st* ⇒ *'v clauses* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
conflicting :: *'st* ⇒ *'v clause option* **and**

cons-trail :: (*'v*, *'v clause*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-learned-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'v clause option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'v clauses* ⇒ *'st* +
fixes
add-init-cls :: *'v clause* ⇒ *'st* ⇒ *'st*
assumes
add-init-cls:
state st = (*M*, *N*, *U*, *S'*) ⟹
state (*add-init-cls C st*) = (*M*, {#*C*#} + *N*, *U*, *S'*)
begin

lemma
trail-add-init-cls[*simp*]:
trail (*add-init-cls C st*) = *trail st* **and**
init-clss-add-init-cls[*simp*]:
init-clss (*add-init-cls C st*) = {#*C*#} + *init-clss st*
and
learned-clss-add-init-cls[*simp*]:
learned-clss (*add-init-cls C st*) = *learned-clss st* **and**
backtrack-lvl-add-init-cls[*simp*]:
backtrack-lvl (*add-init-cls C st*) = *backtrack-lvl st* **and**
conflicting-add-init-cls[*simp*]:
conflicting (*add-init-cls C st*) = *conflicting st*
using *add-init-cls*[*of st - - - - C*] **by** (*cases state st*; *auto*)+

lemma *clauses-add-init-cls*[*simp*]:
clauses (*add-init-cls N S*) = {#*N*#} + *init-clss S* + *learned-clss S*
unfolding *clauses-def* **by** *auto*

lemma *reduce-trail-to-add-init-cls*[*simp*]:
trail (*reduce-trail-to F* (*add-init-cls C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *conflicting-add-init-cls-iff-conflicting*[*simp*]:
conflicting (*add-init-cls C S*) = *None* ⟷ *conflicting S* = *None*
by *fastforce*+
end

locale *conflict-driven-clause-learning-with-adding-init-clause*$_W$ =
state$_W$*-adding-init-clause*

— functions for the state:
— access functions:
trail init-clss learned-clss backtrack-lvl conflicting
— changing state:
cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl update-conflicting

— get state:
init-state
— Adding a clause:
add-init-cls
for
trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lit* **and**
init-clss :: *'st* ⇒ *'v clauses* **and**
learned-clss :: *'st* ⇒ *'v clauses* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
conflicting :: *'st* ⇒ *'v clause option* **and**

cons-trail :: (*'v*, *'v clause*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-learned-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'v clause* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'v clause option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'v clauses* ⇒ *'st* **and**
add-init-cls :: *'v clause* ⇒ *'st* ⇒ *'st*
begin

sublocale *conflict-driven-clause-learning*$_W$
by *unfold-locales*

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl*$_W$*-all-struct-inv*

definition *cdcl*$_W$*-stgy-invariant* **where**
cdcl$_W$*-stgy-invariant S* ⟷
conflict-is-false-with-level S
∧ *no-clause-is-false S*
∧ *no-smaller-confl S*
∧ *no-clause-is-false S*

lemma *cdcl*$_W$*-stgy-cdcl*$_W$*-stgy-invariant*:
assumes
cdcl$_W$: *cdcl*$_W$*-stgy S T* **and**
inv-s: *cdcl*$_W$*-stgy-invariant S* **and**
inv: *cdcl*$_W$*-all-struct-inv S*
shows
cdcl$_W$*-stgy-invariant T*
unfolding *cdcl*$_W$*-stgy-invariant-def cdcl*$_W$*-all-struct-inv-def* **apply** (*intro conjI*)
apply (*rule cdcl*$_W$*-stgy-ex-lit-of-max-level*[*of S*])
using *assms* **unfolding** *cdcl*$_W$*-stgy-invariant-def cdcl*$_W$*-all-struct-inv-def* **apply** *auto*[*7*]
using *cdcl*$_W$ *cdcl*$_W$*-stgy-not-non-negated-init-clss* **apply** *simp*
apply (*rule cdcl*$_W$*-stgy-no-smaller-confl-inv*)
using *assms* **unfolding** *cdcl*$_W$*-stgy-invariant-def cdcl*$_W$*-all-struct-inv-def* **apply** *auto*[*4*]
using *cdcl*$_W$ *cdcl*$_W$*-stgy-not-non-negated-init-clss* **by** *auto*

lemma *rtranclp-cdcl$_W$-stgy-cdcl$_W$-stgy-invariant*:
  assumes
   cdcl$_W$: *cdcl$_W$-stgy** S T* **and**
   inv-s: *cdcl$_W$-stgy-invariant S* **and**
   inv: *cdcl$_W$-all-struct-inv S*
  shows
    cdcl$_W$-stgy-invariant T
  using *assms* **apply** (*induction*)
    apply *simp*
  using *cdcl$_W$-stgy-cdcl$_W$-stgy-invariant rtranclp-cdcl$_W$-all-struct-inv-inv*
  rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$ **by** *blast*

abbreviation *decr-bt-lvl* **where**
decr-bt-lvl S ≡ *update-backtrack-lvl* (*backtrack-lvl S* − *1*) *S*

When we add a new clause, we reduce the trail until we get to tho first literal included in C. Then we can mark the conflict.

fun *cut-trail-wrt-clause* **where**
cut-trail-wrt-clause C [] *S* = *S* |
cut-trail-wrt-clause C (*Decided L* # *M*) *S* =
  (*if* −*L* ∈# *C then S*
    else cut-trail-wrt-clause C M (*decr-bt-lvl* (*tl-trail S*))) |
cut-trail-wrt-clause C (*Propagated L - # M*) *S* =
  (*if* −*L* ∈# *C then S*
    else cut-trail-wrt-clause C M (*tl-trail S*))

definition *add-new-clause-and-update* :: *'v clause* ⇒ *'st* ⇒ *'st* **where**
add-new-clause-and-update C S =
  (*if trail S* ⊨*as CNot C*
  then update-conflicting (*Some C*) (*add-init-cls C*
    (*cut-trail-wrt-clause C* (*trail S*) *S*))
  else add-init-cls C S)

thm *cut-trail-wrt-clause.induct*
lemma *init-clss-cut-trail-wrt-clause*[*simp*]:
  init-clss (*cut-trail-wrt-clause C M S*) = *init-clss S*
  by (*induction rule*: *cut-trail-wrt-clause.induct*) *auto*

lemma *learned-clss-cut-trail-wrt-clause*[*simp*]:
  learned-clss (*cut-trail-wrt-clause C M S*) = *learned-clss S*
  by (*induction rule*: *cut-trail-wrt-clause.induct*) *auto*

lemma *conflicting-clss-cut-trail-wrt-clause*[*simp*]:
  conflicting (*cut-trail-wrt-clause C M S*) = *conflicting S*
  by (*induction rule*: *cut-trail-wrt-clause.induct*) *auto*

lemma *trail-cut-trail-wrt-clause*:
  ∃ *M*. *trail S* = *M* @ *trail* (*cut-trail-wrt-clause C* (*trail S*) *S*)
proof (*induction trail S arbitrary*: *S rule*: *ann-lit-list-induct*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Decided L M*) **note** *IH* = *this*(*1*)[*of decr-bt-lvl* (*tl-trail S*)] **and** *M* = *this*(*2*)[*symmetric*]
  then show *?case* **using** *Cons-eq-appendI* **by** *fastforce+*
next

case (*Propagated L l M*) **note** *IH* = *this*(*1*)[*of tl-trail S*] **and** *M* = *this*(*2*)[*symmetric*]
then show *?case* **using** *Cons-eq-appendI* **by** *fastforce+*
qed

lemma *n-dup-no-dup-trail-cut-trail-wrt-clause*[*simp*]:
assumes *n-d*: *no-dup* (*trail T*)
shows *no-dup* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))
proof −
obtain *M* **where**
M: *trail T* = *M* @ *trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)
using *trail-cut-trail-wrt-clause*[*of T C*] **by** *auto*
show *?thesis*
using *n-d* **unfolding** *arg-cong*[*OF M*, *of no-dup*] **by** *auto*
qed

lemma *cut-trail-wrt-clause-backtrack-lvl-length-decided*:
assumes
backtrack-lvl T = *count-decided* (*trail T*)
shows
backtrack-lvl (*cut-trail-wrt-clause C* (*trail T*) *T*) =
count-decided (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))
using *assms*
proof (*induction trail T arbitrary*:*T rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Decided L M*) **note** *IH* = *this*(*1*)[*of decr-bt-lvl* (*tl-trail T*)] **and** *M* = *this*(*2*)[*symmetric*]
and *bt* = *this*(*3*)
then show *?case* **by** *auto*
next
case (*Propagated L l M*) **note** *IH* = *this*(*1*)[*of tl-trail T*] **and** *M* = *this*(*2*)[*symmetric*] **and** *bt* = *this*(*3*)
then show *?case* **by** *auto*
qed

lemma *cut-trail-wrt-clause-CNot-trail*:
assumes *trail T* ⊨*as CNot C*
shows
(*trail* ((*cut-trail-wrt-clause C* (*trail T*) *T*))) ⊨*as CNot C*
using *assms*
proof (*induction trail T arbitrary*:*T rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Decided L M*) **note** *IH* = *this*(*1*)[*of decr-bt-lvl* (*tl-trail T*)] **and** *M* = *this*(*2*)[*symmetric*]
and *bt* = *this*(*3*)
show *?case*
proof (*cases count C* (−*L*) = *0*)
case *False*
then show *?thesis*
using *IH M bt* **by** (*auto simp*: *true-annots-true-cls*)
next
case *True*
obtain *mma* :: *'v clause* **where**
f6: (*mma* ∈ {{#− *l*#} |*l*. *l* ∈# *C*} ⟶ *M* ⊨*a mma*) ⟶ *M* ⊨*as* {{#− *l*#} |*l*. *l* ∈# *C*}
using *true-annots-def* **by** *blast*

have $mma \in \{\{\#- l\#\} \mid l.\ l \in\# C\} \longrightarrow trail\ T \models a\ mma$
using *CNot-def M bt* **by** (*metis* (*no-types*) *true-annots-def*)
then have $M \models as\ \{\{\#- l\#\} \mid l.\ l \in\# C\}$
using *f6 True M bt* **by** (*force simp*: *count-eq-zero-iff*)
then show *?thesis*
using *IH true-annots-true-cls M* **by** (*auto simp*: *CNot-def*)
qed
next
case (*Propagated L l M*) **note** *IH* = *this*(*1*)[*of tl-trail T*] **and** *M* = *this*(*2*)[*symmetric*] **and** *bt* = *this*(*3*)
show *?case*
proof (*cases count C* $(-L) = 0$)
case *False*
then show *?thesis*
using *IH M bt* **by** (*auto simp*: *true-annots-true-cls*)
next
case *True*
obtain *mma* :: *'v clause* **where**
f6: $(mma \in \{\{\#- l\#\} \mid l.\ l \in\# C\} \longrightarrow M \models a\ mma) \longrightarrow M \models as\ \{\{\#- l\#\} \mid l.\ l \in\# C\}$
using *true-annots-def* **by** *blast*
have $mma \in \{\{\#- l\#\} \mid l.\ l \in\# C\} \longrightarrow trail\ T \models a\ mma$
using *CNot-def M bt* **by** (*metis* (*no-types*) *true-annots-def*)
then have $M \models as\ \{\{\#- l\#\} \mid l.\ l \in\# C\}$
using *f6 True M bt* **by** (*force simp*: *count-eq-zero-iff*)
then show *?thesis*
using *IH true-annots-true-cls M* **by** (*auto simp*: *CNot-def*)
qed
qed

lemma *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*:
$((\forall L \in\# C.\ -L \notin lits\text{-}of\text{-}l\ (trail\ T)) \wedge trail\ (cut\text{-}trail\text{-}wrt\text{-}clause\ C\ (trail\ T)\ T) = [])$
$\vee\ (-lit\text{-}of\ (hd\ (trail\ (cut\text{-}trail\text{-}wrt\text{-}clause\ C\ (trail\ T)\ T))) \in\# C$
$\wedge\ length\ (trail\ (cut\text{-}trail\text{-}wrt\text{-}clause\ C\ (trail\ T)\ T)) \geq 1)$
using *assms*
proof (*induction trail T arbitrary*:*T rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Decided L M*) **note** *IH* = *this*(*1*)[*of decr-bt-lvl* (*tl-trail T*)] **and** *M* = *this*(*2*)[*symmetric*]
then show *?case* **by** *simp force*
next
case (*Propagated L l M*) **note** *IH* = *this*(*1*)[*of tl-trail T*] **and** *M* = *this*(*2*)[*symmetric*]
then show *?case* **by** *simp force*
qed

We can fully run $cdcl_W$*-s* or add a clause. Remark that we use $cdcl_W$*-s* to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict C if possible.

inductive *incremental-cdcl*$_W$ :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* **where**
add-confl:
$trail\ S \models asm\ init\text{-}clss\ S \Longrightarrow distinct\text{-}mset\ C \Longrightarrow conflicting\ S = None \Longrightarrow$
$trail\ S \models as\ CNot\ C \Longrightarrow$
full $cdcl_W$*-stgy*
(*update-conflicting* (*Some C*)
(*add-init-cls C* (*cut-trail-wrt-clause C* (*trail S*) *S*))) $T \Longrightarrow$
incremental-cdcl$_W$ *S T* |
add-no-confl:

```
  trail S ⊨asm init-clss S ⟹ distinct-mset C ⟹ conflicting S = None ⟹
   ¬trail S ⊨as CNot C ⟹
   full cdcl_W-stgy (add-init-cls C S) T ⟹
   incremental-cdcl_W S T

lemma cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv:
  assumes
    inv-T: cdcl_W-all-struct-inv T and
    tr-T-N[simp]: trail T ⊨asm N and
    tr-C[simp]: trail T ⊨as CNot C and
    [simp]: distinct-mset C
  shows cdcl_W-all-struct-inv (add-new-clause-and-update C T) (is cdcl_W-all-struct-inv ?T')
proof -
  let ?T = update-conflicting (Some C)
    (add-init-cls C (cut-trail-wrt-clause C (trail T) T))
  obtain M where
    M: trail T = M @ trail (cut-trail-wrt-clause C (trail T) T)
      using trail-cut-trail-wrt-clause[of T C] by blast
  have H[dest]: ⋀x. x ∈ lits-of-l (trail (cut-trail-wrt-clause C (trail T) T)) ⟹
    x ∈ lits-of-l (trail T)
    using inv-T arg-cong[OF M, of lits-of-l] by auto
  have H'[dest]: ⋀x. x ∈ set (trail (cut-trail-wrt-clause C (trail T) T)) ⟹
    x ∈ set (trail T)
    using inv-T arg-cong[OF M, of set] by auto

  have H-proped:⋀x. x ∈ set (get-all-mark-of-propagated (trail (cut-trail-wrt-clause C
   (trail T) T))) ⟹ x ∈ set (get-all-mark-of-propagated (trail T))
  using inv-T arg-cong[OF M, of get-all-mark-of-propagated] by auto

  have [simp]: no-strange-atm ?T
    using inv-T unfolding cdcl_W-all-struct-inv-def no-strange-atm-def add-new-clause-and-update-def
    cdcl_W-M-level-inv-def by (auto 20 1)
  have M-lev: cdcl_W-M-level-inv T
    using inv-T unfolding cdcl_W-all-struct-inv-def by blast
  then have no-dup (M @ trail (cut-trail-wrt-clause C (trail T) T))
    unfolding cdcl_W-M-level-inv-def unfolding M[symmetric] by auto
  then have [simp]: no-dup (trail (cut-trail-wrt-clause C (trail T) T))
    by auto

  have consistent-interp (lits-of-l (M @ trail (cut-trail-wrt-clause C (trail T) T)))
    using M-lev unfolding cdcl_W-M-level-inv-def unfolding M[symmetric] by auto
  then have [simp]: consistent-interp (lits-of-l (trail (cut-trail-wrt-clause C
  (trail T) T)))
    unfolding consistent-interp-def by auto

  have [simp]: cdcl_W-M-level-inv ?T
    using M-lev unfolding cdcl_W-M-level-inv-def by (auto dest: H H'
      simp: M-lev cdcl_W-M-level-inv-def cut-trail-wrt-clause-backtrack-lvl-length-decided)

  have [simp]: ⋀s. s ∈# learned-clss T ⟹ ¬tautology s
    using inv-T unfolding cdcl_W-all-struct-inv-def by auto

  have distinct-cdcl_W-state T
    using inv-T unfolding cdcl_W-all-struct-inv-def by auto
  then have [simp]: distinct-cdcl_W-state ?T
    unfolding distinct-cdcl_W-state-def by auto
```

have *cdcl$_W$-conflicting T*
using *inv-T* **unfolding** *cdcl$_W$-all-struct-inv-def* **by** *auto*
have *trail ?T* $\models$*as CNot C*
by (*simp add*: *cut-trail-wrt-clause-CNot-trail*)
then have [*simp*]: *cdcl$_W$-conflicting ?T*
unfolding *cdcl$_W$-conflicting-def* **apply** *simp*
by (*metis M* ‹*cdcl$_W$-conflicting T*› *append-assoc cdcl$_W$-conflicting-decomp*(*2*))

have
decomp-T: *all-decomposition-implies-m* (*init-clss T*) (*get-all-ann-decomposition* (*trail T*))
using *inv-T* **unfolding** *cdcl$_W$-all-struct-inv-def* **by** *auto*
have *all-decomposition-implies-m* (*init-clss ?T*)
(*get-all-ann-decomposition* (*trail ?T*))
unfolding *all-decomposition-implies-def*
proof *clarify*
fix *a b*
assume (*a*, *b*) $\in$ *set* (*get-all-ann-decomposition* (*trail ?T*))
from *in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend*[*OF this*, *of M*]
obtain *b'* **where**
(*a*, *b'* @ *b*) $\in$ *set* (*get-all-ann-decomposition* (*trail T*))
using *M* **by** *auto*
then have *unmark-l a* $\cup$ *set-mset* (*init-clss T*) $\models$*ps unmark-l* (*b'* @ *b*)
using *decomp-T* **unfolding** *all-decomposition-implies-def* **by** *fastforce*
then have *unmark-l a* $\cup$ *set-mset* (*init-clss ?T*) $\models$*ps unmark-l* (*b* @ *b'*)
by (*simp add*: *Un-commute*)
then show *unmark-l a* $\cup$ *set-mset* (*init-clss ?T*) $\models$*ps unmark-l b*
by (*auto simp*: *image-Un*)
qed

have [*simp*]: *cdcl$_W$-learned-clause ?T*
using *inv-T* **unfolding** *cdcl$_W$-all-struct-inv-def cdcl$_W$-learned-clause-def*
by (*auto dest*!: *H-proped simp*: *clauses-def*)
show *?thesis*
using ‹*all-decomposition-implies-m* (*init-clss ?T*)
(*get-all-ann-decomposition* (*trail ?T*))›
unfolding *cdcl$_W$-all-struct-inv-def* **by** (*auto simp*: *add-new-clause-and-update-def*)
qed

lemma *cdcl$_W$-all-struct-inv-add-new-clause-and-update-cdcl$_W$-stgy-inv*:
assumes
inv-s: *cdcl$_W$-stgy-invariant T* **and**
inv: *cdcl$_W$-all-struct-inv T* **and**
tr-T-N[*simp*]: *trail T* $\models$*asm N* **and**
tr-C[*simp*]: *trail T* $\models$*as CNot C* **and**
[*simp*]: *distinct-mset C*
shows *cdcl$_W$-stgy-invariant* (*add-new-clause-and-update C T*)
(**is** *cdcl$_W$-stgy-invariant ?T'*)
proof −
have *cdcl$_W$-all-struct-inv ?T'*
using *cdcl$_W$-all-struct-inv-add-new-clause-and-update-cdcl$_W$-all-struct-inv assms* **by** *blast*
then have
no-dup-cut-T[*simp*]: *no-dup* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)) **and**
n-d[*simp*]: *no-dup* (*trail T*)
using *cdcl$_W$-M-level-inv-decomp*(*2*) *cdcl$_W$-all-struct-inv-def inv*
n-dup-no-dup-trail-cut-trail-wrt-clause **by** *blast+*

then have *trail* (*add-new-clause-and-update C T*) $\models$*as CNot C*
  by (*simp add*: *add-new-clause-and-update-def cut-trail-wrt-clause-CNot-trail*
    cdcl$_W$*-M-level-inv-def cdcl*$_W$*-all-struct-inv-def*)
obtain *MT* **where**
  MT: *trail T* = *MT* @ *trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)
  using *trail-cut-trail-wrt-clause* **by** *blast*
consider
    (*false*) $\forall$ *L*$\in$#*C*. $-$ *L* $\notin$ *lits-of-l* (*trail T*) **and**
      trail (*cut-trail-wrt-clause C* (*trail T*) *T*) = []
  | (*not-false*)
    $-$ *lit-of* (*hd* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))) $\in$# *C* **and**
    1 $\leq$ *length* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))
  using *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*[*of C T*] **by** *auto*
then show *?thesis*
  proof *cases*
    case *false* **note** *C* = *this*(*1*) **and** *empty-tr* = *this*(*2*)
    then have [*simp*]: *C* = {#}
      by (*simp add*: *in-CNot-implies-uminus*(*2*) *multiset-eqI*)
    show *?thesis*
      using *empty-tr* **unfolding** *cdcl*$_W$*-stgy-invariant-def no-smaller-confl-def*
      cdcl$_W$*-all-struct-inv-def* **by** (*auto simp*: *add-new-clause-and-update-def*)
  next
    case *not-false* **note** *C* = *this*(*1*) **and** *l* = *this*(*2*)
    let *?L* = $-$ *lit-of* (*hd* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)))
    have *L*: *get-level* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)) ($-$*?L*)
      = *count-decided* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))
      apply (*cases trail* (*add-init-cls C*
          (*cut-trail-wrt-clause C* (*trail T*) *T*));
       cases hd (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)))
      using *l* **by** (*auto split*: *if-split-asm*
        simp:*rev-swap*[*symmetric*] *add-new-clause-and-update-def*)

    have *L'*: *count-decided*(*trail* (*cut-trail-wrt-clause C*
      (*trail T*) *T*))
      = *backtrack-lvl* (*cut-trail-wrt-clause C* (*trail T*) *T*)
      using ‹*cdcl*$_W$*-all-struct-inv ?T'*› **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def*
      by (*auto simp*:*add-new-clause-and-update-def*)

    have [*simp*]: *no-smaller-confl* (*update-conflicting* (*Some C*)
      (*add-init-cls C* (*cut-trail-wrt-clause C* (*trail T*) *T*)))
      unfolding *no-smaller-confl-def*
    proof (*clarify*, *goal-cases*)
      case (*1 M K M' D*)
      then consider
          (*DC*) *D* = *C*
        | (*D-T*) *D* $\in$# *clauses T*
        by (*auto simp*: *clauses-def split*: *if-split-asm*)
      then show *False*
        proof *cases*
          case *D-T*
          have *no-smaller-confl T*
            using *inv-s* **unfolding** *cdcl*$_W$*-stgy-invariant-def* **by** *auto*
          have (*MT* @ *M'*) @ *Decided K* # *M* = *trail T*
            using *MT 1*(*1*) **by** *auto*
          then show *False*
            using *D-T* ‹*no-smaller-confl T*› *1*(*3*) **unfolding** *no-smaller-confl-def* **by** *blast*

next
case *DC* **note** -[*simp*] = *this*
then have *atm-of* $(-?L) \in$ *atm-of* ' (*lits-of-l M*)
using *1*(*3*) *C in-CNot-implies-uminus*(*2*) **by** *blast*
moreover
have *lit-of* (*hd* (M' @ *Decided K* # [])) = $-?L$
using *l 1*(*1*)[*symmetric*] *inv*
by (*cases* M', *cases trail* (*add-init-cls C*
(*cut-trail-wrt-clause C* (*trail T*) *T*)))
(*auto dest*!: *arg-cong*[*of* - # - - *hd*] *simp*: *hd-append* $cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-M-level-inv-def*)
from *arg-cong*[*OF this*, *of atm-of*]
have *atm-of* $(-?L) \in$ *atm-of* ' (*lits-of-l* (M' @ *Decided K* # []))
by (*cases* (M' @ *Decided K* # [])) *auto*
moreover have *no-dup* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))
using ‹$cdcl_W$*-all-struct-inv ?T'*› **unfolding** $cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *add-new-clause-and-update-def*)
ultimately show *False*
unfolding *1*(*1*)[*symmetric*, *simplified*] **by** (*auto simp*: *lits-of-def*)
qed
qed
show *?thesis* **using** *L L' C*
unfolding $cdcl_W$*-stgy-invariant-def* $cdcl_W$*-all-struct-inv-def*
by (*auto simp*: *add-new-clause-and-update-def intro*: *rev-bexI*)
qed
qed

lemma *full-*$cdcl_W$*-stgy-inv-normal-form*:
assumes
full: *full* $cdcl_W$*-stgy S T* **and**
inv-s: $cdcl_W$*-stgy-invariant S* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows *conflicting T* = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss S*))
$\vee$ *conflicting T* = *None* $\wedge$ *trail T* $\models$*asm init-clss S* $\wedge$ *satisfiable* (*set-mset* (*init-clss S*))
proof –
have *no-step* $cdcl_W$*-stgy T*
using *full* **unfolding** *full-def* **by** *blast*
moreover have $cdcl_W$*-all-struct-inv T* **and** *inv-s*: $cdcl_W$*-stgy-invariant T*
apply (*metis rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$ *full full-def inv*
rtranclp-$cdcl_W$*-all-struct-inv-inv*)
by (*metis full full-def inv inv-s rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-stgy-invariant*)
ultimately have *conflicting T* = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss T*))
$\vee$ *conflicting T* = *None* $\wedge$ *trail T* $\models$*asm init-clss T*
using $cdcl_W$*-stgy-final-state-conclusive*[*of T*] *full*
unfolding $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-stgy-invariant-def full-def* **by** *fast*
moreover have *consistent-interp* (*lits-of-l* (*trail T*))
using ‹$cdcl_W$*-all-struct-inv T*› **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def*
by *auto*
moreover have *init-clss S* = *init-clss T*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def*
by (*metis rtranclp-*$cdcl_W$*-stgy-no-more-init-clss full full-def*)
ultimately show *?thesis*
by (*metis satisfiable-carac' true-annot-def true-annots-def true-clss-def*)
qed

lemma *incremental-*$cdcl_W$*-inv*:

assumes
inc: *incremental-cdcl*$_W$ *S T* **and**
inv: *cdcl*$_W$*-all-struct-inv S* **and**
s-inv: *cdcl*$_W$*-stgy-invariant S*
shows
cdcl$_W$*-all-struct-inv T* **and**
cdcl$_W$*-stgy-invariant T*
using *inc*
proof (*induction*)
case (*add-confl C T*)
let *?T* = (*update-conflicting* (*Some C*) (*add-init-cls C*
(*cut-trail-wrt-clause C* (*trail S*) *S*)))
have *cdcl*$_W$*-all-struct-inv ?T* **and** *inv-s-T*: *cdcl*$_W$*-stgy-invariant ?T*
using *add-confl.hyps*(*1*,*2*,*4*) *add-new-clause-and-update-def*
cdcl$_W$*-all-struct-inv-add-new-clause-and-update-cdcl*$_W$*-all-struct-inv inv* **apply** *auto*[*1*]
using *add-confl.hyps*(*1*,*2*,*4*) *add-new-clause-and-update-def*
cdcl$_W$*-all-struct-inv-add-new-clause-and-update-cdcl*$_W$*-stgy-inv inv s-inv* **by** *auto*
case *1* **show** *?case*
by (*metis add-confl.hyps*(*1*,*2*,*4*,*5*) *add-new-clause-and-update-def*
cdcl$_W$*-all-struct-inv-add-new-clause-and-update-cdcl*$_W$*-all-struct-inv*
rtranclp-cdcl$_W$*-all-struct-inv-inv rtranclp-cdcl*$_W$*-stgy-rtranclp-cdcl*$_W$ *full-def inv*)

case *2* **show** *?case*
by (*metis inv-s-T add-confl.hyps*(*1*,*2*,*4*,*5*) *add-new-clause-and-update-def*
cdcl$_W$*-all-struct-inv-add-new-clause-and-update-cdcl*$_W$*-all-struct-inv full-def inv*
rtranclp-cdcl$_W$*-stgy-cdcl*$_W$*-stgy-invariant*)
next
case (*add-no-confl C T*)
case *1*
have *cdcl*$_W$*-all-struct-inv* (*add-init-cls C S*)
using *inv* ⟨*distinct-mset C*⟩ **unfolding** *cdcl*$_W$*-all-struct-inv-def no-strange-atm-def*
cdcl$_W$*-M-level-inv-def distinct-cdcl*$_W$*-state-def cdcl*$_W$*-conflicting-def cdcl*$_W$*-learned-clause-def*
by (*auto 9 1 simp*: *all-decomposition-implies-insert-single clauses-def*)

then show *?case*
using *add-no-confl*(*5*) **unfolding** *full-def* **by** (*auto intro*: *rtranclp-cdcl*$_W$*-stgy-cdcl*$_W$*-all-struct-inv*)
case *2*
have *nc*: $\forall M.\ (\exists K\ i\ M'.\ trail\ S = M'\ @\ Decided\ K\ \#\ M) \longrightarrow \neg\ M \models as\ CNot\ C$
using ⟨¬ *trail S* ⊨*as CNot C*⟩
by (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model*)

have *cdcl*$_W$*-stgy-invariant* (*add-init-cls C S*)
using *s-inv* ⟨¬ *trail S* ⊨*as CNot C*⟩ *inv* **unfolding** *cdcl*$_W$*-stgy-invariant-def*
no-smaller-confl-def eq-commute[*of - trail -*] *cdcl*$_W$*-M-level-inv-def cdcl*$_W$*-all-struct-inv-def*
by (*auto simp*: *clauses-def nc*)
then show *?case*
by (*metis* ⟨*cdcl*$_W$*-all-struct-inv* (*add-init-cls C S*)⟩ *add-no-confl.hyps*(*5*) *full-def*
rtranclp-cdcl$_W$*-stgy-cdcl*$_W$*-stgy-invariant*)
qed

lemma *rtranclp-incremental-cdcl*$_W$*-inv*:
assumes
inc: *incremental-cdcl*$_W$** *S T* **and**
inv: *cdcl*$_W$*-all-struct-inv S* **and**
s-inv: *cdcl*$_W$*-stgy-invariant S*
shows

cdcl$_W$*-all-struct-inv T* **and**
cdcl$_W$*-stgy-invariant T*
using *inc* **apply** *induction*
using *inv* **apply** *simp*
using *s-inv* **apply** *simp*
using *incremental-cdcl*$_W$*-inv* **by** *blast+*

lemma *incremental-conclusive-state*:
assumes
inc: *incremental-cdcl*$_W$ *S T* **and**
inv: *cdcl*$_W$*-all-struct-inv S* **and**
s-inv: *cdcl*$_W$*-stgy-invariant S*
shows *conflicting T* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss T*))
∨ *conflicting T* = *None* ∧ *trail T* ⊨*asm init-clss T* ∧ *satisfiable* (*set-mset* (*init-clss T*))
using *inc*
proof *induction*
print-cases
case (*add-confl C T*) **note** *tr* = *this*(*1*) **and** *dist* = *this*(*2*) **and** *conf* = *this*(*3*) **and** *C* = *this*(*4*) **and** *full* = *this*(*5*)

have *full cdcl*$_W$*-stgy T T*
using *full* **unfolding** *full-def* **by** *auto*
then show *?case*
using *full C conf dist tr*
by (*metis full-cdcl*$_W$*-stgy-inv-normal-form incremental-cdcl*$_W$*.simps incremental-cdcl*$_W$*-inv*(*1*) *incremental-cdcl*$_W$*-inv*(*2*) *inv s-inv*)
next
case (*add-no-confl C T*) **note** *tr* = *this*(*1*) **and** *dist* = *this*(*2*) **and** *conf* = *this*(*3*) **and** *C* = *this*(*4*) **and** *full* = *this*(*5*)

have *full cdcl*$_W$*-stgy T T*
using *full* **unfolding** *full-def* **by** *auto*
then show *?case*
by (*meson C conf dist full full-cdcl*$_W$*-stgy-inv-normal-form incremental-cdcl*$_W$*.add-no-confl incremental-cdcl*$_W$*-inv*(*1*) *incremental-cdcl*$_W$*-inv*(*2*) *inv s-inv tr*)
qed

lemma *tranclp-incremental-correct*:
assumes
inc: *incremental-cdcl*$_W$$^{++}$ *S T* **and**
inv: *cdcl*$_W$*-all-struct-inv S* **and**
s-inv: *cdcl*$_W$*-stgy-invariant S*
shows *conflicting T* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss T*))
∨ *conflicting T* = *None* ∧ *trail T* ⊨*asm init-clss T* ∧ *satisfiable* (*set-mset* (*init-clss T*))
using *inc* **apply** *induction*
using *assms incremental-conclusive-state* **apply** *blast*
by (*meson incremental-conclusive-state inv rtranclp-incremental-cdcl*$_W$*-inv s-inv tranclp-into-rtranclp*)

end

end
theory *DPLL-CDCL-W-Implementation*
imports *Partial-Annotated-Clausal-Logic CDCL-W-Level*
begin

Chapter 7

Implementation of DPLL and CDCL

We then reuse all the theorems to go towards an implementation using 2-watched literals:

- `CDCL_W_Abstract_State.thy` defines a better-suited state: the operation operating on it are more constrained, allowing simpler proofs and less edge cases later.

7.1 Simple List-Based Implementation of the DPLL and CDCL

The idea of the list-based implementation is to test the stack: the theories about the calculi, adapting the theorems to a simple implementation and the code exportation. The implementation are very simple ans simply iterate over-and-over on lists.

7.1.1 Common Rules

Propagation

The following theorem holds:

lemma *lits-of-l-unfold*[*iff*]:
  $(\forall\, c \in \mathit{set}\ C.\ -c \in \mathit{lits\text{-}of\text{-}l}\ \mathit{Ms}) \longleftrightarrow \mathit{Ms} \models\!\mathit{as}\ \mathit{CNot}\ (\mathit{mset}\ C)$
  unfolding *true-annots-def Ball-def true-annot-def CNot-def* **by** *auto*

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: $'a$ *literal list* $\Rightarrow$ $('a, 'b)$ *ann-lits* $\Rightarrow$ $'a$ *literal option*
 where
 is-unit-clause l M =
  (*case List.filter* ($\lambda a.$ *atm-of a* $\notin$ *atm-of* ‘ *lits-of-l M*) *l of*
   a # [] $\Rightarrow$ *if M* $\models$*as CNot* (*mset l* $-$ {#*a*#}) *then Some a else None*
  | - $\Rightarrow$ *None*)

definition *is-unit-clause-code* :: $'a$ *literal list* $\Rightarrow$ $('a, 'b)$ *ann-lits*
  $\Rightarrow$ $'a$ *literal option* **where**
 is-unit-clause-code l M =
  (*case List.filter* ($\lambda a.$ *atm-of a* $\notin$ *atm-of* ‘ *lits-of-l M*) *l of*
   a # [] $\Rightarrow$ *if* ($\forall$ *c* $\in$*set* (*remove1 a l*). $-c \in$ *lits-of-l M*) *then Some a else None*
  | - $\Rightarrow$ *None*)

lemma *is-unit-clause-is-unit-clause-code*[*code*]:
  is-unit-clause l M = *is-unit-clause-code l M*

proof −
  have *1*: $\bigwedge a.$ ($\forall$ *c*∈*set* (*remove1 a l*). − *c* ∈ *lits-of-l M*) ⟷ *M* ⊨*as CNot* (*mset l* − {#*a*#})
    using *lits-of-l-unfold*[*of remove1 - l*, *of - M*] **by** *simp*
  then show *?thesis*
    unfolding *is-unit-clause-code-def is-unit-clause-def 1* **by** *blast*
qed

lemma *is-unit-clause-some-undef*:
  assumes *is-unit-clause l M* = *Some a*
  shows *undefined-lit M a*
proof −
  have (*case* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*] *of* [] ⇒ *None*
          | [*a*] ⇒ *if M* ⊨*as CNot* (*mset l* − {#*a*#}) *then Some a else None*
          | *a* # *ab* # *xa* ⇒ *Map.empty xa*) = *Some a*
    using *assms* **unfolding** *is-unit-clause-def* **.**
  then have *a* ∈ *set* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*]
    apply (*cases* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*])
      apply *simp*
    apply (*rename-tac aa list*; *case-tac list*) **by** (*auto split*: *if-split-asm*)
  then have *atm-of a* ∉ *atm-of* ' *lits-of-l M* **by** *auto*
  then show *?thesis*
    by (*simp add*: *Decided-Propagated-in-iff-in-lits-of-l*
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
qed

lemma *is-unit-clause-some-CNot*: *is-unit-clause l M* = *Some a* ⟹ *M* ⊨*as CNot* (*mset l* − {#*a*#})
  unfolding *is-unit-clause-def*
proof −
  assume (*case* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*] *of* [] ⇒ *None*
          | [*a*] ⇒ *if M* ⊨*as CNot* (*mset l* − {#*a*#}) *then Some a else None*
          | *a* # *ab* # *xa* ⇒ *Map.empty xa*) = *Some a*
  then show *?thesis*
    apply (*cases* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*], *simp*)
      apply *simp*
    apply (*rename-tac aa list*, *case-tac list*) **by** (*auto split*: *if-split-asm*)
qed

lemma *is-unit-clause-some-in*: *is-unit-clause l M* = *Some a* ⟹ *a* ∈ *set l*
  unfolding *is-unit-clause-def*
proof −
  assume (*case* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*] *of* [] ⇒ *None*
         | [*a*] ⇒ *if M* ⊨*as CNot* (*mset l* − {#*a*#}) *then Some a else None*
         | *a* # *ab* # *xa* ⇒ *Map.empty xa*) = *Some a*
  then show *a* ∈ *set l*
    by (*cases* [*a*←*l* . *atm-of a* ∉ *atm-of* ' *lits-of-l M*])
       (*fastforce dest*: *filter-eq-ConsD split*: *if-split-asm* *split*: *list.splits*)+
qed

lemma *is-unit-clause-Nil*[*simp*]: *is-unit-clause* [] *M* = *None*
  unfolding *is-unit-clause-def* **by** *auto*

Unit propagation for all clauses

Finding the first clause to propagate

fun *find-first-unit-clause* :: *'a literal list list* ⇒ (*'a*, *'b*) *ann-lits*

$\Rightarrow$ (*'a literal* $\times$ *'a literal list*) *option* **where**
find-first-unit-clause $(a \# l)$ M =
 (*case is-unit-clause a M of*
  None $\Rightarrow$ *find-first-unit-clause l M*
 | *Some L* $\Rightarrow$ *Some* (L, a)) |
find-first-unit-clause [] - = *None*

lemma *find-first-unit-clause-some*:
 find-first-unit-clause l M = *Some* (a, c)
 $\Longrightarrow c \in$ *set* l $\wedge$ M $\models$*as CNot* (*mset* $c - \{\#a\#\}$) $\wedge$ *undefined-lit M a* $\wedge$ $a \in$ *set* c
 apply (*induction l*)
  apply *simp*
 by (*auto split*: *option.splits dest*: *is-unit-clause-some-in is-unit-clause-some-CNot*
   is-unit-clause-some-undef)

lemma *propagate-is-unit-clause-not-None*:
 assumes *dist*: *distinct c* **and**
 M: M $\models$*as CNot* (*mset* $c - \{\#a\#\}$) **and**
 undef: *undefined-lit M a* **and**
 ac: $a \in$ *set* c
 shows *is-unit-clause c M* $\neq$ *None*
proof −
 have [$a \leftarrow c$. *atm-of* $a \notin$ *atm-of* ' *lits-of-l M*] = [a]
  using *assms*
  proof (*induction c*)
   case *Nil* **then show** *?case* **by** *simp*
  next
   case (*Cons ac c*)
   show *?case*
    proof (*cases* $a = ac$)
     case *True*
     then show *?thesis* **using** *Cons*
      by (*auto simp del*: *lits-of-l-unfold*
        simp add: *lits-of-l-unfold*[*symmetric*] *Decided-Propagated-in-iff-in-lits-of-l*
         atm-of-eq-atm-of atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
    next
     case *False*
     then have *T*: *mset* $c + \{\#ac\#\} - \{\#a\#\}$ = *mset* $c - \{\#a\#\} + \{\#ac\#\}$
      by (*auto simp add*: *multiset-eq-iff*)
     show *?thesis* **using** *False Cons*
      by (*auto simp add*: *T atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)
    qed
  qed
 then show *?thesis*
  using *M* **unfolding** *is-unit-clause-def* **by** *auto*
qed

lemma *find-first-unit-clause-none*:
 distinct c $\Longrightarrow c \in$ *set* $l \Longrightarrow$ M $\models$*as CNot* (*mset* $c - \{\#a\#\}$) $\Longrightarrow$ *undefined-lit M a* $\Longrightarrow a \in$ *set* c
 $\Longrightarrow$ *find-first-unit-clause l M* $\neq$ *None*
 by (*induction l*)
  (*auto split*: *option.split simp add*: *propagate-is-unit-clause-not-None*)

Decide

fun *find-first-unused-var* :: *'a literal list list* $\Rightarrow$ *'a literal set* $\Rightarrow$ *'a literal option* **where**

find-first-unused-var $(a \# l)$ M =
  (*case List.find* (λ*lit. lit* $\notin M \wedge -$*lit* $\notin M$) *a of*
    None $\Rightarrow$ *find-first-unused-var l M*
  | *Some a* $\Rightarrow$ *Some a*) |
find-first-unused-var [] - = *None*

lemma *find-none*[*iff*]:
  List.find (λ*lit. lit* $\notin M \wedge -$*lit* $\notin M$) *a* = *None* $\longleftrightarrow$ *atm-of* ' *set a* $\subseteq$ *atm-of* ' *M*
  apply (*induct a*)
  using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*
    by (*force simp add*: *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)+

lemma *find-some*: *List.find* (λ*lit. lit* $\notin M \wedge -$*lit* $\notin M$) *a* = *Some b* $\Longrightarrow$ $b \in$ *set a* $\wedge$ $b \notin M \wedge -b \notin M$
  unfolding *find-Some-iff* **by** (*metis nth-mem*)

lemma *find-first-unused-var-None*[*iff*]:
  find-first-unused-var l M = *None* $\longleftrightarrow$ ($\forall a \in$ *set l. atm-of* ' *set a* $\subseteq$ *atm-of* ' *M*)
  by (*induct l*)
     (*auto split*: *option.splits dest*!: *find-some*
       simp add: *image-subset-iff atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)

lemma *find-first-unused-var-Some-not-all-incl*:
  assumes *find-first-unused-var l M* = *Some c*
  shows $\neg$($\forall a \in$ *set l. atm-of* ' *set a* $\subseteq$ *atm-of* ' *M*)
proof –
  have *find-first-unused-var l M* $\neq$ *None*
    using *assms* **by** (*cases find-first-unused-var l M*) *auto*
  then show $\neg$($\forall a \in$ *set l. atm-of* ' *set a* $\subseteq$ *atm-of* ' *M*) **by** *auto*
qed

lemma *find-first-unused-var-Some*:
  find-first-unused-var l M = *Some a* $\Longrightarrow$ ($\exists m \in$ *set l. a* $\in$ *set m* $\wedge$ $a \notin M \wedge -a \notin M$)
  by (*induct l*) (*auto split*: *option.splits dest*: *find-some*)

lemma *find-first-unused-var-undefined*:
  find-first-unused-var l (*lits-of-l Ms*) = *Some a* $\Longrightarrow$ *undefined-lit Ms a*
  using *find-first-unused-var-Some*[*of l lits-of-l Ms a*] *Decided-Propagated-in-iff-in-lits-of-l*
  by *blast*

7.1.2 CDCL specific functions

Level

fun *maximum-level-code*:: *'a literal list* $\Rightarrow$ (*'a*, *'b*) *ann-lits* $\Rightarrow$ *nat*
  where
maximum-level-code [] - = *0* |
maximum-level-code (*L* # *Ls*) *M* = *max* (*get-level M L*) (*maximum-level-code Ls M*)

lemma *maximum-level-code-eq-get-maximum-level*[*simp*]:
  maximum-level-code D M = *get-maximum-level M* (*mset D*)
  by (*induction D*) (*auto simp add*: *get-maximum-level-plus*)

lemma [*code*]:
  fixes *M* :: (*'a*, *'b*) *ann-lits*
  shows *get-maximum-level M* (*mset D*) = *maximum-level-code D M*
  by *simp*

Backjumping

```
fun find-level-decomp where
find-level-decomp M [] D k = None |
find-level-decomp M (L # Ls) D k =
  (case (get-level M L, maximum-level-code (D @ Ls) M) of
    (i, j) ⇒ if i = k ∧ j < i then Some (L, j) else find-level-decomp M Ls (L#D) k
  )

lemma find-level-decomp-some:
  assumes find-level-decomp M Ls D k = Some (L, j)
  shows L ∈ set Ls ∧ get-maximum-level M (mset (remove1 L (Ls @ D))) = j ∧ get-level M L = k
  using assms
proof (induction Ls arbitrary: D)
  case Nil
  then show ?case by simp
next
  case (Cons L' Ls) note IH = this(1) and H = this(2)

  def find ≡ (if get-level M L' ≠ k ∨ ¬ get-maximum-level M (mset D + mset Ls) < get-level M L'
    then find-level-decomp M Ls (L' # D) k
    else Some (L', get-maximum-level M (mset D + mset Ls)))
  have a1: ⋀D. find-level-decomp M Ls D k = Some (L, j) ⟹
     L ∈ set Ls ∧ get-maximum-level M (mset Ls + mset D − {#L#}) = j ∧ get-level M L = k
    using IH by simp
  have a2: find = Some (L, j)
    using H unfolding find-def by (auto split: if-split-asm)
  { assume Some (L', get-maximum-level M (mset D + mset Ls)) ≠ find
    then have f3: L ∈ set Ls and get-maximum-level M (mset Ls + mset (L' # D) − {#L#}) = j
      using a1 IH a2 unfolding find-def by meson+
    moreover then have mset Ls + mset D − {#L#} + {#L'#} = {#L'#} + mset D + (mset Ls
− {#L#})
      by (auto simp: ac-simps multiset-eq-iff Suc-leI)
    ultimately have f4: get-maximum-level M (mset Ls + mset D − {#L#} + {#L'#}) = j
      by (metis add.commute diff-union-single-conv in-multiset-in-set mset.simps(2))
  } note f4 = this
  have {#L'#} + (mset Ls + mset D) = mset Ls + (mset D + {#L'#})
      by (auto simp: ac-simps)
  then have
    L = L' ⟶ get-maximum-level M (mset Ls + mset D) = j ∧ get-level M L' = k and
    L ≠ L' ⟶ L ∈ set Ls ∧ get-maximum-level M (mset Ls + mset D − {#L#} + {#L'#}) = j ∧
      get-level M L = k
      using a2 a1[of L' # D] unfolding find-def apply (metis add-diff-cancel-left' mset.simps(2)
        option.inject prod.inject union-commute)
    using f4 a2 a1[of L' # D] unfolding find-def by (metis option.inject prod.inject)
  then show ?case by simp
qed

lemma find-level-decomp-none:
  assumes find-level-decomp M Ls E k = None and mset (L#D) = mset (Ls @ E)
  shows ¬(L ∈ set Ls ∧ get-maximum-level M (mset D) < k ∧ k = get-level M L)
  using assms
proof (induction Ls arbitrary: E L D)
  case Nil
  then show ?case by simp
next
```

case (*Cons L′ Ls*) **note** *IH* = *this*(*1*) **and** *find-none* = *this*(*2*) **and** *LD* = *this*(*3*)
have *mset D* + {#*L′*#} = *mset E* + (*mset Ls* + {#*L′*#}) $\Longrightarrow$ *mset D* = *mset E* + *mset Ls*
by (*metis add-right-imp-eq union-assoc*)
then show *?case*
using *find-none IH*[*of L′ # E L D*] *LD* **by** (*auto simp add*: *ac-simps split*: *if-split-asm*)
qed

fun *bt-cut* **where**
bt-cut i (*Propagated - - # Ls*) = *bt-cut i Ls* |
bt-cut i (*Decided K # Ls*) = (*if count-decided Ls* = *i then Some* (*Decided K # Ls*) *else bt-cut i Ls*) |
bt-cut i [] = *None*

lemma *bt-cut-some-decomp*:
assumes *no-dup M* **and** *bt-cut i M* = *Some M′*
shows ∃ *K M2 M1*. *M* = *M2* @ *M′* ∧ *M′* = *Decided K # M1* ∧ *get-level M K* = (*i*+*1*)
using *assms* **by** (*induction i M rule*: *bt-cut.induct*) (*auto split*: *if-split-asm*)

lemma *bt-cut-not-none*:
assumes *no-dup M* **and** *M* = *M2* @ *Decided K # M′* **and** *get-level M K* = (*i*+*1*)
shows *bt-cut i M* ≠ *None*
using *assms* **by** (*induction M2 arbitrary*: *M rule*: *ann-lit-list-induct*)
(*auto simp*: *atm-lit-of-set-lits-of-l*)

lemma *get-all-ann-decomposition-ex*:
∃ *N*. (*Decided K # M′*, *N*) ∈ *set* (*get-all-ann-decomposition* (*M2*@*Decided K # M′*))
apply (*induction M2 rule*: *ann-lit-list-induct*)
apply *auto*[*2*]
by (*rename-tac L m xs*, *case-tac get-all-ann-decomposition* (*xs* @ *Decided K # M′*))
auto

lemma *bt-cut-in-get-all-ann-decomposition*:
assumes *no-dup M* **and** *bt-cut i M* = *Some M′*
shows ∃ *M2*. (*M′*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)
using *bt-cut-some-decomp*[*OF assms*] **by** (*auto simp add*: *get-all-ann-decomposition-ex*)

fun *do-backtrack-step* **where**
do-backtrack-step (*M*, *N*, *U*, *k*, *Some D*) =
(*case find-level-decomp M D* [] *k of*
None ⇒ (*M*, *N*, *U*, *k*, *Some D*)
| *Some* (*L*, *j*) ⇒
(*case bt-cut j M of*
Some (*Decided - # Ls*) ⇒ (*Propagated L D # Ls*, *N*, *D # U*, *j*, *None*)
| *-* ⇒ (*M*, *N*, *U*, *k*, *Some D*))
) |
do-backtrack-step S = *S*

end
theory *DPLL-W-Implementation*
imports *DPLL-CDCL-W-Implementation DPLL-W* ~~/*src*/*HOL*/*Library*/*Code-Target-Numeral*
begin

7.1.3 Simple Implementation of DPLL

Combining the propagate and decide: a DPLL step

definition *DPLL-step* :: *int* $dpll_W$*-ann-lits* × *int literal list list*

$\Rightarrow$ *int* $dpll_W$*-ann-lits* $\times$ *int literal list list* **where**
DPLL-step = (λ(*Ms*, *N*).
(*case find-first-unit-clause N Ms of*
Some (*L*, -) $\Rightarrow$ (*Propagated L* () # *Ms*, *N*)
| - $\Rightarrow$
if $\exists$ *C* $\in$ *set N*. ($\forall$ *c* $\in$ *set C*. $-c$ $\in$ *lits-of-l Ms*)
then
(*case backtrack-split Ms of*
(-, *L* # *M*) $\Rightarrow$ (*Propagated* ($-$ (*lit-of L*)) () # *M*, *N*)
| (-, -) $\Rightarrow$ (*Ms*, *N*)
)
else
(*case find-first-unused-var N* (*lits-of-l Ms*) *of*
Some a $\Rightarrow$ (*Decided a* # *Ms*, *N*)
| *None* $\Rightarrow$ (*Ms*, *N*))))

Example of propagation:

value *DPLL-step* ([*Decided* (*Neg 1*)], [[*Pos* (*1*::*int*), *Neg 2*]])

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation *toS* $\equiv$ λ(*Ms*::(*int*, *unit*) *ann-lits*)
(*N*:: *int literal list list*). (*Ms*, *mset* (*map mset N*))
abbreviation *toS'* $\equiv$ λ(*Ms*::(*int*, *unit*) *ann-lits*,
N:: *int literal list list*). (*Ms*, *mset* (*map mset N*))

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-*$dpll_W$*-step*:
assumes *step*: (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*)
and *neq*: (*Ms*, *N*) $\neq$ (*Ms'*, *N'*)
shows $dpll_W$ (*toS Ms N*) (*toS Ms' N'*)
proof $-$
let *?S* = (*Ms*, *mset* (*map mset N*))
{ fix *L E*
assume *unit*: *find-first-unit-clause N Ms* = *Some* (*L*, *E*)
then have *Ms'N*: (*Ms'*, *N'*) = (*Propagated L* () # *Ms*, *N*)
using *step* **unfolding** *DPLL-step-def* **by** *auto*
obtain *C* **where**
C: *C* $\in$ *set N* **and**
Ms: *Ms* $\models$*as CNot* (*mset C* $-$ {#*L*#}) **and**
undef: *undefined-lit Ms L* **and**
L $\in$ *set C* **using** *find-first-unit-clause-some*[*OF unit*] **by** *metis*
have $dpll_W$ (*Ms*, *mset* (*map mset N*))
(*Propagated L* () # *fst* (*Ms*, *mset* (*map mset N*)), *snd* (*Ms*, *mset* (*map mset N*)))
apply (*rule* $dpll_W$*.propagate*)
using *Ms undef C* ‹*L* $\in$ *set C*› **by** (*auto simp add*: *C*)
then have *?thesis* **using** *Ms'N* **by** *auto*
}
moreover
{ assume *unit*: *find-first-unit-clause N Ms* = *None*
assume *exC*: $\exists$ *C* $\in$ *set N*. *Ms* $\models$*as CNot* (*mset C*)
then obtain *C* **where** *C*: *C* $\in$ *set N* **and** *Ms*: *Ms* $\models$*as CNot* (*mset C*) **by** *auto*
then obtain *L M M'* **where** *bt*: *backtrack-split Ms* = (*M'*, *L* # *M*)
using *step exC neq* **unfolding** *DPLL-step-def prod.case unit*
by (*cases backtrack-split Ms*, *rename-tac b*, *case-tac b*) *auto*

then have *is-decided L* **using** *backtrack-split-snd-hd-decided*[*of Ms*] **by** *auto*
have *1*: $dpll_W$ (*Ms*, *mset* (*map mset N*))
(*Propagated* (− *lit-of L*) () # *M*, *snd* (*Ms*, *mset* (*map mset N*)))
apply (*rule* $dpll_W$*.backtrack*[*OF* - ‹*is-decided L*›, *of*])
using *C Ms bt* **by** *auto*
moreover have (*Ms'*, *N'*) = (*Propagated* (− (*lit-of L*)) () # *M*, *N*)
using *step exC* **unfolding** *DPLL-step-def bt prod.case unit* **by** *auto*
ultimately have *?thesis* **by** *auto*
}
moreover
{ assume *unit*: *find-first-unit-clause N Ms* = *None*
assume *exC*: ¬ (∃ *C* ∈ *set N*. *Ms* ⊨*as CNot* (*mset C*))
obtain *L* **where** *unused*: *find-first-unused-var N* (*lits-of-l Ms*) = *Some L*
using *step exC neq* **unfolding** *DPLL-step-def prod.case unit*
by (*cases find-first-unused-var N* (*lits-of-l Ms*)) *auto*
have $dpll_W$ (*Ms*, *mset* (*map mset N*))
(*Decided L* # *fst* (*Ms*, *mset* (*map mset N*)), *snd* (*Ms*, *mset* (*map mset N*)))
apply (*rule* $dpll_W$*.decided*[*of ?S L*])
using *find-first-unused-var-Some*[*OF unused*]
by (*auto simp add*: *Decided-Propagated-in-iff-in-lits-of-l atms-of-ms-def*)
moreover have (*Ms'*, *N'*) = (*Decided L* # *Ms*, *N*)
using *step exC* **unfolding** *DPLL-step-def unused prod.case unit* **by** *auto*
ultimately have *?thesis* **by** *auto*
}
ultimately show *?thesis* **by** (*cases find-first-unit-clause N Ms*) *auto*
qed

lemma *DPLL-step-stuck-final-state*:
assumes *step*: (*Ms*, *N*) = *DPLL-step* (*Ms*, *N*)
shows *conclusive-*$dpll_W$*-state* (*toS Ms N*)
proof −
have *unit*: *find-first-unit-clause N Ms* = *None*
using *step* **unfolding** *DPLL-step-def* **by** (*auto split*:*option.splits*)

{ assume *n*: ∃ *C* ∈ *set N*. *Ms* ⊨*as CNot* (*mset C*)
then have *Ms*: (*Ms*, *N*) = (*case backtrack-split Ms of* (*x*, []) ⇒ (*Ms*, *N*)
| (*x*, *L* # *M*) ⇒ (*Propagated* (− *lit-of L*) () # *M*, *N*))
using *step* **unfolding** *DPLL-step-def* **by** (*simp add*:*unit*)

have *snd* (*backtrack-split Ms*) = []
proof (*cases backtrack-split Ms*, *cases snd* (*backtrack-split Ms*))
fix *a b*
assume *backtrack-split Ms* = (*a*, *b*) **and** *snd* (*backtrack-split Ms*) = []
then show *snd* (*backtrack-split Ms*) = [] **by** *blast*
next
fix *a b aa list*
assume
bt: *backtrack-split Ms* = (*a*, *b*) **and**
bt': *snd* (*backtrack-split Ms*) = *aa* # *list*
then have *Ms*: *Ms* = *Propagated* (− *lit-of aa*) () # *list* **using** *Ms* **by** *auto*
have *is-decided aa* **using** *backtrack-split-snd-hd-decided*[*of Ms*] *bt bt'* **by** *auto*
moreover have *fst* (*backtrack-split Ms*) @ *aa* # *list* = *Ms*
using *backtrack-split-list-eq*[*of Ms*] *bt'* **by** *auto*
ultimately have *False* **unfolding** *Ms* **by** *auto*
then show *snd* (*backtrack-split Ms*) = [] **by** *blast*
qed

then have *?thesis*
using *n backtrack-snd-empty-not-decided*[*of Ms*] **unfolding** *conclusive-dpll$_W$-state-def*
by (*cases backtrack-split Ms*) *auto*
}
moreover {
assume *n*: ¬ (∃ *C* ∈ *set N*. *Ms* ⊨*as CNot* (*mset C*))
then have *find-first-unused-var N* (*lits-of-l Ms*) = *None*
using *step* **unfolding** *DPLL-step-def* **by** (*simp add*: *unit split*: *option.splits*)
then have *a*: ∀ *a* ∈ *set N*. *atm-of* ‘ *set a* ⊆ *atm-of* ‘ (*lits-of-l Ms*) **by** *auto*
have *fst* (*toS Ms N*) ⊨*asm snd* (*toS Ms N*) **unfolding** *true-annots-def CNot-def Ball-def*
proof *clarify*
fix *x*
assume *x*: *x* ∈ *set-mset* (*clauses* (*toS Ms N*))
then have ¬*Ms* ⊨*as CNot x* **using** *n* **unfolding** *true-annots-def CNot-def Ball-def* **by** *auto*
moreover have *total-over-m* (*lits-of-l Ms*) {*x*}
using *a x image-iff in-mono atms-of-s-def*
unfolding *total-over-m-def total-over-set-def lits-of-def* **by** *fastforce*
ultimately show *fst* (*toS Ms N*) ⊨*a x*
using *total-not-CNot*[*of lits-of-l Ms x*] **by** (*simp add*: *true-annot-def true-annots-true-cls*)
qed
then have *?thesis* **unfolding** *conclusive-dpll$_W$-state-def* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

Adding invariants

Invariant tested in the function **function** *DPLL-ci* :: *int dpll$_W$-ann-lits* ⇒ *int literal list list* ⇒ *int dpll$_W$-ann-lits* × *int literal list list* **where**
DPLL-ci Ms N =
(*if* ¬*dpll$_W$-all-inv* (*Ms*, *mset* (*map mset N*))
then (*Ms*, *N*)
else
let (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*) *in*
if (*Ms'*, *N'*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci Ms' N*)
by *fast+*
termination
proof (*relation* {(*S'*, *S*). (*toS' S'*, *toS' S*) ∈ {(*S'*, *S*). *dpll$_W$-all-inv S* ∧ *dpll$_W$ S S'*}})
show *wf* {(*S'*, *S*).(*toS' S'*, *toS' S*) ∈ {(*S'*, *S*). *dpll$_W$-all-inv S* ∧ *dpll$_W$ S S'*}}
using *wf-if-measure-f*[*OF dpll$_W$-wf*, *of toS'*] **by** *auto*
next
fix *Ms* :: *int dpll$_W$-ann-lits* **and** *N x xa y*
assume¬ ¬ *dpll$_W$-all-inv* (*toS Ms N*)
and *step*: *x* = *DPLL-step* (*Ms*, *N*)
and *x*: (*xa*, *y*) = *x*
and (*xa*, *y*) ≠ (*Ms*, *N*)
then show ((*xa*, *N*), *Ms*, *N*) ∈ {(*S'*, *S*). (*toS' S'*, *toS' S*) ∈ {(*S'*, *S*). *dpll$_W$-all-inv S* ∧ *dpll$_W$ S S'*}}
using *DPLL-step-is-a-dpll$_W$-step dpll$_W$-same-clauses split-conv* **by** *fastforce*
qed

No invariant tested **function** (*domintros*) *DPLL-part*:: *int dpll$_W$-ann-lits* ⇒ *int literal list list* ⇒ *int dpll$_W$-ann-lits* × *int literal list list* **where**
DPLL-part Ms N =
(*let* (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*) *in*
if (*Ms'*, *N'*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-part Ms' N*)

by *fast+*

lemma *snd-DPLL-step*[*simp*]:
  snd (*DPLL-step* (*Ms*, *N*)) = *N*
  unfolding *DPLL-step-def* **by** (*auto split*: *if-split option.splits prod.splits list.splits*)

lemma *dpll*$_W$*-all-inv-implieS-2-eq3-and-dom*:
  assumes *dpll*$_W$*-all-inv* (*Ms*, *mset* (*map mset N*))
  shows *DPLL-ci Ms N* = *DPLL-part Ms N* ∧ *DPLL-part-dom* (*Ms*, *N*)
  using *assms*
proof (*induct rule*: *DPLL-ci.induct*)
  case (*1 Ms N*)
  have *snd* (*DPLL-step* (*Ms*, *N*)) = *N* **by** *auto*
  then obtain *Ms'* **where** *Ms'*: *DPLL-step* (*Ms*, *N*) = (*Ms'*, *N*) **by** (*cases DPLL-step* (*Ms*, *N*)) *auto*
  have *inv'*: *dpll*$_W$*-all-inv* (*toS Ms' N*) **by** (*metis* (*mono-tags*) *1.prems DPLL-step-is-a-dpll*$_W$*-step*
    Ms' dpll$_W$*-all-inv old.prod.inject*)
  { assume (*Ms'*, *N*) ≠ (*Ms*, *N*)
    then have *DPLL-ci Ms' N* = *DPLL-part Ms' N* ∧ *DPLL-part-dom* (*Ms'*, *N*) **using** *1*(*1*)[*of - Ms'*
N] *Ms'*
      1(*2*) *inv'* **by** *auto*
    then have *DPLL-part-dom* (*Ms*, *N*) **using** *DPLL-part.domintros Ms'* **by** *fastforce*
    moreover have *DPLL-ci Ms N* = *DPLL-part Ms N* **using** *1.prems DPLL-part.psimps Ms'*
      ‹*DPLL-ci Ms' N* = *DPLL-part Ms' N* ∧ *DPLL-part-dom* (*Ms'*, *N*)› ‹*DPLL-part-dom* (*Ms*, *N*)› **by**
auto
    ultimately have *?case* **by** *blast*
  }
  moreover {
    assume (*Ms'*, *N*) = (*Ms*, *N*)
    then have *?case* **using** *DPLL-part.domintros DPLL-part.psimps Ms'* **by** *fastforce*
  }
  ultimately show *?case* **by** *blast*
qed

lemma *DPLL-ci-dpll*$_W$*-rtranclp*:
  assumes *DPLL-ci Ms N* = (*Ms'*, *N'*)
  shows *dpll*$_W$** (*toS Ms N*) (*toS Ms' N*)
  using *assms*
proof (*induct Ms N arbitrary*: *Ms' N' rule*: *DPLL-ci.induct*)
  case (*1 Ms N Ms' N'*) **note** *IH* = *this*(*1*) **and** *step* = *this*(*2*)
  obtain S_1 S_2 **where** *S*: (S_1, S_2) = *DPLL-step* (*Ms*, *N*) **by** (*cases DPLL-step* (*Ms*, *N*)) *auto*

  { assume ¬*dpll*$_W$*-all-inv* (*toS Ms N*)
    then have (*Ms*, *N*) = (*Ms'*, *N*) **using** *step* **by** *auto*
    then have *?case* **by** *auto*
  }
  moreover
  { assume *dpll*$_W$*-all-inv* (*toS Ms N*)
    and (S_1, S_2) = (*Ms*, *N*)
    then have *?case* **using** *S step* **by** *auto*
  }
  moreover
  { assume *dpll*$_W$*-all-inv* (*toS Ms N*)
    and (S_1, S_2) ≠ (*Ms*, *N*)
    moreover obtain $S_1{'}$ $S_2{'}$ **where** *DPLL-ci* S_1 *N* = ($S_1{'}$, $S_2{'}$) **by** (*cases DPLL-ci* S_1 *N*) *auto*
    moreover have *DPLL-ci Ms N* = *DPLL-ci* S_1 *N* **using** *DPLL-ci.simps*[*of Ms N*] *calculation*
      proof –

have (*case* (S_1, S_2) *of* (*ms*, *lss*) $\Rightarrow$
if (*ms*, *lss*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci ms N*) = *DPLL-ci Ms N*
using *S DPLL-ci.simps*[*of Ms N*] *calculation* **by** *presburger*
then have (*if* (S_1, S_2) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci* S_1 *N*) = *DPLL-ci Ms N*
by *fastforce*
then show *?thesis*
using *calculation*(*2*) **by** *presburger*
qed
ultimately have $dpll_W^{**}$ (*toS* S_1' *N*) (*toS Ms' N*) **using** *IH*[*of* (S_1, S_2) S_1 S_2] *S step* **by** *simp*

moreover have $dpll_W$ (*toS Ms N*) (*toS* S_1 *N*)
by (*metis DPLL-step-is-a-*$dpll_W$*-step S* ‹$(S_1, S_2) \neq$ (*Ms*, *N*)› *prod.sel*(*2*) *snd-DPLL-step*)
ultimately have *?case* **by** (*metis* (*mono-tags*, *hide-lams*) *IH S* ‹$(S_1, S_2) \neq$ (*Ms*, *N*)›
‹*DPLL-ci Ms N* = *DPLL-ci* S_1 *N*› ‹$dpll_W$*-all-inv* (*toS Ms N*)› *converse-rtranclp-into-rtranclp*
local.step)
}
ultimately show *?case* **by** *blast*
qed

lemma $dpll_W$*-all-inv-*$dpll_W$*-tranclp-irrefl*:
assumes $dpll_W$*-all-inv* (*Ms*, *N*)
and $dpll_W^{++}$ (*Ms*, *N*) (*Ms*, *N*)
shows *False*
proof –
have *1*: *wf* {(*S'*, *S*). $dpll_W$*-all-inv S* $\wedge$ $dpll_W^{++}$ *S S'*} **using** $dpll_W$*-wf-tranclp* **by** *auto*
have ((*Ms*, *N*), (*Ms*, *N*)) $\in$ {(*S'*, *S*). $dpll_W$*-all-inv S* $\wedge$ $dpll_W^{++}$ *S S'*} **using** *assms* **by** *auto*
then show *False* **using** *wf-not-refl*[*OF 1*] **by** *blast*
qed

lemma *DPLL-ci-final-state*:
assumes *step*: *DPLL-ci Ms N* = (*Ms*, *N*)
and *inv*: $dpll_W$*-all-inv* (*toS Ms N*)
shows *conclusive-*$dpll_W$*-state* (*toS Ms N*)
proof –
have *st*: $dpll_W^{**}$ (*toS Ms N*) (*toS Ms N*) **using** *DPLL-ci-*$dpll_W$*-rtranclp*[*OF step*] **.**
have *DPLL-step* (*Ms*, *N*) = (*Ms*, *N*)
proof (*rule ccontr*)
obtain *Ms' N'* **where** *Ms'N*: (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*)
by (*cases DPLL-step* (*Ms*, *N*)) *auto*
assume ¬ *?thesis*
then have *DPLL-ci Ms' N* = (*Ms*, *N*) **using** *step inv st Ms'N*[*symmetric*] **by** *fastforce*
then have $dpll_W^{++}$ (*toS Ms N*) (*toS Ms N*)
by (*metis DPLL-ci-*$dpll_W$*-rtranclp DPLL-step-is-a-*$dpll_W$*-step Ms'N* ‹*DPLL-step* (*Ms*, *N*) $\neq$ (*Ms*,
N)›
prod.sel(*2*) *rtranclp-into-tranclp2 snd-DPLL-step*)
then show *False* **using** $dpll_W$*-all-inv-*$dpll_W$*-tranclp-irrefl inv* **by** *auto*
qed
then show *?thesis* **using** *DPLL-step-stuck-final-state*[*of Ms N*] **by** *simp*
qed

lemma *DPLL-step-obtains*:
obtains *Ms'* **where** (*Ms'*, *N*) = *DPLL-step* (*Ms*, *N*)
unfolding *DPLL-step-def* **by** (*metis* (*no-types*, *lifting*) *DPLL-step-def prod.collapse snd-DPLL-step*)

lemma *DPLL-ci-obtains*:
obtains *Ms'* **where** (*Ms'*, *N*) = *DPLL-ci Ms N*

proof (*induct rule*: *DPLL-ci.induct*)
  case (*1 Ms N*) **note** *IH* = *this*(*1*) **and** *that* = *this*(*2*)
  obtain *S* **where** *SN*: (*S*, *N*) = *DPLL-step* (*Ms*, *N*) **using** *DPLL-step-obtains* **by** *metis*
  { assume ¬ $dpll_W$*-all-inv* (*toS Ms N*)
    then have *?case* **using** *that* **by** *auto*
  }
  moreover {
    assume *n*: (*S*, *N*) ≠ (*Ms*, *N*)
    and *inv*: $dpll_W$*-all-inv* (*toS Ms N*)
    have ∃ *ms*. *DPLL-step* (*Ms*, *N*) = (*ms*, *N*)
      by (*metis* ‹⋀*thesisa*. (⋀*S*. (*S*, *N*) = *DPLL-step* (*Ms*, *N*) ⟹ *thesisa*) ⟹ *thesisa*›)
    then have *?thesis*
      using *IH that* **by** *fastforce*
  }
  moreover {
    assume *n*: (*S*, *N*) = (*Ms*, *N*)
    then have *?case* **using** *SN that* **by** *fastforce*
  }
  ultimately show *?case* **by** *blast*
qed

lemma *DPLL-ci-no-more-step*:
  assumes *step*: *DPLL-ci Ms N* = (*Ms'*, *N'*)
  shows *DPLL-ci Ms' N'* = (*Ms'*, *N'*)
  using *assms*
proof (*induct arbitrary*: *Ms' N' rule*: *DPLL-ci.induct*)
  case (*1 Ms N Ms' N'*) **note** *IH* = *this*(*1*) **and** *step* = *this*(*2*)
  obtain S_1 **where** *S*: (S_1, *N*) = *DPLL-step* (*Ms*, *N*) **using** *DPLL-step-obtains* **by** *auto*
  { assume ¬$dpll_W$*-all-inv* (*toS Ms N*)
    then have *?case* **using** *step* **by** *auto*
  }
  moreover {
    assume $dpll_W$*-all-inv* (*toS Ms N*)
    and (S_1, *N*) = (*Ms*, *N*)
    then have *?case* **using** *S step* **by** *auto*
  }
  moreover
  { assume *inv*: $dpll_W$*-all-inv* (*toS Ms N*)
    assume *n*: (S_1, *N*) ≠ (*Ms*, *N*)
    obtain S_1' **where** *SS*: (S_1', *N*) = *DPLL-ci* S_1 *N* **using** *DPLL-ci-obtains* **by** *blast*
    moreover have *DPLL-ci Ms N* = *DPLL-ci* S_1 *N*
      proof −
        have (*case* (S_1, *N*) *of* (*ms*, *lss*) ⇒ *if* (*ms*, *lss*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci ms N*)
 = *DPLL-ci Ms N*
          using *S DPLL-ci.simps*[*of Ms N*] *calculation inv* **by** *presburger*
        then have (*if* (S_1, *N*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci* S_1 *N*) = *DPLL-ci Ms N*
          by *fastforce*
        then show *?thesis*
          using *calculation n* **by** *presburger*
      qed
    moreover
      have *DPLL-ci* S_1' *N* = (S_1', *N*) **using** *step IH*[*OF - - S n SS*[*symmetric*]] *inv* **by** *blast*
    ultimately have *?case* **using** *step* **by** *fastforce*
  }
  ultimately show *?case* **by** *blast*

qed

lemma *DPLL-part-dpll$_W$-all-inv-final*:
fixes *M Ms'*:: (*int*, *unit*) *ann-lits* **and**
N :: *int literal list list*
assumes *inv*: *dpll$_W$-all-inv* (*Ms*, *mset* (*map mset N*))
and *MsN*: *DPLL-part Ms N* = (*Ms'*, *N*)
shows *conclusive-dpll$_W$-state* (*toS Ms' N*) ∧ *dpll$_W$*** (*toS Ms N*) (*toS Ms' N*)
proof −
have *2*: *DPLL-ci Ms N* = *DPLL-part Ms N* **using** *inv dpll$_W$-all-inv-implieS-2-eq3-and-dom* **by** *blast*
then have *star*: *dpll$_W$*** (*toS Ms N*) (*toS Ms' N*) **unfolding** *MsN* **using** *DPLL-ci-dpll$_W$-rtranclp* **by** *blast*
then have *inv'*: *dpll$_W$-all-inv* (*toS Ms' N*) **using** *inv rtranclp-dpll$_W$-all-inv* **by** *blast*
show *?thesis* **using** *star DPLL-ci-final-state*[*OF DPLL-ci-no-more-step inv'*] *2* **unfolding** *MsN* **by** *blast*
qed

Embedding the invariant into the type

Defining the type **typedef** *dpll$_W$-state* =
{(*M*::(*int*, *unit*) *ann-lits*, *N*::*int literal list list*).
dpll$_W$-all-inv (*toS M N*)}
morphisms *rough-state-of state-of*
proof
show ([],[]) ∈ {(*M*, *N*). *dpll$_W$-all-inv* (*toS M N*)} **by** (*auto simp add*: *dpll$_W$-all-inv-def*)
qed

lemma
DPLL-part-dom ([], *N*)
using *assms dpll$_W$-all-inv-implieS-2-eq3-and-dom*[*of* [] *N*] **by** (*simp add*: *dpll$_W$-all-inv-def*)

Some type classes **instantiation** *dpll$_W$-state* :: *equal*
begin
definition *equal-dpll$_W$-state* :: *dpll$_W$-state* ⇒ *dpll$_W$-state* ⇒ *bool* **where**
equal-dpll$_W$-state S S' = (*rough-state-of S* = *rough-state-of S'*)
instance
by *standard* (*simp add*: *rough-state-of-inject equal-dpll$_W$-state-def*)
end

DPLL **definition** *DPLL-step'* :: *dpll$_W$-state* ⇒ *dpll$_W$-state* **where**
DPLL-step' S = *state-of* (*DPLL-step* (*rough-state-of S*))

declare *rough-state-of-inverse*[*simp*]

lemma *DPLL-step-dpll$_W$-conc-inv*:
DPLL-step (*rough-state-of S*) ∈ {(*M*, *N*). *dpll$_W$-all-inv* (*toS M N*)}
by (*smt DPLL-ci.simps DPLL-ci-dpll$_W$-rtranclp case-prodE case-prodI2 rough-state-of mem-Collect-eq old.prod.case prod.sel*(*2*) *rtranclp-dpll$_W$-all-inv snd-DPLL-step*)

lemma *rough-state-of-DPLL-step'-DPLL-step*[*simp*]:
rough-state-of (*DPLL-step' S*) = *DPLL-step* (*rough-state-of S*)
using *DPLL-step-dpll$_W$-conc-inv DPLL-step'-def state-of-inverse* **by** *auto*

function *DPLL-tot*:: *dpll$_W$-state* ⇒ *dpll$_W$-state* **where**

DPLL-tot S =
  (*let S′ = DPLL-step′ S in*
   if S′ = S then S else DPLL-tot S′)
  by *fast+*
termination
proof (*relation* {(*T′*, *T*).
      (*rough-state-of T′*, *rough-state-of T*)
         ∈ {(*S′*, *S*). (*toS′ S′*, *toS′ S*)
              ∈ {(*S′*, *S*). $dpll_W$*-all-inv S* ∧ $dpll_W$ *S S′*}}})
  show *wf* {(*b*, *a*).
          (*rough-state-of b*, *rough-state-of a*)
            ∈ {(*b*, *a*). (*toS′ b*, *toS′ a*)
              ∈ {(*b*, *a*). $dpll_W$*-all-inv a* ∧ $dpll_W$ *a b*}}}
    using *wf-if-measure-f*[*OF wf-if-measure-f*[*OF* $dpll_W$*-wf*, *of toS′*], *of rough-state-of*] **.**
next
  fix *S x*
  assume *x*: *x* = *DPLL-step′ S*
  and *x* ≠ *S*
  have $dpll_W$*-all-inv* (*case rough-state-of S of* (*Ms*, *N*) ⇒ (*Ms*, *mset* (*map mset N*)))
    by (*metis* (*no-types*, *lifting*) *case-prodE mem-Collect-eq old.prod.case rough-state-of*)
  moreover have $dpll_W$ (*case rough-state-of S of* (*Ms*, *N*) ⇒ (*Ms*, *mset* (*map mset N*)))
                      (*case rough-state-of* (*DPLL-step′ S*) *of* (*Ms*, *N*) ⇒ (*Ms*, *mset* (*map mset N*)))
    proof −
      obtain *Ms N* **where** *Ms*: (*Ms*, *N*) = *rough-state-of S* **by** (*cases rough-state-of S*) *auto*
      have $dpll_W$*-all-inv* (*toS′* (*Ms*, *N*)) **using** *calculation* **unfolding** *Ms* **by** *blast*
      moreover obtain *Ms′ N′* **where** *Ms′*: (*Ms′*, *N′*) = *rough-state-of* (*DPLL-step′ S*)
        by (*cases rough-state-of* (*DPLL-step′ S*)) *auto*
      ultimately have $dpll_W$*-all-inv* (*toS′* (*Ms′*, *N′*)) **unfolding** *Ms′*
        by (*metis* (*no-types*, *lifting*) *case-prod-unfold mem-Collect-eq rough-state-of*)

      have $dpll_W$ (*toS Ms N*) (*toS Ms′ N′*)
        apply (*rule DPLL-step-is-a-*$dpll_W$*-step*[*of Ms′ N′ Ms N*])
        unfolding *Ms Ms′* **using** ‹*x* ≠ *S*› *rough-state-of-inject x* **by** *fastforce+*
      then show *?thesis* **unfolding** *Ms*[*symmetric*] *Ms′*[*symmetric*] **by** *auto*
    qed
  ultimately show (*x*, *S*) ∈ {(*T′*, *T*). (*rough-state-of T′*, *rough-state-of T*)
    ∈ {(*S′*, *S*). (*toS′ S′*, *toS′ S*) ∈ {(*S′*, *S*). $dpll_W$*-all-inv S* ∧ $dpll_W$ *S S′*}}}
    by (*auto simp add*: *x*)
qed

lemma [*code*]:
DPLL-tot S =
  (*let S′ = DPLL-step′ S in*
   if S′ = S then S else DPLL-tot S′) **by** *auto*

lemma *DPLL-tot-DPLL-step-DPLL-tot*[*simp*]: *DPLL-tot* (*DPLL-step′ S*) = *DPLL-tot S*
  apply (*cases DPLL-step′ S = S*)
  apply *simp*
  unfolding *DPLL-tot.simps*[*of S*] **by** (*simp del*: *DPLL-tot.simps*)

lemma *DOPLL-step′-DPLL-tot*[*simp*]:
  DPLL-step′ (*DPLL-tot S*) = *DPLL-tot S*
  by (*rule DPLL-tot.induct*[*of λS. DPLL-step′* (*DPLL-tot S*) = *DPLL-tot S S*])
     (*metis* (*full-types*) *DPLL-tot.simps*)

lemma *DPLL-tot-final-state*:
assumes *DPLL-tot S = S*
shows *conclusive-dpll$_W$-state* (*toS'* (*rough-state-of S*))
proof −
have *DPLL-step' S = S* **using** *assms*[*symmetric*] *DOPLL-step'-DPLL-tot* **by** *metis*
then have *DPLL-step* (*rough-state-of S*) = (*rough-state-of S*)
unfolding *DPLL-step'-def* **using** *DPLL-step-dpll$_W$-conc-inv rough-state-of-inverse*
by (*metis rough-state-of-DPLL-step'-DPLL-step*)
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *DPLL-step-stuck-final-state old.prod.exhaust split-conv*)
qed

lemma *DPLL-tot-star*:
assumes *rough-state-of* (*DPLL-tot S*) = *S'*
shows *dpll$_W$*** (*toS'* (*rough-state-of S*)) (*toS' S'*)
using *assms*
proof (*induction arbitrary*: *S' rule*: *DPLL-tot.induct*)
case (*1 S S'*)
let *?x = DPLL-step' S*
{ assume *?x = S*
then have *?case* **using** *1*(*2*) **by** *simp*
}
moreover {
assume *S*: *?x ≠ S*
have *?case*
apply (*cases DPLL-step' S = S*)
using *S* **apply** *blast*
by (*smt 1.IH 1.prems DPLL-step-is-a-dpll$_W$-step DPLL-tot.simps case-prodE2 rough-state-of-DPLL-step'-DPLL-step rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl rtranclp-idemp split-conv*)
}
ultimately show *?case* **by** *auto*
qed

lemma *rough-state-of-rough-state-of-Nil*[*simp*]:
rough-state-of (*state-of* ([], *N*)) = ([], *N*)
apply (*rule DPLL-W-Implementation.dpll$_W$-state.state-of-inverse*)
unfolding *dpll$_W$-all-inv-def* **by** *auto*

Theorem of correctness

lemma *DPLL-tot-correct*:
assumes *rough-state-of* (*DPLL-tot* (*state-of* (([], *N*)))) = (*M*, *N'*)
and (*M'*, *N''*) = *toS'* (*M*, *N'*)
shows *M'* ⊨*asm N''* ⟷ *satisfiable* (*set-mset N''*)
proof −
have *dpll$_W$*** (*toS'* ([], *N*)) (*toS'* (*M*, *N'*)) **using** *DPLL-tot-star*[*OF assms*(*1*)] **by** *auto*
moreover have *conclusive-dpll$_W$-state* (*toS'* (*M*, *N'*))
using *DPLL-tot-final-state* **by** (*metis* (*mono-tags*, *lifting*) *DOPLL-step'-DPLL-tot DPLL-tot.simps assms*(*1*))
ultimately show *?thesis* **using** *dpll$_W$-conclusive-state-correct* **by** (*smt DPLL-ci.simps DPLL-ci-dpll$_W$-rtranclp assms*(*2*) *dpll$_W$-all-inv-def prod.case prod.sel*(*1*) *prod.sel*(*2*) *rtranclp-dpll$_W$-inv*(*3*) *rtranclp-dpll$_W$-inv-starting-from-0*)
qed

Code export

A conversion to *DPLL-W-Implementation.dpll$_W$-state* **definition** *Con* :: (*int*, *unit*) *ann-lits* × *int literal list list*
⇒ *dpll$_W$-state* **where**
Con xs = *state-of* (*if dpll$_W$-all-inv* (*toS* (*fst xs*) (*snd xs*)) *then xs else* ([], []))
lemma [*code abstype*]:
Con (*rough-state-of S*) = *S*
using *rough-state-of*[*of S*] **unfolding** *Con-def* **by** *auto*

declare *rough-state-of-DPLL-step′-DPLL-step*[*code abstract*]

lemma *Con-DPLL-step-rough-state-of-state-of*[*simp*]:
Con (*DPLL-step* (*rough-state-of s*)) = *state-of* (*DPLL-step* (*rough-state-of s*))
unfolding *Con-def* **by** (*metis* (*mono-tags*, *lifting*) *DPLL-step-dpll$_W$-conc-inv mem-Collect-eq prod.case-eq-if*)

A slightly different version of *DPLL-tot* where the returned boolean indicates the result.

definition *DPLL-tot-rep* **where**
DPLL-tot-rep S =
(*let* (*M*, *N*) = (*rough-state-of* (*DPLL-tot S*)) *in* (∀ *A* ∈ *set N*. (∃ *a*∈*set A*. *a* ∈ *lits-of-l* (*M*)), *M*))

One version of the generated SML code is here, but not included in the generated document. The only differences are:

- export *′a literal* from the SML Module *Clausal-Logic*;
- export the constructor *Con* from *DPLL-W-Implementation*;
- export the *int* constructor from *Arith*.

  All these allows to test on the code on some examples.

end
theory *CDCL-W-Implementation*
imports *DPLL-CDCL-W-Implementation CDCL-W-Termination*
begin

7.1.4 List-based CDCL Implementation

We here have a very simple implementation of Weidenbach's CDCL, based on the same principle as the implementation of DPLL: iterating over-and-over on lists. We do not use any fancy data-structure (see the two-watched literals for a better suited data-structure).

The goal was (as for DPLL) to test the infrastructure and see if an important lemma was missing to prove the correctness and the termination of a simple implementation.

Types and Instantiation

notation *image-mset* (**infixr** ‘# *90*)

type-synonym *′a cdcl$_W$-mark* = *′a clause*

type-synonym *′v cdcl$_W$-ann-lit* = (*′v*, *′v cdcl$_W$-mark*) *ann-lit*
type-synonym *′v cdcl$_W$-ann-lits* = (*′v*, *′v cdcl$_W$-mark*) *ann-lits*
type-synonym *′v cdcl$_W$-state* =

$'v\ cdcl_W\text{-}ann\text{-}lits \times 'v\ clauses \times 'v\ clauses \times nat \times 'v\ clause\ option$

abbreviation *raw-trail* $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a$ **where**
raw-trail $\equiv (\lambda(M, \text{-}).\ M)$

abbreviation *raw-cons-trail* $:: 'a \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$ **where**
raw-cons-trail $\equiv (\lambda L\ (M, S).\ (L\#M, S))$

abbreviation *raw-tl-trail* $:: 'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$ **where**
raw-tl-trail $\equiv (\lambda(M, S).\ (tl\ M, S))$

abbreviation *raw-init-clss* $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b$ **where**
raw-init-clss $\equiv \lambda(M, N, \text{-}).\ N$

abbreviation *raw-learned-clss* $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c$ **where**
raw-learned-clss $\equiv \lambda(M, N, U, \text{-}).\ U$

abbreviation *raw-backtrack-lvl* $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd$ **where**
raw-backtrack-lvl $\equiv \lambda(M, N, U, k, \text{-}).\ k$

abbreviation *raw-update-backtrack-lvl* $:: 'd \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$ **where**
raw-update-backtrack-lvl $\equiv \lambda k\ (M, N, U, \text{-}, S).\ (M, N, U, k, S)$

abbreviation *raw-conflicting* $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$ **where**
raw-conflicting $\equiv \lambda(M, N, U, k, D).\ D$

abbreviation *raw-update-conflicting* $:: 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$ **where**
raw-update-conflicting $\equiv \lambda S\ (M, N, U, k, \text{-}).\ (M, N, U, k, S)$

abbreviation $S0\text{-}cdcl_W\ N \equiv (([], N, \{\#\}, 0, None) :: 'v\ cdcl_W\text{-}state)$

abbreviation *raw-add-learned-clss* **where**
raw-add-learned-clss $\equiv \lambda C\ (M, N, U, S).\ (M, N, \{\#C\#\} + U, S)$

abbreviation *raw-remove-cls* **where**
raw-remove-cls $\equiv \lambda C\ (M, N, U, S).\ (M, removeAll\text{-}mset\ C\ N, removeAll\text{-}mset\ C\ U, S)$

lemma *raw-trail-conv*: *raw-trail* $(M, N, U, k, D) = M$ **and**
clauses-conv: *raw-init-clss* $(M, N, U, k, D) = N$ **and**
raw-learned-clss-conv: *raw-learned-clss* $(M, N, U, k, D) = U$ **and**
raw-conflicting-conv: *raw-conflicting* $(M, N, U, k, D) = D$ **and**
raw-backtrack-lvl-conv: *raw-backtrack-lvl* $(M, N, U, k, D) = k$
by *auto*

lemma *state-conv*:
$S = (raw\text{-}trail\ S, raw\text{-}init\text{-}clss\ S, raw\text{-}learned\text{-}clss\ S, raw\text{-}backtrack\text{-}lvl\ S, raw\text{-}conflicting\ S)$
by (*cases S*) *auto*

interpretation $state_W$
raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting
$\lambda L\ (M, S).\ (L\ \#\ M, S)$
$\lambda(M, S).\ (tl\ M, S)$

```
  λC (M, N, U, S). (M, N, {#C#} + U, S)
  λC (M, N, U, S). (M, removeAll-mset C N, removeAll-mset C U, S)
  λ(k::nat) (M, N, U, -, D). (M, N, U, k, D)
  λD (M, N, U, k, -). (M, N, U, k, D)
  λN. ([], N, {#}, 0, None)
  by unfold-locales auto

interpretation conflict-driven-clause-learning_W raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl
raw-conflicting
  λL (M, S). (L # M, S)
  λ(M, S). (tl M, S)
  λC (M, N, U, S). (M, N, {#C#} + U, S)
  λC (M, N, U, S). (M, removeAll-mset C N, removeAll-mset C U, S)
  λ(k::nat) (M, N, U, -, D). (M, N, U, k, D)
  λD (M, N, U, k, -). (M, N, U, k, D)
  λN. ([], N, {#}, 0, None)
  by unfold-locales auto

declare clauses-def[simp]

lemma cdcl_W-state-eq-equality[iff]: state-eq S T ⟷ S = T
  unfolding state-eq-def by (cases S, cases T) auto
declare state-simp[simp del]

lemma reduce-trail-to-empty-trail[simp]:
  reduce-trail-to F ([], aa, ab, ac, b) = ([], aa, ab, ac, b)
  using reduce-trail-to.simps by auto

lemma raw-trail-reduce-trail-to-length-le:
  assumes length F > length (raw-trail S)
  shows raw-trail (reduce-trail-to F S) = []
  using assms trail-reduce-trail-to-length-le[of S F]
  by (cases S, cases reduce-trail-to F S) auto

lemma reduce-trail-to:
  reduce-trail-to F S =
    ((if length (raw-trail S) ≥ length F
    then drop (length (raw-trail S) − length F) (raw-trail S)
    else []), raw-init-clss S, raw-learned-clss S, raw-backtrack-lvl S, raw-conflicting S)
    (is ?S = -)
proof (induction F S rule: reduce-trail-to.induct)
  case (1 F S) note IH = this
  show ?case
    proof (cases raw-trail S)
      case Nil
      then show ?thesis using IH by (cases S) auto
    next
      case (Cons L M)
      then show ?thesis
        apply (cases Suc (length M) > length F)
         prefer 2 using IH reduce-trail-to-length-ne[of S F] apply (cases S) apply auto[]
        apply (subgoal-tac Suc (length M) − length F = Suc (length M − length F))
        using reduce-trail-to-length-ne[of S F] IH by (cases S) auto
    qed
qed
```

7.1.5 CDCL Implementation

Definition of the rules

Types **lemma** *true-raw-init-clss-remdups*[*simp*]:
  $I \models s$ (*mset* ∘ *remdups*) ‘ $N \longleftrightarrow I \models s$ *mset* ‘ N
  by (*simp add*: *true-clss-def*)

lemma *satisfiable-mset-remdups*[*simp*]:
  satisfiable ((*mset* ∘ *remdups*) ‘ N) ⟷ *satisfiable* (*mset* ‘ N)
unfolding *satisfiable-carac*[*symmetric*] **by** *simp*

type-synonym *'v* $cdcl_W$*-state-inv-st* = (*'v*, *'v literal list*) *ann-lit list* ×
  'v literal list list × *'v literal list list* × *nat* × *'v literal list option*

We need some functions to convert between our abstract state *'v* $cdcl_W$*-state* and the concrete state *'v* $cdcl_W$*-state-inv-st*.

fun *convert* :: (*'a*, *'c list*) *ann-lit* ⇒ (*'a*, *'c multiset*) *ann-lit* **where**
convert (*Propagated L C*) = *Propagated L* (*mset C*) |
convert (*Decided K*) = *Decided K*

abbreviation *convertC* :: *'a list option* ⇒ *'a multiset option* **where**
convertC ≡ *map-option mset*

lemma *convert-Propagated*[*elim*!]:
  convert z = *Propagated L C* ⟹ (∃ C'. *z* = *Propagated L* C' ∧ *C* = *mset* C')
  by (*cases z*) *auto*

lemma *is-decided-convert*[*simp*]: *is-decided* (*convert x*) = *is-decided x*
  by (*cases x*) *auto*

lemma *get-level-map-convert*[*simp*]:
  get-level (*map convert M*) *x* = *get-level M x*
  by (*induction M rule*: *ann-lit-list-induct*) (*auto simp*: *comp-def*)

lemma *get-maximum-level-map-convert*[*simp*]:
  get-maximum-level (*map convert M*) *D* = *get-maximum-level M D*
  by (*induction D*)
    (*auto simp add*: *get-maximum-level-plus*)

Conversion function

fun *toS* :: *'v* $cdcl_W$*-state-inv-st* ⇒ *'v* $cdcl_W$*-state* **where**
toS (*M*, *N*, *U*, *k*, *C*) = (*map convert M*, *mset* (*map mset N*), *mset* (*map mset U*), *k*, *convertC C*)

Definition an abstract type

typedef *'v* $cdcl_W$*-state-inv* = {*S*::*'v* $cdcl_W$*-state-inv-st*. $cdcl_W$*-all-struct-inv* (*toS S*)}
  morphisms *rough-state-of state-of*
proof
  show ([],[], [], *0*, *None*) ∈ {*S*. $cdcl_W$*-all-struct-inv* (*toS S*)}
    by (*auto simp add*: $cdcl_W$*-all-struct-inv-def*)
qed

instantiation $cdcl_W$*-state-inv* :: (*type*) *equal*
begin
definition *equal-*$cdcl_W$*-state-inv* :: *'v* $cdcl_W$*-state-inv* ⇒ *'v* $cdcl_W$*-state-inv* ⇒ *bool* **where**

equal-cdcl$_W$-state-inv S S′ = (*rough-state-of S* = *rough-state-of S′*)
instance
  by *standard* (*simp add*: *rough-state-of-inject equal-cdcl$_W$-state-inv-def*)
end

lemma *lits-of-map-convert*[*simp*]: *lits-of-l* (*map convert M*) = *lits-of-l M*
  by (*induction M rule*: *ann-lit-list-induct*) *simp-all*

lemma *atm-lit-of-convert*[*simp*]:
  lit-of (*convert x*) = *lit-of x*
  by (*cases x*) *auto*

lemma *undefined-lit-map-convert*[*iff*]:
  undefined-lit (*map convert M*) *L* ⟷ *undefined-lit M L*
  by (*auto simp add*: *defined-lit-map image-image*)

lemma *true-annot-map-convert*[*simp*]: *map convert M* ⊨*a N* ⟷ *M* ⊨*a N*
  by (*simp-all add*: *true-annot-def image-image lits-of-def*)

lemma *true-annots-map-convert*[*simp*]: *map convert M* ⊨*as N* ⟷ *M* ⊨*as N*
  unfolding *true-annots-def* **by** *auto*

lemmas *propagateE*
lemma *find-first-unit-clause-some-is-propagate*:
  assumes *H*: *find-first-unit-clause* (*N* @ *U*) *M* = *Some* (*L*, *C*)
  shows *propagate* (*toS* (*M*, *N*, *U*, *k*, *None*)) (*toS* (*Propagated L C* # *M*, *N*, *U*, *k*, *None*))
  using *assms*
  by (*auto dest*!: *find-first-unit-clause-some simp add*: *propagate.simps*
    intro!: *exI*[*of* - *mset C* − {#*L*#}])

The Transitions

Propagate **definition** *do-propagate-step* **where**
do-propagate-step S =
  (*case S of*
    (*M*, *N*, *U*, *k*, *None*) ⇒
      (*case find-first-unit-clause* (*N* @ *U*) *M of*
        Some (*L*, *C*) ⇒ (*Propagated L C* # *M*, *N*, *U*, *k*, *None*)
      | *None* ⇒ (*M*, *N*, *U*, *k*, *None*))
  | *S* ⇒ *S*)

lemma *do-propgate-step*:
  do-propagate-step S ≠ *S* ⟹ *propagate* (*toS S*) (*toS* (*do-propagate-step S*))
  apply (*cases S*, *cases raw-conflicting S*)
  using *find-first-unit-clause-some-is-propagate*[*of raw-init-clss S raw-learned-clss S raw-trail S* - -
    raw-backtrack-lvl S]
  by (*auto simp add*: *do-propagate-step-def split*: *option.splits*)

lemma *do-propagate-step-option*[*simp*]:
  raw-conflicting S ≠ *None* ⟹ *do-propagate-step S* = *S*
  unfolding *do-propagate-step-def* **by** (*cases S*, *cases raw-conflicting S*) *auto*

lemma *do-propagate-step-no-step*:
  assumes *dist*: ∀ *c*∈*set* (*raw-init-clss S* @ *raw-learned-clss S*). *distinct c* **and**
  prop-step: *do-propagate-step S* = *S*
  shows *no-step propagate* (*toS S*)

proof (*standard*, *standard*)
  fix *T*
  assume *propagate* (*toS S*) *T*
  then obtain *M N U k C L E* **where**
    toSS: *toS S* = (*M*, *N*, *U*, *k*, *None*) **and**
    LE: *L* ∈# *E* **and**
    T: *T* = (*Propagated L E* # *M*, *N*, *U*, *k*, *None*) **and**
    MC: *M* ⊨*as CNot C* **and**
    undef: *undefined-lit M L* **and**
    CL: *C* + {#*L*#} ∈# *N* + *U*
    apply − **by** (*cases toS S*) (*auto elim*!: *propagateE*)
  let *?M* = *raw-trail S*
  let *?N* = *raw-init-clss S*
  let *?U* = *raw-learned-clss S*
  let *?k* = *raw-backtrack-lvl S*
  let *?D* = *None*
  have *S*: *S* = (*?M*, *?N*, *?U*, *?k*, *?D*)
    using *toSS* **by** (*cases S*, *cases raw-conflicting S*) *simp-all*
  have *S*: *toS S* = *toS* (*?M*, *?N*, *?U*, *?k*, *?D*)
    unfolding *S*[*symmetric*] **by** *simp*

  have
    M: *M* = *map convert ?M* **and**
    N: *N* = *mset* (*map mset ?N*) **and**
    U: *U* = *mset* (*map mset ?U*)
    using *toSS*[*unfolded S*] **by** *auto*

  obtain *D* **where**
    DCL: *mset D* = *C* + {#*L*#} **and**
    D: *D* ∈ *set* (*?N* @ *?U*)
    using *CL* **unfolding** *N U* **by** *auto*
  obtain *C′ L′* **where**
    setD: *set D* = *set* (*L′* # *C′*) **and**
    C′: *mset C′* = *C* **and**
    L: *L* = *L′*
    using *DCL* **by** (*metis ex-mset mset.simps*(*2*) *mset-eq-setD*)
  have *find-first-unit-clause* (*?N* @ *?U*) *?M* ≠ *None*
    apply (*rule dist find-first-unit-clause-none*[*of D ?N* @ *?U ?M L*, *OF - D*])
       using *D assms*(*1*) **apply** *auto*[*1*]
      using *MC setD DCL M MC* **unfolding** *C′*[*symmetric*] **apply** *auto*[*1*]
     using *M undef* **apply** *auto*[*1*]
    unfolding *setD L* **by** *auto*
  then show *False* **using** *prop-step S* **unfolding** *do-propagate-step-def* **by** (*cases S*) *auto*
qed

Conflict **fun** *find-conflict* **where**
find-conflict M [] = *None* |
find-conflict M (*N* # *Ns*) = (*if* (∀ *c* ∈ *set N*. −*c* ∈ *lits-of-l M*) *then Some N else find-conflict M Ns*)

lemma *find-conflict-Some*:
  find-conflict M Ns = *Some N* ⟹ *N* ∈ *set Ns* ∧ *M* ⊨*as CNot* (*mset N*)
  by (*induction Ns rule*: *find-conflict.induct*)
    (*auto split*: *if-split-asm*)

lemma *find-conflict-None*:
  find-conflict M Ns = *None* ⟷ (∀ *N* ∈ *set Ns*. ¬*M* ⊨*as CNot* (*mset N*))

by (*induction Ns*) *auto*

lemma *find-conflict-None-no-confl*:
find-conflict M (*N*@*U*) = *None* ⟷ *no-step conflict* (*toS* (*M*, *N*, *U*, *k*, *None*))
by (*auto simp add*: *find-conflict-None conflict.simps*)

definition *do-conflict-step* **where**
do-conflict-step S =
(*case S of*
(*M*, *N*, *U*, *k*, *None*) ⇒
(*case find-conflict M* (*N* @ *U*) *of*
Some a ⇒ (*M*, *N*, *U*, *k*, *Some a*)
| *None* ⇒ (*M*, *N*, *U*, *k*, *None*))
| *S* ⇒ *S*)

lemma *do-conflict-step*:
do-conflict-step S ≠ *S* ⟹ *conflict* (*toS S*) (*toS* (*do-conflict-step S*))
apply (*cases S*, *cases raw-conflicting S*)
unfolding *conflict.simps do-conflict-step-def*
by (*auto dest*!:*find-conflict-Some split*: *option.splits*)

lemma *do-conflict-step-no-step*:
do-conflict-step S = *S* ⟹ *no-step conflict* (*toS S*)
apply (*cases S*, *cases raw-conflicting S*)
unfolding *do-conflict-step-def*
using *find-conflict-None-no-confl*[*of raw-trail S raw-init-clss S raw-learned-clss S raw-backtrack-lvl S*]
by (*auto split*: *option.splits elim*!: *conflictE*)

lemma *do-conflict-step-option*[*simp*]:
raw-conflicting S ≠ *None* ⟹ *do-conflict-step S* = *S*
unfolding *do-conflict-step-def* **by** (*cases S*, *cases raw-conflicting S*) *auto*

lemma *do-conflict-step-raw-conflicting*[*dest*]:
do-conflict-step S ≠ *S* ⟹ *raw-conflicting* (*do-conflict-step S*) ≠ *None*
unfolding *do-conflict-step-def* **by** (*cases S*, *cases raw-conflicting S*) (*auto split*: *option.splits*)

definition *do-cp-step* **where**
do-cp-step S =
(*do-propagate-step o do-conflict-step*) *S*

lemma *cp-step-is-*$cdcl_W$*-cp*:
assumes *H*: *do-cp-step S* ≠ *S*
shows $cdcl_W$*-cp* (*toS S*) (*toS* (*do-cp-step S*))
proof −
show *?thesis*
proof (*cases do-conflict-step S* ≠ *S*)
case *True*
then show *?thesis*
by (*auto simp add*: *do-conflict-step do-conflict-step-raw-conflicting do-cp-step-def*)
next
case *False*
then have *confl*[*simp*]: *do-conflict-step S* = *S* **by** *simp*
show *?thesis*
proof (*cases do-propagate-step S* = *S*)
case *True*

then show *?thesis*
using *H* **by** (*simp add*: *do-cp-step-def*)
next
case *False*
let *?S* = *toS S*
let *?T* = *toS* (*do-propagate-step S*)
let *?U* = *toS* (*do-conflict-step* (*do-propagate-step S*))
have *propa*: *propagate* (*toS S*) *?T* **using** *False do-propgate-step* **by** *blast*
moreover have *ns*: *no-step conflict* (*toS S*) **using** *confl do-conflict-step-no-step* **by** *blast*
ultimately show *?thesis*
using $cdcl_W$*-cp.intros*(*2*)[*of ?S ?T*] *confl* **unfolding** *do-cp-step-def* **by** *auto*
qed
qed
qed

lemma *do-cp-step-eq-no-prop-no-confl*:
do-cp-step S = *S* ⟹ *do-conflict-step S* = *S* ∧ *do-propagate-step S* = *S*
by (*cases S*, *cases raw-conflicting S*)
(*auto simp add*: *do-conflict-step-def do-propagate-step-def do-cp-step-def split*: *option.splits*)

lemma *no-*$cdcl_W$*-cp-iff-no-propagate-no-conflict*:
no-step $cdcl_W$*-cp S* ⟷ *no-step propagate S* ∧ *no-step conflict S*
by (*auto simp*: $cdcl_W$*-cp.simps*)

lemma *do-cp-step-eq-no-step*:
assumes *H*: *do-cp-step S* = *S* **and** ∀ *c* ∈ *set* (*raw-init-clss S* @ *raw-learned-clss S*). *distinct c*
shows *no-step* $cdcl_W$*-cp* (*toS S*)
unfolding *no-*$cdcl_W$*-cp-iff-no-propagate-no-conflict*
using *assms* **apply** (*cases S*, *cases raw-conflicting S*)
using *do-propagate-step-no-step*[*of S*]
by (*auto dest*!: *do-cp-step-eq-no-prop-no-confl*[*simplified*] *do-conflict-step-no-step*
split: *option.splits*)

lemma $cdcl_W$*-cp-*$cdcl_W$*-st*: $cdcl_W$*-cp S S'* ⟹ $cdcl_W^{**}$ *S S'*
by (*simp add*: $cdcl_W$*-cp-tranclp-*$cdcl_W$ *tranclp-into-rtranclp*)

lemma $cdcl_W$*-all-struct-inv-rough-state*[*simp*]: $cdcl_W$*-all-struct-inv* (*toS* (*rough-state-of S*))
using *rough-state-of* **by** *auto*

lemma [*simp*]: $cdcl_W$*-all-struct-inv* (*toS S*) ⟹ *rough-state-of* (*state-of S*) = *S*
by (*simp add*: *state-of-inverse*)

lemma *rough-state-of-state-of-do-cp-step*[*simp*]:
rough-state-of (*state-of* (*do-cp-step* (*rough-state-of S*))) = *do-cp-step* (*rough-state-of S*)
proof −
have $cdcl_W$*-all-struct-inv* (*toS* (*do-cp-step* (*rough-state-of S*)))
apply (*cases do-cp-step* (*rough-state-of S*) = (*rough-state-of S*))
apply *simp*
using *cp-step-is-*$cdcl_W$*-cp*[*of rough-state-of S*] $cdcl_W$*-all-struct-inv-rough-state*[*of S*]
$cdcl_W$*-cp-*$cdcl_W$*-st rtranclp-*$cdcl_W$*-all-struct-inv-inv* **by** *blast*
then show *?thesis* **by** *auto*
qed

Skip **fun** *do-skip-step* :: *'v* $cdcl_W$*-state-inv-st* ⇒ *'v* $cdcl_W$*-state-inv-st* **where**
do-skip-step (*Propagated L C* # *Ls*,*N*,*U*,*k*, *Some D*) =
(*if* −*L* ∉ *set D* ∧ *D* ≠ []

then (*Ls*, *N*, *U*, *k*, *Some D*)
else (*Propagated L C* #*Ls*, *N*, *U*, *k*, *Some D*)) |
do-skip-step S = *S*

lemma *do-skip-step*:
do-skip-step S $\neq$ *S* $\Longrightarrow$ *skip* (*toS S*) (*toS* (*do-skip-step S*))
apply (*induction S rule*: *do-skip-step.induct*)
by (*auto simp add*: *skip.simps*)

lemma *do-skip-step-no*:
do-skip-step S = *S* $\Longrightarrow$ *no-step skip* (*toS S*)
by (*induction S rule*: *do-skip-step.induct*)
(*auto simp add*: *other split*: *if-split-asm elim*: *skipE*)

lemma *do-skip-step-raw-trail-is-None*[*iff*]:
do-skip-step S = (*a*, *b*, *c*, *d*, *None*) $\longleftrightarrow$ *S* = (*a*, *b*, *c*, *d*, *None*)
by (*cases S rule*: *do-skip-step.cases*) *auto*

Resolve **fun** *maximum-level-code*:: *'a literal list* $\Rightarrow$ (*'a*, *'a literal list*) *ann-lit list* $\Rightarrow$ *nat*
where
maximum-level-code [] - = *0* |
maximum-level-code (*L* # *Ls*) *M* = *max* (*get-level M L*) (*maximum-level-code Ls M*)

lemma *maximum-level-code-eq-get-maximum-level*[*code*, *simp*]:
maximum-level-code D M = *get-maximum-level M* (*mset D*)
by (*induction D*) (*auto simp add*: *get-maximum-level-plus*)

fun *do-resolve-step* :: *'v* $cdcl_W$*-state-inv-st* $\Rightarrow$ *'v* $cdcl_W$*-state-inv-st* **where**
do-resolve-step (*Propagated L C* # *Ls*, *N*, *U*, *k*, *Some D*) =
(*if* −*L* $\in$ *set D* $\wedge$ *maximum-level-code* (*remove1* (−*L*) *D*) (*Propagated L C* # *Ls*) = *k*
then (*Ls*, *N*, *U*, *k*, *Some* (*remdups* (*remove1 L C* @ *remove1* (−*L*) *D*)))
else (*Propagated L C* # *Ls*, *N*, *U*, *k*, *Some D*)) |
do-resolve-step S = *S*

lemma *do-resolve-step*:
$cdcl_W$*-all-struct-inv* (*toS S*) $\Longrightarrow$ *do-resolve-step S* $\neq$ *S*
$\Longrightarrow$ *resolve* (*toS S*) (*toS* (*do-resolve-step S*))
proof (*induction S rule*: *do-resolve-step.induct*)
case (*1 L C M N U k D*)
then have
− *L* $\in$ *set D* **and**
M: *maximum-level-code* (*remove1* (−*L*) *D*) (*Propagated L C* # *M*) = *k*
by (*cases mset D* − {#− *L*#} = {#},
auto dest!: *get-maximum-level-exists-lit-of-max-level*[*of* - *Propagated L C* # *M*]
split: *if-split-asm*)+
have *every-mark-is-a-conflict* (*toS* (*Propagated L C* # *M*, *N*, *U*, *k*, *Some D*))
using *1*(*1*) **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def* **by** *fast*
then have *L* $\in$ *set C* **by** *fastforce*
then obtain *C'* **where** *C*: *mset C* = *C'* + {#*L*#}
by (*metis add.commute in-multiset-in-set insert-DiffM*)
obtain *D'* **where** *D*: *mset D* = *D'* + {#−*L*#}
using ‹− *L* $\in$ *set D*› **by** (*metis add.commute in-multiset-in-set insert-DiffM*)
have *D'L*: *D'* + {#− *L*#} − {#−*L*#} = *D'* **by** (*auto simp add*: *multiset-eq-iff*)

have *CL*: *mset C* − {#*L*#} + {#*L*#} = *mset C* **using** ‹*L* $\in$ *set C*› **by** (*auto simp add*: *multiset-eq-iff*)
have *get-maximum-level* (*Propagated L* (*C'* + {#*L*#}) # *map convert M*) *D'* = *k*

using *M*[*simplified*] **unfolding** *maximum-level-code-eq-get-maximum-level C*[*symmetric*] *CL*
by (*metis D D′L convert.simps*(*1*) *get-maximum-level-map-convert list.simps*(*9*))
then have
resolve
(*map convert* (*Propagated L C # M*), *mset ‘# mset N*, *mset ‘# mset U*, *k*, *Some* (*mset D*))
(*map convert M*, *mset ‘# mset N*, *mset ‘# mset U*, *k*,
Some (((*mset D* − {#−*L*#}) #∪ (*mset C* − {#*L*#}))))
unfolding *resolve.simps*
by (*simp add*: *C D*)
moreover have
(*map convert* (*Propagated L C # M*), *mset ‘# mset N*, *mset ‘# mset U*, *k*, *Some* (*mset D*))
= *toS* (*Propagated L C # M*, *N*, *U*, *k*, *Some D*)
by (*auto simp*: *mset-map*)
moreover
have *distinct-mset* (*mset C*) **and** *distinct-mset* (*mset D*)
using ‹$cdcl_W$*-all-struct-inv* (*toS* (*Propagated L C # M*, *N*, *U*, *k*, *Some D*))›
unfolding $cdcl_W$*-all-struct-inv-def distinct-*$cdcl_W$*-state-def*
by *auto*
then have (*mset C* − {#*L*#}) #∪ (*mset D* − {#− *L*#}) =
remdups-mset (*mset C* − {#*L*#} + (*mset D* − {#− *L*#}))
by (*auto simp*: *distinct-mset-rempdups-union-mset*)
then have (*map convert M*, *mset ‘# mset N*, *mset ‘# mset U*, *k*,
Some ((*mset D* − {#− *L*#}) #∪ (*mset C* − {#*L*#})))
= *toS* (*do-resolve-step* (*Propagated L C # M*, *N*, *U*, *k*, *Some D*))
using ‹− *L* ∈ *set D*› *M* **by** (*auto simp*:*ac-simps mset-map*)
ultimately show *?case*
by *simp*
qed *auto*

lemma *do-resolve-step-no*:
do-resolve-step S = *S* ⟹ *no-step resolve* (*toS S*)
apply (*cases S*; *cases hd* (*raw-trail S*);*cases raw-trail S*; *cases raw-conflicting S*)
by (*auto*
elim!: *resolveE split*: *if-split-asm*
dest!: *union-single-eq-member*
simp del: *in-multiset-in-set get-maximum-level-map-convert*
simp: *get-maximum-level-map-convert*[*symmetric*])

lemma *rough-state-of-state-of-resolve*[*simp*]:
$cdcl_W$*-all-struct-inv* (*toS S*) ⟹ *rough-state-of* (*state-of* (*do-resolve-step S*)) = *do-resolve-step S*
apply (*rule state-of-inverse*)
apply (*cases do-resolve-step S* = *S*)
apply *simp*
by (*blast dest*: *other resolve bj do-resolve-step* $cdcl_W$*-all-struct-inv-inv*)

lemma *do-resolve-step-raw-trail-is-None*[*iff*]:
do-resolve-step S = (*a*, *b*, *c*, *d*, *None*) ⟷ *S* = (*a*, *b*, *c*, *d*, *None*)
by (*cases S rule*: *do-resolve-step.cases*) *auto*

Backjumping **lemma** *get-all-ann-decomposition-map-convert*:
(*get-all-ann-decomposition* (*map convert M*)) =
map (λ(*a*, *b*). (*map convert a*, *map convert b*)) (*get-all-ann-decomposition M*)
apply (*induction M rule*: *ann-lit-list-induct*)
apply *simp*
by (*rename-tac L xs*, *case-tac get-all-ann-decomposition xs*; *auto*)+

```
lemma do-backtrack-step:
  assumes
    db: do-backtrack-step S ≠ S and
    inv: cdcl_W-all-struct-inv (toS S)
  shows backtrack (toS S) (toS (do-backtrack-step S))
  proof (cases S, cases raw-conflicting S, goal-cases)
    case (1 M N U k E)
    then show ?case using db by auto
  next
    case (2 M N U k E C) note S = this(1) and confl = this(2)
    have E: E = Some C using S confl by auto

    obtain L j where fd: find-level-decomp M C [] k = Some (L, j)
      using db unfolding S E  by (cases C) (auto split: if-split-asm option.splits list.splits
        ann-lit.splits)
    have
      L ∈ set C and
      j: get-maximum-level M (mset (remove1 L C)) = j and
      levL: get-level M L = k
      using find-level-decomp-some[OF fd] by auto
    obtain C' where C: mset C = mset C' + {#L#}
      using ‹L ∈ set C› by (metis add.commute ex-mset in-multiset-in-set insert-DiffM)
    obtain M2 where M2: bt-cut j M = Some M2
      using db fd unfolding S E by (auto split: option.splits)
    have no-dup M and k: k = count-decided (filter is-decided M)
      using inv unfolding cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def S by (auto simp: comp-def)
    then obtain M1 K c where
      M1: M2 = Decided K # M1 and lev-K: get-level M K = j + 1 and
      c: M = c @ M2
      using bt-cut-some-decomp[OF - M2] by (cases M2) auto
     have j ≤ k unfolding c j[symmetric] k
      by (metis (mono-tags, lifting) count-decided-ge-get-maximum-level filter-cong filter-filter)
    have max-l-j: maximum-level-code C' M = j
      using db fd M2 C unfolding S E by (auto
          split: option.splits list.splits ann-lit.splits
          dest!: find-level-decomp-some)[1]
    have get-maximum-level M (mset C) ≥ k
      using ‹L ∈ set C› levL get-maximum-level-ge-get-level by (metis set-mset-mset)
    moreover have get-maximum-level M (mset C) ≤ k
      using get-maximum-level-exists-lit-of-max-level[of mset C M] inv
        cdcl_W-M-level-inv-get-level-le-backtrack-lvl[of toS S]
      unfolding C cdcl_W-all-struct-inv-def S by (auto dest: sym[of get-level - -])
    ultimately have get-maximum-level M (mset C) = k by auto

    obtain M2' where M2': (M2, M2') ∈ set (get-all-ann-decomposition M)
      using bt-cut-in-get-all-ann-decomposition[OF ‹no-dup M› M2] by metis
    have decomp:
      (Decided K # (map convert M1),
      (map convert M2')) ∈
      set (get-all-ann-decomposition (map convert M))
      using imageI[of - - λ(a, b). (map convert a, map convert b), OF M2'] j
      unfolding S E M1 by (simp add: get-all-ann-decomposition-map-convert)
    show ?case
      apply (rule backtrack-rule)
        using M2 fd confl ‹L ∈ set C› j decomp levL ‹get-maximum-level M (mset C) = k›
        unfolding S E M1 apply (auto simp: mset-map)[6]
```

using *M2′ M2 fd j lev-K* **unfolding** *S E M1 CDCL-W-Implementation.state-eq-def*
by (*auto simp*: *comp-def ac-simps*)[*2*]
qed

lemma *map-eq-list-length*:
map f L = *L′* $\Longrightarrow$ *length L* = *length L′*
by *auto*

lemma *map-mmset-of-mlit-eq-cons*:
assumes *map convert M* = *a* @ *c*
obtains *a′ c′* **where**
M = *a′* @ *c′* **and**
a = *map convert a′* **and**
c = *map convert c′*
using *that*[*of take* (*length a*) *M drop* (*length a*) *M*]
assms **by** (*metis append-eq-conv-conj append-take-drop-id drop-map take-map*)

lemma *Decided-convert-iff*:
Decided K = *convert za* $\longleftrightarrow$ *za* = *Decided K*
by (*cases za*) *auto*

lemma *do-backtrack-step-no*:
assumes
db: *do-backtrack-step S* = *S* **and**
inv: $cdcl_W$*-all-struct-inv* (*toS S*)
shows *no-step backtrack* (*toS S*)
proof (*rule ccontr*, *cases S*, *cases raw-conflicting S*, *goal-cases*)
case *1*
then show *?case* **using** *db* **by** (*auto split*: *option.splits elim*: *backtrackE*)
next
case (*2 M N U k E C*) **note** *bt* = *this*(*1*) **and** *S* = *this*(*2*) **and** *confl* = *this*(*3*)
obtain *K j M1 M2 L D* **where**
CE: *raw-conflicting S* = *Some D* **and**
LD: *L* ∈# *mset D* **and**
decomp: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*raw-trail S*)) **and**
levL: *get-level* (*raw-trail S*) *L* = *raw-backtrack-lvl S* **and**
k: *get-level* (*raw-trail S*) *L* = *get-maximum-level* (*raw-trail S*) (*mset D*) **and**
j: *get-maximum-level* (*raw-trail S*) (*remove1-mset L* (*mset D*)) ≡ *j* **and**
lev-K: *get-level* (*raw-trail S*) *K* = *Suc j*
using *bt* **apply** *clarsimp*
apply (*elim backtrackE*)
apply (*cases S*)
by (*auto simp add*: *get-all-ann-decomposition-map-convert reduce-trail-to*
Decided-convert-iff)
obtain *c* **where** *c*: *raw-trail S* = *c* @ *M2* @ *Decided K* # *M1*
using *decomp* **by** *blast*
have *k* = *count-decided* (*raw-trail S*) **and** *n-d*: *no-dup M*
using *inv S* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def*
by (*auto simp*: *comp-def*)
then have *k* > *j*
using *j count-decided-ge-get-maximum-level*[*of raw-trail S remove1-mset L* (*mset D*)]
count-decided-ge-get-level[*of K raw-trail S*]
unfolding *k lev-K*
unfolding *c* **by** (*auto simp*: *get-all-ann-decomposition-map-convert simp del*: *count-decided-ge-get-level*)
have [*simp*]: *L* ∈ *set D*
using *LD* **by** *auto*

have *CD*: *C* = *D*
using *CE confl* **by** *auto*
obtain *D′* **where**
E: *E* = *Some D* **and**
DD′: *mset D* = {#*L*#} + *mset D′*
using *that*[*of remove1 L D*]
using *S CE confl LD* **by** (*auto simp add*: *insert-DiffM*)
have *find-level-decomp M D* [] *k* ≠ *None*
apply *rule*
apply (*drule find-level-decomp-none*[*of - - - - L D′*])
using *DD′* ‹*k* > *j*› *mset-eq-setD S levL* **unfolding** *k*[*symmetric*] *j*[*symmetric*]
by (*auto simp*: *ac-simps*)
then obtain *L′ j′* **where** *fd-some*: *find-level-decomp M D* [] *k* = *Some* (*L′*, *j′*)
by (*cases find-level-decomp M D* [] *k*) *auto*
have *L′*: *L′* = *L*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *L′* ∈# *mset* (*remove1 L D*)
by (*metis fd-some find-level-decomp-some in-set-remove1 set-mset-mset*)
then have *get-level M L′* ≤ *get-maximum-level M* (*mset* (*remove1 L D*))
using *get-maximum-level-ge-get-level* **by** *blast*
then show *False* **using** ‹*k* > *j*› *j find-level-decomp-some*[*OF fd-some*] *S DD′* **by** *auto*
qed
then have *j′*: *j′* = *j* **using** *find-level-decomp-some*[*OF fd-some*] *j S DD′* **by** *auto*

obtain *c′ M1′* **where** *cM*: *M* = *c′* @ *Decided K* # *M1′*
apply (*rule map-mmset-of-mlit-eq-cons*[*of M map convert* (*c* @ *M2*)
map convert (*Decided K* # *M1*)])
using *c S* **apply** *simp*
apply (*rule map-mmset-of-mlit-eq-cons*[*of - map convert* [*Decided K*] *map convert M1*])
apply *auto*[]
apply (*rename-tac a b′ aa b*, *case-tac aa*)
apply *auto*[]
apply (*rename-tac a b′ aa b*, *case-tac aa*)
by *auto*
have *btc-none*: *bt-cut j M* ≠ *None*
apply (*rule bt-cut-not-none*[*of M*])
using *n-d cM S lev-K S* **apply** *blast+*
using *lev-K S* **by** *auto*
show *?case* **using** *db n-d* **unfolding** *S E*
by (*auto split*: *option.splits list.splits ann-lit.splits*
simp add: *fd-some L′ j′ btc-none*
dest: *bt-cut-some-decomp*)
qed

lemma *rough-state-of-state-of-backtrack*[*simp*]:
assumes *inv*: $cdcl_W$*-all-struct-inv* (*toS S*)
shows *rough-state-of* (*state-of* (*do-backtrack-step S*))= *do-backtrack-step S*
proof (*rule state-of-inverse*)
consider
(*step*) *backtrack* (*toS S*) (*toS* (*do-backtrack-step S*)) |
(*0*) *do-backtrack-step S* = *S*
using *do-backtrack-step inv* **by** *blast*
then show *do-backtrack-step S* ∈ {*S*. $cdcl_W$*-all-struct-inv* (*toS S*)}
proof *cases*
case *0*

thus *?thesis* **using** *inv* **by** *simp*
next
case *step*
then show *?thesis*
using *inv*
by (*auto dest!*: $cdcl_W$*.other* $cdcl_W$*-o.bj* $cdcl_W$*-bj.backtrack intro*: $cdcl_W$*-all-struct-inv-inv*)
qed
qed

Decide

fun *do-decide-step* **where**
do-decide-step (*M*, *N*, *U*, *k*, *None*) =
(*case find-first-unused-var N* (*lits-of-l M*) *of*
None ⇒ (*M*, *N*, *U*, *k*, *None*)
| *Some L* ⇒ (*Decided L* # *M*, *N*, *U*, *k+1*, *None*)) |
do-decide-step S = *S*

lemma *do-decide-step*:
do-decide-step S ≠ *S* ⟹ *decide* (*toS S*) (*toS* (*do-decide-step S*))
apply (*cases S*, *cases raw-conflicting S*)
defer
apply (*auto split*: *option.splits simp add*: *decide.simps*
dest: *find-first-unused-var-undefined find-first-unused-var-Some*
intro: *atms-of-atms-of-ms-mono*)[*1*]
proof −
fix *a* :: (′*a*, ′*a literal list*) *ann-lit list* **and**
b :: ′*a literal list list* **and** *c* :: ′*a literal list list* **and**
d :: *nat* **and** *e* :: ′*a literal list option*
{
fix *a* :: (′*a*, ′*a literal list*) *ann-lit list* **and**
b :: ′*a literal list list* **and** *c* :: ′*a literal list list* **and**
d :: *nat* **and** *x2* :: ′*a literal* **and** *m* :: ′*a literal list*
assume *a1*: *m* ∈ *set b*
assume *x2* ∈ *set m*
then have *f2*: *atm-of x2* ∈ *atms-of* (*mset m*)
by *simp*
have ⋀*f*. (*f m*::′*a literal multiset*) ∈ *f* ‘ *set b*
using *a1* **by** *blast*
then have ⋀*f*. (*atms-of* (*f m*)::′*a set*) ⊆ *atms-of-ms* (*f* ‘ *set b*)
using *atms-of-atms-of-ms-mono* **by** *blast*
then have ⋀*n f*. (*n*::′*a*) ∈ *atms-of-ms* (*f* ‘ *set b*) ∨ *n* ∉ *atms-of* (*f m*)
by (*meson contra-subsetD*)
then have *atm-of x2* ∈ *atms-of-ms* (*mset* ‘ *set b*)
using *f2* **by** *blast*
} **note** *H* = *this*
{
fix *m* :: ′*a literal list* **and** *x2*
have *m* ∈ *set b* ⟹ *x2* ∈ *set m* ⟹ *x2* ∉ *lits-of-l a* ⟹ − *x2* ∉ *lits-of-l a* ⟹
∃ *aa*∈*set b*. ¬ *atm-of* ‘ *set aa* ⊆ *atm-of* ‘ *lits-of-l a*
by (*meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI*)
} **note** *H*′ = *this*

assume *do-decide-step S* ≠ *S* **and**
S = (*a*, *b*, *c*, *d*, *e*) **and**
raw-conflicting S = *None*
then show *decide* (*toS S*) (*toS* (*do-decide-step S*))
using *H H*′ **by** (*auto split*: *option.splits simp*: *decide.simps defined-lit-map lits-of-def*

image-image atm-of-eq-atm-of dest!: *find-first-unused-var-Some*)
qed

lemma *do-decide-step-no*:
do-decide-step S = *S* $\Longrightarrow$ *no-step decide* (*toS S*)
apply (*cases S*, *cases raw-conflicting S*)
apply (*auto simp*: *atms-of-ms-mset-unfold Decided-Propagated-in-iff-in-lits-of-l lits-of-def*
dest!: *atm-of-in-atm-of-set-in-uminus*
elim!: *decideE*
split: *option.splits*)+
using *atm-of-eq-atm-of* **by** *blast*+

lemma *rough-state-of-state-of-do-decide-step*[*simp*]:
$cdcl_W$*-all-struct-inv* (*toS S*) $\Longrightarrow$ *rough-state-of* (*state-of* (*do-decide-step S*)) = *do-decide-step S*
proof (*subst state-of-inverse*, *goal-cases*)
case *1*
then show *?case*
by (*cases do-decide-step S* = *S*)
(*auto dest*: *do-decide-step decide other intro*: $cdcl_W$*-all-struct-inv-inv*)
qed *simp*

lemma *rough-state-of-state-of-do-skip-step*[*simp*]:
$cdcl_W$*-all-struct-inv* (*toS S*) $\Longrightarrow$ *rough-state-of* (*state-of* (*do-skip-step S*)) = *do-skip-step S*
apply (*subst state-of-inverse*, *cases do-skip-step S* = *S*)
apply *simp*
by (*blast dest*: *other skip bj do-skip-step* $cdcl_W$*-all-struct-inv-inv*)+

Code generation

Type definition There are two invariants: one while applying conflict and propagate and one for the other rules

declare *rough-state-of-inverse*[*simp add*]
definition *Con* **where**
Con xs = *state-of* (*if* $cdcl_W$*-all-struct-inv* (*toS* (*fst xs*, *snd xs*)) *then xs*
else ([], [], [], *0*, *None*))

lemma [*code abstype*]:
Con (*rough-state-of S*) = *S*
using *rough-state-of*[*of S*] **unfolding** *Con-def* **by** *simp*

definition *do-cp-step'* **where**
do-cp-step' S = *state-of* (*do-cp-step* (*rough-state-of S*))

typedef *'v* $cdcl_W$*-state-inv-from-init-state* =
{*S*:: *'v* $cdcl_W$*-state-inv-st*. $cdcl_W$*-all-struct-inv* (*toS S*)
$\wedge$ $cdcl_W$*-stgy*** (*S0-*$cdcl_W$ (*raw-init-clss* (*toS S*))) (*toS S*)}
morphisms *rough-state-from-init-state-of state-from-init-state-of*
proof
show ([],[], [], *0*, *None*) $\in$ {*S*. $cdcl_W$*-all-struct-inv* (*toS S*)
$\wedge$ $cdcl_W$*-stgy*** (*S0-*$cdcl_W$ (*raw-init-clss* (*toS S*))) (*toS S*)}
by (*auto simp add*: $cdcl_W$*-all-struct-inv-def*)
qed

instantiation $cdcl_W$*-state-inv-from-init-state* :: (*type*) *equal*

begin
definition *equal-cdcl$_W$-state-inv-from-init-state* :: *'v cdcl$_W$-state-inv-from-init-state* ⇒ *'v cdcl$_W$-state-inv-from-init-state* ⇒ *bool* **where**
equal-cdcl$_W$-state-inv-from-init-state S S' ⟷
(*rough-state-from-init-state-of S* = *rough-state-from-init-state-of S'*)
instance
by *standard* (*simp add*: *rough-state-from-init-state-of-inject equal-cdcl$_W$-state-inv-from-init-state-def*)
end

definition *ConI* **where**
ConI S = *state-from-init-state-of* (*if cdcl$_W$-all-struct-inv* (*toS* (*fst S*, *snd S*))
∧ *cdcl$_W$-stgy*** (*S0-cdcl$_W$* (*raw-init-clss* (*toS S*))) (*toS S*) *then S else* ([], [], [], *0*, *None*))

lemma [*code abstype*]:
ConI (*rough-state-from-init-state-of S*) = *S*
using *rough-state-from-init-state-of*[*of S*] **unfolding** *ConI-def*
by (*simp add*: *rough-state-from-init-state-of-inverse*)

definition *id-of-I-to*:: *'v cdcl$_W$-state-inv-from-init-state* ⇒ *'v cdcl$_W$-state-inv* **where**
id-of-I-to S = *state-of* (*rough-state-from-init-state-of S*)

lemma [*code abstract*]:
rough-state-of (*id-of-I-to S*) = *rough-state-from-init-state-of S*
unfolding *id-of-I-to-def* **using** *rough-state-from-init-state-of*[*of S*] **by** *auto*

Conflict and Propagate **function** *do-full1-cp-step* :: *'v cdcl$_W$-state-inv* ⇒ *'v cdcl$_W$-state-inv* **where**
do-full1-cp-step S =
(*let S'* = *do-cp-step' S in*
if S = *S' then S else do-full1-cp-step S'*)
by *auto*
termination
proof (*relation* {(*T'*, *T*). (*rough-state-of T'*, *rough-state-of T*) ∈ {(*S'*, *S*).
(*toS S'*, *toS S*) ∈ {(*S'*, *S*). *cdcl$_W$-all-struct-inv S* ∧ *cdcl$_W$-cp S S'*}}}, *goal-cases*)
case *1*
show *?case*
using *wf-if-measure-f*[*OF wf-if-measure-f*[*OF cdcl$_W$-cp-wf-all-inv*, *of toS*], *of rough-state-of*] **.**
next
case (*2 S' S*)
then show *?case*
unfolding *do-cp-step'-def*
apply *simp*
by (*metis cp-step-is-cdcl$_W$-cp rough-state-of-inverse*)
qed

lemma *do-full1-cp-step-fix-point-of-do-full1-cp-step*:
do-cp-step(*rough-state-of* (*do-full1-cp-step S*)) = (*rough-state-of* (*do-full1-cp-step S*))
by (*rule do-full1-cp-step.induct*[*of λS. do-cp-step*(*rough-state-of* (*do-full1-cp-step S*))
= (*rough-state-of* (*do-full1-cp-step S*))]])
(*metis* (*full-types*) *do-full1-cp-step.elims rough-state-of-state-of-do-cp-step do-cp-step'-def*)

lemma *in-clauses-rough-state-of-is-distinct*:
c∈*set* (*raw-init-clss* (*rough-state-of S*) @ *raw-learned-clss* (*rough-state-of S*)) ⟹ *distinct c*
apply (*cases rough-state-of S*)
using *rough-state-of*[*of S*] **by** (*auto simp add*: *distinct-mset-set-distinct cdcl$_W$-all-struct-inv-def*

distinct-cdcl$_W$-state-def)

lemma *do-full1-cp-step-full*:
full cdcl$_W$-cp (*toS* (*rough-state-of S*))
(*toS* (*rough-state-of* (*do-full1-cp-step S*)))
unfolding *full-def*
proof (*rule conjI*, *induction S rule*: *do-full1-cp-step.induct*)
case (*1 S*)
then have *f1*:
cdcl$_W$-cp** (*toS* (*do-cp-step* (*rough-state-of S*))) (
toS (*rough-state-of* (*do-full1-cp-step* (*state-of* (*do-cp-step* (*rough-state-of S*))))))
$\vee$ *state-of* (*do-cp-step* (*rough-state-of S*)) = *S*
using *rough-state-of-state-of-do-cp-step* **unfolding** *do-cp-step′-def* **by** *fastforce*
have *f2*: $\bigwedge$*c*. (*if c* = *state-of* (*do-cp-step* (*rough-state-of c*))
then c else do-full1-cp-step (*state-of* (*do-cp-step* (*rough-state-of c*))))
= *do-full1-cp-step c*
by (*metis* (*full-types*) *do-cp-step′-def do-full1-cp-step.simps*)
have *f3*: $\neg$ *cdcl$_W$-cp* (*toS* (*rough-state-of S*)) (*toS* (*do-cp-step* (*rough-state-of S*)))
$\vee$ *state-of* (*do-cp-step* (*rough-state-of S*)) = *S*
$\vee$ *cdcl$_W$-cp^{++}* (*toS* (*rough-state-of S*))
(*toS* (*rough-state-of* (*do-full1-cp-step* (*state-of* (*do-cp-step* (*rough-state-of S*))))))
using *f1* **by** (*meson rtranclp-into-tranclp2*)
{ assume *do-full1-cp-step S* $\neq$ *S*
then have *do-cp-step* (*rough-state-of S*) = *rough-state-of S*
$\longrightarrow$ *cdcl$_W$-cp*** (*toS* (*rough-state-of S*)) (*toS* (*rough-state-of* (*do-full1-cp-step S*)))
$\vee$ *do-cp-step* (*rough-state-of S*) $\neq$ *rough-state-of S*
$\wedge$ *state-of* (*do-cp-step* (*rough-state-of S*)) $\neq$ *S*
using *f2 f1* **by** (*metis* (*no-types*))
then have *do-cp-step* (*rough-state-of S*) $\neq$ *rough-state-of S*
$\wedge$ *state-of* (*do-cp-step* (*rough-state-of S*)) $\neq$ *S*
$\vee$ *cdcl$_W$-cp*** (*toS* (*rough-state-of S*)) (*toS* (*rough-state-of* (*do-full1-cp-step S*)))
by (*metis rough-state-of-state-of-do-cp-step*)
then have *cdcl$_W$-cp*** (*toS* (*rough-state-of S*)) (*toS* (*rough-state-of* (*do-full1-cp-step S*)))
using *f3 f2* **by** (*metis* (*no-types*) *cp-step-is-cdcl$_W$-cp tranclp-into-rtranclp*) **}**
then show *?case*
by *fastforce*
next
show *no-step cdcl$_W$-cp* (*toS* (*rough-state-of* (*do-full1-cp-step S*)))
apply (*rule do-cp-step-eq-no-step*[*OF do-full1-cp-step-fix-point-of-do-full1-cp-step*[*of S*]])
using *in-clauses-rough-state-of-is-distinct* **unfolding** *do-cp-step′-def* **by** *blast*
qed

lemma [*code abstract*]:
rough-state-of (*do-cp-step′ S*) = *do-cp-step* (*rough-state-of S*)
unfolding *do-cp-step′-def* **by** *auto*

The other rules **fun** *do-other-step* **where**

do-other-step S =
(*let T* = *do-skip-step S in*
if T $\neq$ *S*
then T
else
(*let U* = *do-resolve-step T in*
if U $\neq$ *T*
then U else
(*let V* = *do-backtrack-step U in*

if V ≠ U then V else do-decide-step V)))

lemma *do-other-step*:
assumes *inv*: $cdcl_W$*-all-struct-inv* (*toS S*) **and**
st: *do-other-step S ≠ S*
shows $cdcl_W$*-o* (*toS S*) (*toS* (*do-other-step S*))
using *st inv* **by** (*auto split*: *if-split-asm*
simp add: *Let-def*
dest!: *do-skip-step do-resolve-step do-backtrack-step do-decide-step*
dest!: $cdcl_W$*-o.intros* $cdcl_W$*-bj.intros*)

lemma *do-other-step-no*:
assumes *inv*: $cdcl_W$*-all-struct-inv* (*toS S*) **and**
st: *do-other-step S = S*
shows *no-step* $cdcl_W$*-o* (*toS S*)
using *st inv* **by** (*auto split*: *if-split-asm elim*: $cdcl_W$*-bjE*
simp add: *Let-def* $cdcl_W$*-bj.simps elim*!: $cdcl_W$*-o.cases*
dest!: *do-skip-step-no do-resolve-step-no do-backtrack-step-no do-decide-step-no*)

lemma *rough-state-of-state-of-do-other-step*[*simp*]:
rough-state-of (*state-of* (*do-other-step* (*rough-state-of S*))) = *do-other-step* (*rough-state-of S*)
proof (*cases do-other-step* (*rough-state-of S*) = *rough-state-of S*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have $cdcl_W$*-o* (*toS* (*rough-state-of S*)) (*toS* (*do-other-step* (*rough-state-of S*)))
by (*metis False* $cdcl_W$*-all-struct-inv-rough-state do-other-step*[*of rough-state-of S*])
then have $cdcl_W$*-all-struct-inv* (*toS* (*do-other-step* (*rough-state-of S*)))
using $cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-all-struct-inv-rough-state other* **by** *blast*
then show *?thesis*
by (*simp add*: *CollectI state-of-inverse*)
qed

definition *do-other-step′* **where**
do-other-step′ S =
state-of (*do-other-step* (*rough-state-of S*))

lemma *rough-state-of-do-other-step′*[*code abstract*]:
rough-state-of (*do-other-step′ S*) = *do-other-step* (*rough-state-of S*)
apply (*cases do-other-step* (*rough-state-of S*) = *rough-state-of S*)
unfolding *do-other-step′-def* **apply** *simp*
using *do-other-step*[*of rough-state-of S*] **by** (*auto intro*: $cdcl_W$*-all-struct-inv-inv*
$cdcl_W$*-all-struct-inv-rough-state other state-of-inverse*)

definition *do-*$cdcl_W$*-stgy-step* **where**
do-$cdcl_W$*-stgy-step S* =
(*let T = do-full1-cp-step S in*
if T ≠ S
then T
else
(*let U* = (*do-other-step′ T*) *in*
(*do-full1-cp-step U*)))

definition *do-*$cdcl_W$*-stgy-step′* **where**
do-$cdcl_W$*-stgy-step′ S* = *state-from-init-state-of* (*rough-state-of* (*do-*$cdcl_W$*-stgy-step* (*id-of-I-to S*)))

lemma *toS-do-full1-cp-step-not-eq*: *do-full1-cp-step S* $\neq$ *S* $\Longrightarrow$
toS (*rough-state-of S*) $\neq$ *toS* (*rough-state-of* (*do-full1-cp-step S*))
proof −
assume *a1*: *do-full1-cp-step S* $\neq$ *S*
then have *S* $\neq$ *do-cp-step′ S*
by *fastforce*
then show *?thesis*
by (*metis* (*no-types*) *cp-step-is-cdcl*$_W$*-cp do-cp-step′-def do-cp-step-eq-no-step do-full1-cp-step-fix-point-of-do-full1-cp-step in-clauses-rough-state-of-is-distinct rough-state-of-inverse*)
qed

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps*[*simp del*]

Correction of the transformation **lemma** *do-cdcl*$_W$*-stgy-step*:
assumes *do-cdcl*$_W$*-stgy-step S* $\neq$ *S*
shows *cdcl*$_W$*-stgy* (*toS* (*rough-state-of S*)) (*toS* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step S*)))
proof (*cases do-full1-cp-step S* = *S*)
case *False*
then show *?thesis*
using *assms do-full1-cp-step-full*[*of S*] **unfolding** *full-unfold do-cdcl*$_W$*-stgy-step-def*
by (*auto intro*!: *cdcl*$_W$*-stgy.intros dest*: *toS-do-full1-cp-step-not-eq*)
next
case *True*
have *cdcl*$_W$*-o* (*toS* (*rough-state-of S*)) (*toS* (*rough-state-of* (*do-other-step′ S*)))
by (*smt True assms cdcl*$_W$*-all-struct-inv-rough-state do-cdcl*$_W$*-stgy-step-def do-other-step rough-state-of-do-other-step′ rough-state-of-inverse*)
moreover
have
np: *no-step propagate* (*toS* (*rough-state-of S*)) **and**
nc: *no-step conflict* (*toS* (*rough-state-of S*))
apply (*metis True do-cp-step-eq-no-prop-no-confl do-full1-cp-step-fix-point-of-do-full1-cp-step do-propagate-step-no-step in-clauses-rough-state-of-is-distinct*)
by (*metis True do-conflict-step-no-step do-cp-step-eq-no-prop-no-confl do-full1-cp-step-fix-point-of-do-full1-cp-step*)
then have *no-step cdcl*$_W$*-cp* (*toS* (*rough-state-of S*))
by (*simp add*: *cdcl*$_W$*-cp.simps*)
moreover have *full cdcl*$_W$*-cp* (*toS* (*rough-state-of* (*do-other-step′ S*)))
(*toS* (*rough-state-of* (*do-full1-cp-step* (*do-other-step′ S*))))
using *do-full1-cp-step-full* **by** *auto*
ultimately show *?thesis*
using *assms True* **unfolding** *do-cdcl*$_W$*-stgy-step-def*
by (*auto intro*!: *cdcl*$_W$*-stgy.other′ dest*: *toS-do-full1-cp-step-not-eq*)
qed

lemma *length-raw-trail-toS*[*simp*]:
length (*raw-trail* (*toS S*)) = *length* (*raw-trail S*)
by (*cases S*) *auto*

lemma *raw-conflicting-noTrue-iff-toS*[*simp*]:
raw-conflicting (*toS S*) $\neq$ *None* $\longleftrightarrow$ *raw-conflicting S* $\neq$ *None*
by (*cases S*) *auto*

lemma *raw-trail-toS-neq-imp-raw-trail-neq*:
raw-trail (*toS S*) $\neq$ *raw-trail* (*toS S'*) $\Longrightarrow$ *raw-trail S* $\neq$ *raw-trail S'*
by (*cases S*, *cases S'*) *auto*

lemma *do-skip-step-raw-trail-changed-or-conflict*:
assumes *d*: *do-other-step S* $\neq$ *S*
and *inv*: *cdcl*$_W$*-all-struct-inv* (*toS S*)
shows *raw-trail S* $\neq$ *raw-trail* (*do-other-step S*)
proof –
have *M*: $\bigwedge$*M K M1 c*. *M* = *c* @ *K* # *M1* $\Longrightarrow$ *Suc* (*length M1*) $\leq$ *length M*
by *auto*
have *cdcl*$_W$*-M-level-inv* (*toS S*)
using *inv* **unfolding** *cdcl*$_W$*-all-struct-inv-def* **by** *auto*
have *cdcl*$_W$*-o* (*toS S*) (*toS* (*do-other-step S*)) **using** *do-other-step*[*OF inv d*] **.**
then show *?thesis*
using ‹*cdcl*$_W$*-M-level-inv* (*toS S*)›
proof (*induction toS* (*do-other-step S*) *rule*: *cdcl*$_W$*-o-induct*)
case *decide*
then show *?thesis*
by (*auto simp add*: *raw-trail-toS-neq-imp-raw-trail-neq*)[]
next
case (*skip*)
then show *?case*
by (*cases S*; *cases do-other-step S*) *force*
next
case (*resolve*)
then show *?case*
by (*cases S*, *cases do-other-step S*) *force*
next
case (*backtrack L D K i M1 M2*) **note** *LD* = *this*(*2*) **and** *decomp* = *this*(*3*) **and** *confl-S* = *this*(*1*)
and *i* = *this*(*6*) **and** *U* = *this*(*8*)

have
bt: *raw-backtrack-lvl* (*toS S*) = *count-decided* (*raw-trail* (*toS S*)) **and**
raw-trail (*toS S*) $\models$*as CNot D* **and**
cons: *consistent-interp* (*lits-of-l* (*raw-trail* (*toS S*)))
using *inv confl-S* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def*
cdcl$_W$*-conflicting-def* **by** *simp-all*
then have $-L \in$ *lits-of-l* (*raw-trail* (*toS S*))
using *LD true-annots-true-cls-def-iff-negation-in-model* **by** *blast*
then have $-L \in$ *lits-of-l* (*raw-trail S*)
by (*cases S*) (*auto simp*: *lits-of-def*)
moreover have *consistent-interp* (*lits-of-l* (*raw-trail S*))
using *cons* **by** (*cases S*) (*auto simp*: *lits-of-def image-image*)
ultimately have $L \notin$ *lits-of-l* (*raw-trail S*)
using *consistent-interp-def* **by** *blast*

moreover
have $L \in$ *lits-of-l* (*raw-trail* (*toS* (*do-other-step S*)))
using *U* **by** *auto*
then have $L \in$ *lits-of-l* (*raw-trail* (*do-other-step S*))
by (*cases do-other-step S*) (*auto simp*: *lits-of-def*)
ultimately show *?thesis*
by *metis*
qed
qed

lemma *do-full1-cp-step-induct*:
$(\bigwedge S.\ (S \neq\ \ do\text{-}cp\text{-}step'\ S \Longrightarrow P\ (do\text{-}cp\text{-}step'\ S)) \Longrightarrow P\ S) \Longrightarrow P\ a0$
using *do-full1-cp-step.induct* **by** *metis*

lemma *do-cp-step-neq-raw-trail-increase*:
$\exists\, c.\ raw\text{-}trail\ (do\text{-}cp\text{-}step\ S) = c\ @\ raw\text{-}trail\ \ S\ \wedge(\forall\, m \in set\ c.\ \neg\ is\text{-}decided\ m)$
by (*cases S*, *cases raw-conflicting S*)
(*auto simp add*: *do-cp-step-def do-conflict-step-def do-propagate-step-def split*: *option.splits*)

lemma *do-full1-cp-step-neq-raw-trail-increase*:
$\exists\, c.\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S)) = c\ @\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)$
$\wedge\ (\forall\, m \in set\ c.\ \neg\ is\text{-}decided\ m)$
apply (*induction rule*: *do-full1-cp-step-induct*)
apply (*rename-tac S*, *case-tac do-cp-step′ S = S*)
apply (*simp add*: *do-full1-cp-step.simps*)
by (*smt Un-iff append-assoc do-cp-step′-def do-cp-step-neq-raw-trail-increase do-full1-cp-step.simps rough-state-of-state-of-do-cp-step set-append*)

lemma *do-cp-step-raw-conflicting*:
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) \neq None \Longrightarrow do\text{-}cp\text{-}step'\ S = S$
unfolding *do-cp-step′-def do-cp-step-def* **by** *simp*

lemma *do-full1-cp-step-raw-conflicting*:
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) \neq None \Longrightarrow do\text{-}full1\text{-}cp\text{-}step\ S = S$
unfolding *do-cp-step′-def do-cp-step-def*
apply (*induction rule*: *do-full1-cp-step-induct*)
by (*rename-tac S*, *case-tac* $S \neq do\text{-}cp\text{-}step'\ S$)
(*auto simp add*: *do-full1-cp-step.simps do-cp-step-raw-conflicting*)

lemma *do-decide-step-not-raw-conflicting-one-more-decide*:
assumes
$raw\text{-}conflicting\ S = None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $Suc\ (length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ S)))$
$= length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
using *assms* **unfolding** *do-other-step′-def*
by (*cases S*) (*auto simp*: *Let-def split*: *if-split-asm option.splits*
dest!: *find-first-unused-var-Some-not-all-incl*)

lemma *do-decide-step-not-raw-conflicting-one-more-decide-bt*:
assumes $raw\text{-}conflicting\ S \neq None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ S)) < length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
using *assms* **unfolding** *do-other-step′-def* **by** (*cases S*, *cases raw-conflicting S*)
(*auto simp add*: *Let-def split*: *if-split-asm option.splits*)

lemma *count-decided-raw-trail-toS*:
$count\text{-}decided\ (raw\text{-}trail\ (toS\ S)) =\ \ count\text{-}decided\ (raw\text{-}trail\ S)$
by (*cases S*) (*auto simp*: *comp-def*)

lemma *do-other-step-not-raw-conflicting-one-more-decide-bt*:
assumes
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ S) \neq None$ **and**
$raw\text{-}conflicting\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S)) = None$ **and**
$do\text{-}other\text{-}step'\ S \neq S$

shows *count-decided* (*raw-trail* (*rough-state-of S*))
  > *count-decided* (*raw-trail* (*rough-state-of* (*do-other-step′ S*)))
proof (*cases S*, *goal-cases*)
  case (*1 y*) **note** *S* = *this*(*1*) **and** *inv* = *this*(*2*)
  obtain *M N U k E* **where** *y*: *y* = (*M*, *N*, *U*, *k*, *Some E*)
    using *assms*(*1*) *S inv* **by** (*cases y*, *cases raw-conflicting y*) *auto*
  have *M*: *rough-state-of* (*state-of* (*M*, *N*, *U*, *k*, *Some E*)) = (*M*, *N*, *U*, *k*, *Some E*)
    using *inv y* **by** (*auto simp add*: *state-of-inverse*)
  have *bt*: *do-other-step′ S* = *state-of* (*do-backtrack-step* (*rough-state-of S*))
    proof (*cases rough-state-of S rule*: *do-decide-step.cases*)
      case *1*
      then show *?thesis*
        using *assms*(*1*,*2*) **by** *auto*[]
    next
      case (*2 v vb vd vf vh*)
      have *f3*: $\bigwedge$*c*. (*if do-skip-step* (*rough-state-of c*) ≠ *rough-state-of c*
        then do-skip-step (*rough-state-of c*)
        else if do-resolve-step (*do-skip-step* (*rough-state-of c*)) ≠ *do-skip-step* (*rough-state-of c*)
            then do-resolve-step (*do-skip-step* (*rough-state-of c*))
            else if do-backtrack-step (*do-resolve-step* (*do-skip-step* (*rough-state-of c*)))
              ≠ *do-resolve-step* (*do-skip-step* (*rough-state-of c*))
            then do-backtrack-step (*do-resolve-step* (*do-skip-step* (*rough-state-of c*)))
            else do-decide-step (*do-backtrack-step* (*do-resolve-step*
              (*do-skip-step* (*rough-state-of c*)))))
        = *rough-state-of* (*do-other-step′ c*)
        by (*simp add*: *rough-state-of-do-other-step′*)
     have (*raw-trail* (*rough-state-of* (*do-other-step′ S*)), *raw-init-clss* (*rough-state-of* (*do-other-step′ S*)),
          raw-learned-clss (*rough-state-of* (*do-other-step′ S*)),
          raw-backtrack-lvl (*rough-state-of* (*do-other-step′ S*)), *None*)
        = *rough-state-of* (*do-other-step′ S*)
        using *assms*(*2*) **by** (*metis* (*no-types*) *state-conv*)
      then show *?thesis*
        using *f3 2* **by** (*metis* (*no-types*) *do-decide-step.simps*(*2*) *do-resolve-step-raw-trail-is-None*
          do-skip-step-raw-trail-is-None rough-state-of-inverse)
    qed
  have
    bt: *raw-backtrack-lvl* (*toS y*) = *count-decided* (*raw-trail* (*toS y*))
    using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def*
    $cdcl_W$*-conflicting-def* **by** *simp-all*
  have *confl-y*: *raw-conflicting* (*toS* (*rough-state-of* (*do-other-step′* (*state-of y*)))) = *None*
   using *assms*(*2*) *y S raw-conflicting-noTrue-iff-toS* **by** *blast*
  have *backtrack* (*toS* (*rough-state-of S*))
     (*toS* (*rough-state-of* (*do-other-step′* (*state-of y*)))) ∨
    resolve (*toS* (*rough-state-of S*))
     (*toS* (*rough-state-of* (*do-other-step′* (*state-of y*)))) ∨
    skip (*toS* (*rough-state-of S*))
     (*toS* (*rough-state-of* (*do-other-step′* (*state-of y*))))
    proof −
      have *f1*: (*M*, *N*, *U*, *k*, *Some E*) = *rough-state-of S*
        by (*simp add*: *M S y*)
      then have *f2*: *do-other-step* (*M*, *N*, *U*, *k*, *Some E*) ≠ (*M*, *N*, *U*, *k*, *Some E*)
        by (*metis assms*(*3*) *rough-state-of-do-other-step′ rough-state-of-inject*)
      have $cdcl_W$*-all-struct-inv* (*toS* (*M*, *N*, *U*, *k*, *Some E*))
        using *f1* **by** *simp*
      then have $cdcl_W$*-o* (*toS* (*M*, *N*, *U*, *k*, *Some E*)) (*toS* (*do-other-step* (*M*, *N*, *U*, *k*, *Some E*)))
        using *f2 do-other-step* **by** *blast*

then have *f3*: *cdcl*$_W$*-o* (*toS* (*rough-state-of S*))
(*toS* (*rough-state-of* (*do-other-step*′ (*state-of* (*M*, *N*, *U*, *k*, *Some E*)))))
using *f1* **by** (*simp add*: *rough-state-of-do-other-step*′)
have ¬ *decide* (*toS* (*rough-state-of S*))
(*toS* (*rough-state-of* (*do-other-step*′ (*state-of* (*M*, *N*, *U*, *k*, *Some E*)))))
using *f1* **by** (*metis* (*no-types*) *do-decide-step.simps*(*2*) *do-decide-step-no*)
then show *?thesis*
using *f3 cdcl*$_W$*-o-rule-cases y* **by** *blast*
qed
then have *bt*: *backtrack* (*toS* (*rough-state-of S*))
(*toS* (*rough-state-of* (*do-other-step*′ (*state-of y*))))
using *confl-y* **by** (*cases rough-state-of S*) (*auto elim*!: *resolveE skipE*)
moreover
have *no-dup* (*raw-trail* (*rough-state-of S*))
using *rough-state-of*[*of S*] **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def*
by (*cases S*) (*auto simp*: *comp-def*)
have *cdcl*$_W$*-M-level-inv* (*toS* (*rough-state-of S*)) **and**
cdcl$_W$*-M-level-inv* (*toS* (*rough-state-of* (*do-other-step*′ (*state-of y*))))
using *inv* **apply** (*simp add*: *cdcl*$_W$*-all-struct-inv-def S*)
using *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-all-struct-inv-rough-state* **by** *blast*
then show *?case*
using *backtrack-lvl-backtrack-decrease*[*OF - bt*]
using *S* **unfolding** *cdcl*$_W$*-M-level-inv-def*
by (*simp add*: *comp-def count-decided-raw-trail-toS*)
qed

lemma *do-other-step-not-raw-conflicting-one-more-decide*:
assumes *raw-conflicting* (*rough-state-of S*) = *None* **and**
do-other-step′ *S* ≠ *S*
shows *1* + *length* (*filter is-decided* (*raw-trail* (*rough-state-of S*)))
= *length* (*filter is-decided* (*raw-trail* (*rough-state-of* (*do-other-step*′ *S*))))
proof (*cases S*, *goal-cases*)
case (*1 y*) **note** *S* = *this*(*1*) **and** *inv* = *this*(*2*)
obtain *M N U k* **where** *y*: *y* = (*M*, *N*, *U*, *k*, *None*) **using** *assms*(*1*) *S inv* **by** (*cases y*) *auto*
have *M*: *rough-state-of* (*state-of* (*M*, *N*, *U*, *k*, *None*)) = (*M*, *N*, *U*, *k*, *None*)
using *inv y* **by** (*auto simp add*: *state-of-inverse*)
have *state-of* (*do-decide-step* (*M*, *N*, *U*, *k*, *None*)) ≠ *state-of* (*M*, *N*, *U*, *k*, *None*)
using *assms*(*2*) **unfolding** *do-other-step*′*-def y inv S* **by** (*auto simp add*: *M*)
then have *f4*: *do-skip-step* (*rough-state-of S*) = *rough-state-of S*
unfolding *S M y* **by** (*metis* (*full-types*) *do-skip-step.simps*(*4*))
have *f5*: *do-resolve-step* (*rough-state-of S*) = *rough-state-of S*
unfolding *S M y* **by** (*metis* (*no-types*) *do-resolve-step.simps*(*4*))
have *f6*: *do-backtrack-step* (*rough-state-of S*) = *rough-state-of S*
unfolding *S M y* **by** (*metis* (*no-types*) *do-backtrack-step.simps*(*2*))
have *do-other-step* (*rough-state-of S*) ≠ *rough-state-of S*
using *assms*(*2*) **unfolding** *S M y do-other-step*′*-def* **by** (*metis* (*no-types*))
then show *?case*
using *f6 f5 f4* **by** (*simp add*: *assms*(*1*) *do-decide-step-not-raw-conflicting-one-more-decide*
do-other-step′*-def*)
qed

lemma *rough-state-of-state-of-do-skip-step-rough-state-of*[*simp*]:
rough-state-of (*state-of* (*do-skip-step* (*rough-state-of S*))) = *do-skip-step* (*rough-state-of S*)
by (*smt do-other-step.simps rough-state-of-inverse rough-state-of-state-of-do-other-step*)

lemma *raw-conflicting-do-resolve-step-iff*[*iff*]:

raw-conflicting (*do-resolve-step S*) = *None* ⟷ *raw-conflicting S* = *None*
by (*cases S rule*: *do-resolve-step.cases*)
(*auto simp add*: *Let-def split*: *option.splits*)

lemma *raw-conflicting-do-skip-step-iff* [*iff*]:
raw-conflicting (*do-skip-step S*) = *None* ⟷ *raw-conflicting S* = *None*
by (*cases S rule*: *do-skip-step.cases*)
(*auto simp add*: *Let-def split*: *option.splits*)

lemma *raw-conflicting-do-decide-step-iff* [*iff*]:
raw-conflicting (*do-decide-step S*) = *None* ⟷ *raw-conflicting S* = *None*
by (*cases S rule*: *do-decide-step.cases*)
(*auto simp add*: *Let-def split*: *option.splits*)

lemma *raw-conflicting-do-backtrack-step-imp*[*simp*]:
do-backtrack-step S ≠ *S* ⟹ *raw-conflicting* (*do-backtrack-step S*) = *None*
by (*cases S rule*: *do-backtrack-step.cases*)
(*auto simp add*: *Let-def split*: *list.splits option.splits ann-lit.splits*)

lemma *do-skip-step-eq-iff-raw-trail-eq*:
do-skip-step S = *S* ⟷ *raw-trail* (*do-skip-step S*) = *raw-trail S*
by (*cases S rule*: *do-skip-step.cases*) *auto*

lemma *do-decide-step-eq-iff-raw-trail-eq*:
do-decide-step S = *S* ⟷ *raw-trail* (*do-decide-step S*) = *raw-trail S*
by (*cases S rule*: *do-decide-step.cases*) (*auto split*: *option.split*)

lemma *do-backtrack-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail S*)
shows *do-backtrack-step S* = *S* ⟷ *raw-trail* (*do-backtrack-step S*) = *raw-trail S*
using *assms* **apply** (*cases S rule*: *do-backtrack-step.cases*)
by (*auto split*: *option.split list.splits ann-lit.splits*
simp: *comp-def*
dest!: *bt-cut-in-get-all-ann-decomposition*)

lemma *do-resolve-step-eq-iff-raw-trail-eq*:
do-resolve-step S = *S* ⟷ *raw-trail* (*do-resolve-step S*) = *raw-trail S*
by (*cases S rule*: *do-resolve-step.cases*) *auto*

lemma *do-other-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail S*)
shows *raw-trail* (*do-other-step S*) = *raw-trail S* ⟷ *do-other-step S* = *S*
using *assms*
by (*auto simp add*: *Let-def do-skip-step-eq-iff-raw-trail-eq*[*symmetric*]
do-decide-step-eq-iff-raw-trail-eq[*symmetric*] *do-backtrack-step-eq-iff-raw-trail-eq*[*symmetric*]
do-resolve-step-eq-iff-raw-trail-eq[*symmetric*])

lemma *do-full1-cp-step-do-other-step'-normal-form*[*dest*!]:
assumes *H*: *do-full1-cp-step* (*do-other-step' S*) = *S*
shows *do-other-step' S* = *S* ∧ *do-full1-cp-step S* = *S*
proof −
let *?T* = *do-other-step' S*
{ assume *confl*: *raw-conflicting* (*rough-state-of ?T*) ≠ *None*
then have *tr*: *raw-trail* (*rough-state-of* (*do-full1-cp-step ?T*)) = *raw-trail* (*rough-state-of ?T*)
using *do-full1-cp-step-raw-conflicting*[*of ?T*] **by** *auto*

have *raw-trail* (*rough-state-of* (*do-full1-cp-step* (*do-other-step'* *S*))) = *raw-trail* (*rough-state-of* *S*)
using *arg-cong*[*OF H*, *of* λ*S*. *raw-trail* (*rough-state-of* *S*)] **.**
then have *raw-trail* (*rough-state-of* (*do-other-step'* *S*)) = *raw-trail* (*rough-state-of* *S*)
by (*auto simp add*: *do-full1-cp-step-raw-conflicting confl*)
then have *do-other-step'* *S* = *S*
using *assms confl*
by (*simp add*: *do-other-step-eq-iff-raw-trail-eq do-other-step'-def*
do-full1-cp-step-raw-conflicting
del: *do-other-step.simps*)

}
moreover {
assume *eq*[*simp*]: *do-other-step'* *S* = *S*
obtain *c* **where** *c*: *raw-trail* (*rough-state-of* (*do-full1-cp-step* *S*)) = *c* @ *raw-trail* (*rough-state-of* *S*)
using *do-full1-cp-step-neq-raw-trail-increase* **by** *auto*

moreover have *raw-trail* (*rough-state-of* (*do-full1-cp-step* *S*)) = *raw-trail* (*rough-state-of* *S*)
using *arg-cong*[*OF H*, *of* λ*S*. *raw-trail* (*rough-state-of* *S*)] **by** *simp*
finally have *c* = [] **by** *blast*
then have *do-full1-cp-step* *S* = *S* **using** *assms* **by** *auto*
}
moreover {
assume *confl*: *raw-conflicting* (*rough-state-of* *?T*) = *None* **and** *neq*: *do-other-step'* *S* ≠ *S*
obtain *c* **where**
c: *raw-trail* (*rough-state-of* (*do-full1-cp-step* *?T*)) = *c* @ *raw-trail* (*rough-state-of* *?T*) **and**
nm: ∀ *m*∈*set c*. ¬ *is-decided m*
using *do-full1-cp-step-neq-raw-trail-increase* **by** *auto*
have *length* (*filter is-decided* (*raw-trail* (*rough-state-of* (*do-full1-cp-step* *?T*))))
= *length* (*filter is-decided* (*raw-trail* (*rough-state-of* *?T*))) **using** *nm* **unfolding** *c* **by** *force*
moreover have *length* (*filter is-decided* (*raw-trail* (*rough-state-of* *S*)))
≠ *length* (*filter is-decided* (*raw-trail* (*rough-state-of* *?T*)))
using *do-other-step-not-raw-conflicting-one-more-decide*[*OF - neq*]
do-other-step-not-raw-conflicting-one-more-decide-bt[*of S*, *OF - confl neq*]
by *linarith*
finally have *False* **unfolding** *H* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

lemma *do-cdcl*$_W$*-stgy-step-no*:
assumes *S*: *do-cdcl*$_W$*-stgy-step* *S* = *S*
shows *no-step cdcl*$_W$*-stgy* (*toS* (*rough-state-of* *S*))
proof −
{
fix *S'*
assume *full1 cdcl*$_W$*-cp* (*toS* (*rough-state-of* *S*)) *S'*
then have *False*
using *do-full1-cp-step-full*[*of S*] **unfolding** *full-def S rtranclp-unfold full1-def*
by (*smt assms do-cdcl*$_W$*-stgy-step-def tranclpD*)
}
moreover {
fix *S' S''*
assume *cdcl*$_W$*-o* (*toS* (*rough-state-of* *S*)) *S'* **and**
no-step propagate (*toS* (*rough-state-of* *S*)) **and**
no-step conflict (*toS* (*rough-state-of* *S*)) **and**
full cdcl$_W$*-cp S' S''*

then have *False*
using *assms* **unfolding** *do-cdcl$_W$-stgy-step-def*
by (*smt cdcl$_W$-all-struct-inv-rough-state do-full1-cp-step-do-other-step'-normal-form do-other-step-no rough-state-of-do-other-step'*)
}
ultimately show *?thesis* **using** *assms* **by** (*force simp*: *cdcl$_W$-cp.simps cdcl$_W$-stgy.simps*)
qed

lemma *toS-rough-state-of-state-of-rough-state-from-init-state-of*[*simp*]:
toS (*rough-state-of* (*state-of* (*rough-state-from-init-state-of S*)))
= *toS* (*rough-state-from-init-state-of S*)
using *rough-state-from-init-state-of*[*of S*] **by** (*auto simp add*: *state-of-inverse*)

lemma *cdcl$_W$-cp-is-rtranclp-cdcl$_W$*: *cdcl$_W$-cp S T* $\Longrightarrow$ *cdcl$_W$*** *S T*
apply (*induction rule*: *cdcl$_W$-cp.induct*)
using *conflict* **apply** *blast*
using *propagate* **by** *blast*

lemma *rtranclp-cdcl$_W$-cp-is-rtranclp-cdcl$_W$*: *cdcl$_W$-cp*** *S T* $\Longrightarrow$ *cdcl$_W$*** *S T*
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
by (*fastforce dest*!: *cdcl$_W$-cp-is-rtranclp-cdcl$_W$*)

lemma *cdcl$_W$-stgy-is-rtranclp-cdcl$_W$*:
cdcl$_W$-stgy S T $\Longrightarrow$ *cdcl$_W$*** *S T*
apply (*induction rule*: *cdcl$_W$-stgy.induct*)
using *cdcl$_W$-stgy.conflict' rtranclp-cdcl$_W$-stgy-rtranclp-cdcl$_W$* **apply** *blast*
unfolding *full-def* **by** (*fastforce dest*!:*other rtranclp-cdcl$_W$-cp-is-rtranclp-cdcl$_W$*)

lemma *cdcl$_W$-stgy-init-raw-init-clss*:
cdcl$_W$-stgy S T $\Longrightarrow$ *cdcl$_W$-M-level-inv S* $\Longrightarrow$ *raw-init-clss S* = *raw-init-clss T*
using *cdcl$_W$-stgy-no-more-init-clss* **by** *blast*

lemma *clauses-toS-rough-state-of-do-cdcl$_W$-stgy-step*[*simp*]:
raw-init-clss (*toS* (*rough-state-of* (*do-cdcl$_W$-stgy-step* (*state-of* (*rough-state-from-init-state-of S*)))))
= *raw-init-clss* (*toS* (*rough-state-from-init-state-of S*)) (**is** - = *raw-init-clss* (*toS ?S*))
apply (*cases do-cdcl$_W$-stgy-step* (*state-of ?S*) = *state-of ?S*)
apply *simp*
by (*metis cdcl$_W$-all-struct-inv-def cdcl$_W$-all-struct-inv-rough-state cdcl$_W$-stgy-no-more-init-clss do-cdcl$_W$-stgy-step toS-rough-state-of-state-of-rough-state-from-init-state-of*)

lemma *rough-state-from-init-state-of-do-cdcl$_W$-stgy-step'*[*code abstract*]:
rough-state-from-init-state-of (*do-cdcl$_W$-stgy-step' S*) =
rough-state-of (*do-cdcl$_W$-stgy-step* (*id-of-I-to S*))
proof −
let *?S* = (*rough-state-from-init-state-of S*)
have *cdcl$_W$-stgy*** (*S0-cdcl$_W$* (*raw-init-clss* (*toS* (*rough-state-from-init-state-of S*))))
(*toS* (*rough-state-from-init-state-of S*))
using *rough-state-from-init-state-of*[*of S*] **by** *auto*
moreover have *cdcl$_W$-stgy***
(*toS* (*rough-state-from-init-state-of S*))
(*toS* (*rough-state-of* (*do-cdcl$_W$-stgy-step*
(*state-of* (*rough-state-from-init-state-of S*)))))
using *do-cdcl$_W$-stgy-step*[*of state-of ?S*]
by (*cases do-cdcl$_W$-stgy-step* (*state-of ?S*) = *state-of ?S*) *auto*

ultimately show *?thesis*
unfolding *do-cdcl$_W$-stgy-step'-def id-of-I-to-def*
by (*auto intro*!: *state-from-init-state-of-inverse*)
qed

All rules together **function** *do-all-cdcl$_W$-stgy* **where**
do-all-cdcl$_W$-stgy S =
(*let T* = *do-cdcl$_W$-stgy-step' S in*
if T = *S then S else do-all-cdcl$_W$-stgy T*)
by *fast*+
termination
proof (*relation* {(*T*, *S*).
(*cdcl$_W$-measure* (*toS* (*rough-state-from-init-state-of T*)),
cdcl$_W$-measure (*toS* (*rough-state-from-init-state-of S*)))
∈ *lexn less-than 3*}, *goal-cases*)
case *1*
show *?case* **by** (*rule wf-if-measure-f*) (*auto intro*!: *wf-lexn wf-less*)
next
case (*2 S T*) **note** *T* = *this*(*1*) **and** *ST* = *this*(*2*)
let *?S* = *rough-state-from-init-state-of S*
have *S*: *cdcl$_W$-stgy*** (*S0-cdcl$_W$* (*raw-init-clss* (*toS ?S*))) (*toS ?S*)
using *rough-state-from-init-state-of*[*of S*] **by** *auto*
moreover have *cdcl$_W$-stgy* (*toS* (*rough-state-from-init-state-of S*))
(*toS* (*rough-state-from-init-state-of T*))
proof −
have $\bigwedge$*c*. *rough-state-of* (*state-of* (*rough-state-from-init-state-of c*)) =
rough-state-from-init-state-of c
using *rough-state-from-init-state-of state-of-inverse* **by** *fastforce*
then have *diff*: *do-cdcl$_W$-stgy-step* (*state-of* (*rough-state-from-init-state-of S*))
≠ *state-of* (*rough-state-from-init-state-of S*)
using *ST T* **by** (*metis* (*no-types*) *id-of-I-to-def rough-state-from-init-state-of-inject*
rough-state-from-init-state-of-do-cdcl$_W$-stgy-step')
have *rough-state-of* (*do-cdcl$_W$-stgy-step* (*state-of* (*rough-state-from-init-state-of S*)))
= *rough-state-from-init-state-of* (*do-cdcl$_W$-stgy-step' S*)
by (*simp add*: *id-of-I-to-def rough-state-from-init-state-of-do-cdcl$_W$-stgy-step'*)
then show *?thesis*
using *do-cdcl$_W$-stgy-step T diff* **unfolding** *id-of-I-to-def do-cdcl$_W$-stgy-step* **by** *fastforce*
qed
moreover
have *cdcl$_W$-all-struct-inv* (*toS* (*rough-state-from-init-state-of S*))
using *rough-state-from-init-state-of*[*of S*] **by** *auto*
then have *cdcl$_W$-all-struct-inv* (*S0-cdcl$_W$* (*raw-init-clss* (*toS* (*rough-state-from-init-state-of S*))))
by (*cases rough-state-from-init-state-of S*)
(*auto simp add*: *cdcl$_W$-all-struct-inv-def distinct-cdcl$_W$-state-def*)
ultimately show *?case*
using *tranclp-cdcl$_W$-stgy-S0-decreasing*
by (*auto intro*!: *cdcl$_W$-stgy-step-decreasing*[*of - - S0-cdcl$_W$* (*raw-init-clss* (*toS ?S*))]
simp del: *cdcl$_W$-measure.simps*)
qed

thm *do-all-cdcl$_W$-stgy.induct*
lemma *do-all-cdcl$_W$-stgy-induct*:
($\bigwedge$*S*. (*do-cdcl$_W$-stgy-step' S* ≠ *S* ⟹ *P* (*do-cdcl$_W$-stgy-step' S*)) ⟹ *P S*) ⟹ *P a0*
using *do-all-cdcl$_W$-stgy.induct* **by** *metis*

lemma *no-step-cdcl$_W$-stgy-cdcl$_W$-all*:

fixes S :: *'a cdcl*$_W$*-state-inv-from-init-state*
shows *no-step cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy S*)))
apply (*induction S rule*:*do-all-cdcl*$_W$*-stgy-induct*)
apply (*rename-tac S*, *case-tac do-cdcl*$_W$*-stgy-step' S* $\neq$ *S*)
proof −
fix *Sa* :: *'a cdcl*$_W$*-state-inv-from-init-state*
assume *a1*: $\neg$ *do-cdcl*$_W$*-stgy-step' Sa* $\neq$ *Sa*
{ fix *pp*
have (*if True then Sa else do-all-cdcl*$_W$*-stgy Sa*) = *do-all-cdcl*$_W$*-stgy Sa*
using *a1* **by** *auto*
then have $\neg$ *cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy Sa*))) *pp*
using *a1* **by** (*metis* (*no-types*) *do-cdcl*$_W$*-stgy-step-no id-of-I-to-def*
rough-state-from-init-state-of-do-cdcl$_W$*-stgy-step' rough-state-of-inverse*) **}**
then show *no-step cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy Sa*)))
by *fastforce*
next
fix *Sa* :: *'a cdcl*$_W$*-state-inv-from-init-state*
assume *a1*: *do-cdcl*$_W$*-stgy-step' Sa* $\neq$ *Sa*
$\Longrightarrow$ *no-step cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of*
(*do-all-cdcl*$_W$*-stgy* (*do-cdcl*$_W$*-stgy-step' Sa*))))
assume *a2*: *do-cdcl*$_W$*-stgy-step' Sa* $\neq$ *Sa*
have *do-all-cdcl*$_W$*-stgy Sa* = *do-all-cdcl*$_W$*-stgy* (*do-cdcl*$_W$*-stgy-step' Sa*)
by (*metis* (*full-types*) *do-all-cdcl*$_W$*-stgy.simps*)
then show *no-step cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy Sa*)))
using *a2 a1* **by** *presburger*
qed

lemma *do-all-cdcl*$_W$*-stgy-is-rtranclp-cdcl*$_W$*-stgy*:
cdcl$_W$*-stgy*** (*toS* (*rough-state-from-init-state-of S*))
(*toS* (*rough-state-from-init-state-of* (*do-all-cdcl*$_W$*-stgy S*)))
proof (*induction S rule*: *do-all-cdcl*$_W$*-stgy-induct*)
case (*1 S*) **note** *IH* = *this*(*1*)
show *?case*
proof (*cases do-cdcl*$_W$*-stgy-step' S* = *S*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *f2*: *do-cdcl*$_W$*-stgy-step* (*id-of-I-to S*) = *id-of-I-to S* $\longrightarrow$
rough-state-from-init-state-of (*do-cdcl*$_W$*-stgy-step' S*)
= *rough-state-of* (*state-of* (*rough-state-from-init-state-of S*))
using *rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step'*
by (*simp add*: *id-of-I-to-def rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step'*)
have *f3*: *do-all-cdcl*$_W$*-stgy S* = *do-all-cdcl*$_W$*-stgy* (*do-cdcl*$_W$*-stgy-step' S*)
by (*metis* (*full-types*) *do-all-cdcl*$_W$*-stgy.simps*)
have *cdcl*$_W$*-stgy* (*toS* (*rough-state-from-init-state-of S*))
(*toS* (*rough-state-from-init-state-of* (*do-cdcl*$_W$*-stgy-step' S*)))
= *cdcl*$_W$*-stgy* (*toS* (*rough-state-of* (*id-of-I-to S*)))
(*toS* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*id-of-I-to S*))))
using *rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step'*
toS-rough-state-of-state-of-rough-state-from-init-state-of
by (*simp add*: *id-of-I-to-def rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step'*)
then show *?thesis*
using *f3 f2 IH do-cdcl*$_W$*-stgy-step* **by** *fastforce*
qed
qed

Final theorem:

```
lemma DPLL-tot-correct:
  assumes
    r: rough-state-from-init-state-of (do-all-cdcl_W-stgy (state-from-init-state-of
      (([], map remdups N, [], 0, None)))) = S and
    S: (M', N', U', k, E) = toS S
  shows (E ≠ Some {#} ∧ satisfiable (set (map mset N)))
    ∨ (E = Some {#} ∧ unsatisfiable (set (map mset N)))
proof -
  let ?N = map remdups N
  have inv: cdcl_W-all-struct-inv (toS ([], map remdups N, [], 0, None))
    unfolding cdcl_W-all-struct-inv-def distinct-cdcl_W-state-def distinct-mset-set-def by auto
  then have S0: rough-state-of (state-of ([], map remdups N, [], 0, None))
    = ([], map remdups N, [], 0, None) by simp
  have 1: full cdcl_W-stgy (toS ([], ?N, [], 0, None)) (toS S)
    unfolding full-def apply rule
      using do-all-cdcl_W-stgy-is-rtranclp-cdcl_W-stgy[of
        state-from-init-state-of ([], map remdups N, [], 0, None)] inv
        no-step-cdcl_W-stgy-cdcl_W-all
        apply (auto simp del: do-all-cdcl_W-stgy.simps simp: state-from-init-state-of-inverse
          r[symmetric] comp-def)[]
      using do-all-cdcl_W-stgy-is-rtranclp-cdcl_W-stgy[of
      state-from-init-state-of ([], map remdups N, [], 0, None)] inv
      no-step-cdcl_W-stgy-cdcl_W-all
      by (force simp: state-from-init-state-of-inverse r[symmetric] comp-def)
  moreover have 2: finite (set (map mset ?N)) by auto
  moreover have 3: distinct-mset-set (set (map mset ?N))
     unfolding distinct-mset-set-def by auto
  moreover
    have cdcl_W-all-struct-inv (toS S)
      by (metis (no-types) cdcl_W-all-struct-inv-rough-state r
        toS-rough-state-of-state-of-rough-state-from-init-state-of)
    then have cons: consistent-interp (lits-of-l M')
      unfolding cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def S[symmetric] by auto
  moreover
    have raw-init-clss (toS ([], ?N, [], 0, None)) = raw-init-clss (toS S)
      apply (rule rtranclp-cdcl_W-stgy-no-more-init-clss)
      using 1 unfolding full-def by (auto simp add: rtranclp-cdcl_W-stgy-rtranclp-cdcl_W)
    then have N': mset (map mset ?N) = N'
      using S[symmetric] by auto
  have (E ≠ Some {#} ∧ satisfiable (set (map mset ?N)))
    ∨ (E = Some {#} ∧ unsatisfiable (set (map mset ?N)))
    using full-cdcl_W-stgy-final-state-conclusive unfolding N' apply rule
        using 1 apply simp
       using 2 apply simp
      using 3 apply simp
     using S[symmetric] N' apply auto[1]
   using S[symmetric] N' cons by (fastforce simp: true-annots-true-cls)
  then show ?thesis by auto
qed
```

The Code The SML code is skipped in the documentation, but stays to ensure that some version of the exported code is working. The only difference between the generated code and the one used here is the export of the constructor ConI.

end
theory *CDCL-Abstract-Clause-Representation*
imports *Main Partial-Clausal-Logic*
begin

type-synonym $'v$ *clause* = $'v$ *literal multiset*
type-synonym $'v$ *clauses* = $'v$ *clause multiset*

7.1.6 Abstract Clause Representation

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

We assume the following:

- there is an equivalent to adding and removing a literal and to taking the union of clauses.

locale *raw-cls* =
  fixes
    mset-cls :: $'cls \Rightarrow 'v$ *clause*
begin
end

The two following locales are the *exact same* locale, but we two different locales. Otherwise, instantiating *raw-clss* would lead to duplicate constants. (TODO: better idea?).

locale *abstract-with-index* =
  fixes
    get :: $'a \Rightarrow 'it \Rightarrow 'conc$ **and**
    valid :: $'it \Rightarrow 'a \Rightarrow bool$ **and**
    convert-to-mset :: $'a \Rightarrow 'conc$ *multiset*
  assumes
    in-clss-mset-cls[*dest*]:
      valid a Cs $\Longrightarrow$ *get Cs a* $\in\#$ *convert-to-mset Cs* **and**
    in-mset-cls-exists-preimage:
      $b \in\#$ *convert-to-mset Cs* $\Longrightarrow \exists b'.$ *valid* b' *Cs* $\wedge$ *get Cs* $b' = b$

locale *abstract-with-index2* =
  fixes
    get :: $'a \Rightarrow 'it \Rightarrow 'conc$ **and**
    valid :: $'it \Rightarrow 'a \Rightarrow bool$ **and**
    convert-to-mset :: $'a \Rightarrow 'conc$ *multiset*
  assumes
    in-clss-mset-clss[*dest*]:
      valid a Cs $\Longrightarrow$ *get Cs a* $\in\#$ *convert-to-mset Cs* **and**
    in-mset-clss-exists-preimage:
      $b \in\#$ *convert-to-mset Cs* $\Longrightarrow \exists b'.$ *valid* b' *Cs* $\wedge$ *get Cs* $b' = b$

locale *raw-clss* =
  abstract-with-index cls-lit in-cls mset-cls +
  abstract-with-index2 clss-cls in-clss mset-clss
  for
    cls-lit :: $'cls \Rightarrow 'lit \Rightarrow 'v$ *literal* **and**
    in-cls :: $'lit \Rightarrow 'cls \Rightarrow bool$ **and**
    mset-cls :: $'cls \Rightarrow 'v$ *clause* **and**

clss-cls :: *'clss* ⇒ *'cls-it* ⇒ *'cls* **and**
in-clss :: *'cls-it* ⇒ *'clss* ⇒ *bool* **and**
mset-clss:: *'clss* ⇒ *'cls multiset*
begin
notation *in-cls* (**infix** $\in\downarrow$ *49*)
notation *in-clss* (**infix** $\in\Downarrow$ *49*)

notation *cls-lit* (**infix** $\downarrow$ *49*)
notation *clss-cls* (**infix** $\Downarrow$ *49*)

abbreviation *raw-clss* **where**
raw-clss S ≡ *image-mset mset-cls* (*mset-clss S*)
end

experiment
begin
interpretation *abstract-with-index*
nth
$\lambda L\ C.\ L < \mathit{length}\ C$
mset
apply *unfold-locales*
by (*metis in-set-conv-nth set-mset-mset*)+

interpretation *abstract-with-index2*
nth
$\lambda L\ C.\ L < \mathit{length}\ C$
mset
apply *unfold-locales*
by (*metis in-set-conv-nth set-mset-mset*)+

interpretation *list-cls*: *raw-clss*
nth
$\lambda L\ C.\ L < \mathit{length}\ C$
mset

nth
$\lambda C\ Cs.\ C < \mathit{length}\ Cs$
$\lambda Cs.\ \mathit{mset}\ Cs$
by *unfold-locales*

end

end
theory *CDCL-W-Abstract-State*
imports *CDCL-Abstract-Clause-Representation CDCL-WNOT*

begin

7.2 Weidenbach's CDCL with Abstract Clause Representation

We first instantiate the locale of Weidenbach's locale. Then we define another abstract state: the goal of this state is to be used for implementations. We add more assumptions on the function about the state. For example *cons-trail* is restricted to undefined literals.

7.2.1 Instantiation of the Multiset Version

type-synonym *'v* $cdcl_W$*-mset* = (*'v*, *'v clause*) *ann-lit list* ×
'v clauses ×
'v clauses ×
nat × *'v clause option*

We use definition, otherwise we could not use the simplification theorems we have already shown.

definition *trail* :: *'v* $cdcl_W$*-mset* ⇒ (*'v*, *'v clause*) *ann-lit list* **where**
trail ≡ λ(*M*, -). *M*

definition *init-clss* :: *'v* $cdcl_W$*-mset* ⇒ *'v clauses* **where**
init-clss ≡ λ(-, *N*, -). *N*

definition *learned-clss* :: *'v* $cdcl_W$*-mset* ⇒ *'v clauses* **where**
learned-clss ≡ λ(-, -, *U*, -). *U*

definition *backtrack-lvl* :: *'v* $cdcl_W$*-mset* ⇒ *nat* **where**
backtrack-lvl ≡ λ(-, -, -, *k*, -). *k*

definition *conflicting* :: *'v* $cdcl_W$*-mset* ⇒ *'v clause option* **where**
conflicting ≡ λ(-, -, -, -, *C*). *C*

definition *cons-trail* :: (*'v*, *'v clause*) *ann-lit* ⇒ *'v* $cdcl_W$*-mset* ⇒ *'v* $cdcl_W$*-mset* **where**
cons-trail ≡ λ*L* (*M*, *R*). (*L* # *M*, *R*)

definition *tl-trail* **where**
tl-trail ≡ λ(*M*, *R*). (*tl M*, *R*)

definition *add-learned-cls* **where**
add-learned-cls ≡ λ*C* (*M*, *N*, *U*, *R*). (*M*, *N*, {#*C*#} + *U*, *R*)

definition *remove-cls* **where**
remove-cls ≡ λ*C* (*M*, *N*, *U*, *R*). (*M*, *removeAll-mset C N*, *removeAll-mset C U*, *R*)

definition *update-backtrack-lvl* **where**
update-backtrack-lvl ≡ λ*k* (*M*, *N*, *U*, -, *D*). (*M*, *N*, *U*, *k*, *D*)

definition *update-conflicting* **where**
update-conflicting ≡ λ*D* (*M*, *N*, *U*, *k*, -). (*M*, *N*, *U*, *k*, *D*)

definition *init-state* **where**
init-state ≡ λ*N*. ([], *N*, {#}, *0*, *None*)

lemmas $cdcl_W$*-mset-state* = *trail-def cons-trail-def tl-trail-def add-learned-cls-def*
remove-cls-def update-backtrack-lvl-def update-conflicting-def init-clss-def learned-clss-def
backtrack-lvl-def conflicting-def init-state-def

interpretation $cdcl_W$*-mset*: $state_W$*-ops* **where**
trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**

tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
.

interpretation $cdcl_W$*-mset*: $state_W$ **where**
trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
by *unfold-locales* (*auto simp*: $cdcl_W$*-mset-state*)

interpretation $cdcl_W$*-mset*: *conflict-driven-clause-learning*$_W$ **where**
trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
by *unfold-locales auto*

lemma $cdcl_W$*-mset-state-eq-eq*: $cdcl_W$*-mset.state-eq* = (*op* =)
apply (*intro ext*)
unfolding $cdcl_W$*-mset.state-eq-def*
by (*auto simp*: $cdcl_W$*-mset-state*)

notation $cdcl_W$*-mset.state-eq* (**infix** ∼*m 49*)

7.2.2 Abstract Relation and Relation Theorems

This locales makes the lifting from the relation defined with multiset *R* and the version with an abstract state *R-abs*. We are lifting many different relations (each rule and the the strategy).

locale *relation-implied-relation-abs* =
fixes
R :: *'v* $cdcl_W$*-mset* ⇒ *'v* $cdcl_W$*-mset* ⇒ *bool* **and**
R-abs :: *'st* ⇒ *'st* ⇒ *bool* **and**

state :: *'st* ⇒ *'v* $cdcl_W$*-mset* **and**
inv :: *'v* $cdcl_W$*-mset* ⇒ *bool*
assumes
relation-compatible-state:
inv (*state S*) ⟹ *R-abs S T* ⟹ *R* (*state S*) (*state T*) **and**
relation-compatible-abs:
⋀*S S' T*. *inv S* ⟹ *S* ∼*m state S'* ⟹ *R S T* ⟹ ∃ *U*. *R-abs S' U* ∧ *T* ∼*m state U* **and**
relation-invariant:
⋀*S T*. *R S T* ⟹ *inv S* ⟹ *inv T* **and**
relation-abs-right-compatible:
⋀*S T U*. *inv* (*state S*) ⟹ *R-abs S T* ⟹ *state T* ∼*m state U* ⟹ *R-abs S U*
begin

lemma *relation-compatible-eq*:
assumes
inv: *inv* (*state S*) **and**
abs: *R-abs S T* **and**
SS': *state S* ∼*m state S'* **and**
TT': *state T* ∼*m state T'*
shows *R-abs S' T'*
proof −
have *R* (*state S*) (*state T*)
using *relation-compatible-state inv abs* **by** *blast*
then obtain *U* **where** *S'U*: *R-abs S' U* **and** *TU*: *state T* ∼*m state U*
using *relation-compatible-abs*[*OF inv SS'*] **by** *blast*
then show *?thesis*
using *relation-abs-right-compatible*[*OF - S'U*, *of T'*] *TT' inv SS'*[*unfolded* $cdcl_W$*-mset-state-eq-eq*]
$cdcl_W$*-mset.state-eq-trans*[*of state T' state T state U*]
by (*auto simp add*: $cdcl_W$*-mset.state-eq-sym*)
qed

lemma *rtranclp-relation-invariant*:
R^{++} *S T* ⟹ *inv S* ⟹ *inv T*
by (*induction rule*: *tranclp-induct*) (*auto simp*: *relation-invariant*)

lemma *rtranclp-abs-rtranclp*:
R-abs** *S T* ⟹ *inv* (*state S*) ⟹ R^{**} (*state S*) (*state T*)
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
by (*metis relation-compatible-state rtranclp.simps rtranclpD rtranclp-relation-invariant*)

lemma *tranclp-relation-tranclp-relation-abs-compatible*:
fixes *S* :: *'st*
assumes
R: R^{++} (*state S*) *T* **and**
inv: *inv* (*state S*)
shows ∃ *U*. *R-abs*$^{++}$ *S U* ∧ *T* ∼*m state U*
using *R*
proof (*induction rule*: *tranclp-induct*)
case (*base T*)
then show *?case*
using *relation-compatible-abs*[*of state S S T*] *inv* **by** *auto*
next
case (*step T U*) **note** *st* = *this*(*1*) **and** *R* = *this*(*2*) **and** *IH* = *this*(*3*)
obtain *V* **where**
SV: *R-abs*$^{++}$ *S V* **and** *TV*: *T* ∼*m state V*

using *IH* **by** *auto*
then obtain *W* **where**
VW: *R-abs V W* **and** *UW*: *U* $\sim$*m state W*
using *relation-compatible-abs*[*OF - TV R*] *inv rtranclp-relation-invariant*[*OF st*] **by** *blast*
have *R-abs*$^{++}$ *S W*
using *SV VW* **by** *auto*
then show *?case* **using** *UW* **by** *blast*
qed

lemma *rtranclp-relation-rtranclp-relation-abs-compatible*:
fixes *S* :: *'st*
assumes
R: R^{**} (*state S*) *T* **and**
inv: *inv* (*state S*)
shows $\exists$ *U*. *R-abs*** *S U* $\wedge$ *T* $\sim$*m state U*
using *R inv* **by** (*auto simp*: *rtranclp-unfold dest*: *tranclp-relation-tranclp-relation-abs-compatible*)

lemma *no-step-iff*:
inv (*state S*) $\Longrightarrow$ *no-step R* (*state S*) $\longleftrightarrow$ *no-step R-abs S*
using *relation-compatible-state relation-compatible-abs* $cdcl_W$*-mset.state-eq-ref*
by *blast*

lemma *tranclp-relation-compatible-eq-and-inv*:
assumes
inv: *inv* (*state S*) **and**
st: *R-abs*$^{++}$ *S T* **and**
SS': *state S* $\sim$*m state S'* **and**
TU: *state T* $\sim$*m state U*
shows *R-abs*$^{++}$ *S' U* $\wedge$ *inv* (*state U*)
using *st TU*
proof (*induction arbitrary*: *U rule*: *tranclp-induct*)
case (*base T*)
moreover then have *inv* (*state U*)
by (*metis* (*full-types*) $cdcl_W$*-mset-state-eq-eq inv relation-compatible-state relation-invariant*)
ultimately show *?case*
using *relation-compatible-eq*[*of S T S' U*] *SS' inv*
by (*auto simp*: *tranclp.r-into-trancl*)
next
case (*step T T'*) **note** *st* = *this*(*1*) **and** *R* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *TU* = *this*(*4*)
have *R-abs*$^{++}$ *S' T* **and** *invT*: *inv* (*state T*) **using** *IH*[*of T*] **by** *auto*
moreover have *R-abs T U*
using *relation-compatible-eq*[*of T T' T U*] *R TU inv rtranclp-relation-invariant invT* **by** *simp*
moreover have *inv* (*state U*)
using *calculation*(*3*) *invT relation-compatible-state relation-invariant* **by** *blast*
ultimately show *?case* **by** *auto*
qed

lemma
assumes
inv: *inv* (*state S*) **and**
st: *R-abs*$^{++}$ *S T* **and**
SS': *state S* $\sim$*m state S'* **and**
TU: *state T* $\sim$*m state U*
shows
tranclp-relation-compatible-eq: *R-abs*$^{++}$ *S' U* **and**

tranclp-relation-abs-invariant: *inv* (*state* *U*)
using *tranclp-relation-compatible-eq-and-inv*[*OF assms*] **by** *blast+*

lemma *tranclp-abs-tranclp*: *R-abs*$^{++}$ *S* *T* $\Longrightarrow$ *inv* (*state* *S*) $\Longrightarrow$ R^{++} (*state* *S*) (*state* *T*)
apply (*induction rule*: *tranclp-induct*)
apply (*auto simp add*: *relation-compatible-state*)[]
apply *clarsimp*
apply (*erule tranclp.trancl-into-trancl*)
using *relation-compatible-state tranclp-relation-abs-invariant* **by** *blast*

lemma *full1-iff*:
assumes *inv*: *inv* (*state* *S*)
shows *full1* *R* (*state* *S*) (*state* *T*) $\longleftrightarrow$ *full1* *R-abs* *S* *T* (**is** *?R* $\longleftrightarrow$ *?R-abs*)
proof
assume *?R*
then have *st*: R^{++} (*state* *S*) (*state* *T*) **and** *ns*: *no-step* *R* (*state* *T*) **unfolding** *full1-def* **by** *auto*
have *invT*: *inv* (*state* *T*)
using *inv rtranclp-relation-invariant st* **by** *blast*
then have *R-abs*$^{++}$ *S* *T*
using *tranclp-relation-tranclp-relation-abs-compatible*[*OF st*] *inv*
tranclp-relation-compatible-eq[*of S - S T*] $cdcl_W$*-mset.state-eq-sym* **by** *blast*
moreover have *no-step* *R-abs* *T*
using *ns inv no-step-iff invT* **by** *blast*
ultimately show *?R-abs*
unfolding *full1-def* **by** *blast*
next
assume *?R-abs*
then have *st*: *R-abs*$^{++}$ *S* *T* **and** *ns*: *no-step* *R-abs* *T* **unfolding** *full1-def* **by** *auto*
have R^{++} (*state* *S*) (*state* *T*)
using *st tranclp-abs-tranclp inv* **by** *blast*
moreover
have *invT*: *inv* (*state* *T*)
using *inv tranclp-relation-abs-invariant st* **by** *blast*
then have *no-step* *R* (*state* *T*)
using *ns inv no-step-iff* **by** *blast*
ultimately show *?R*
unfolding *full1-def* **by** *blast*
qed

lemma *full1-iff-compatible*:
assumes *inv*: *inv* (*state* *S*) **and** *SS'*: S' $\sim$*m* *state* *S* **and** *TT'*: T' $\sim$*m* *state* *T*
shows *full1* *R* S' T' $\longleftrightarrow$ *full1* *R-abs* *S* *T* (**is** *?R* $\longleftrightarrow$ *?R-abs*)
using *full1-iff assms* **unfolding** $cdcl_W$*-mset-state-eq-eq* **by** *simp*

lemma *full-if-full-abs*:
assumes *inv* (*state* *S*) **and** *full* *R-abs* *S* *T*
shows *full* *R* (*state* *S*) (*state* *T*)
using *assms full1-iff* $cdcl_W$*-mset-state-eq-eq relation-compatible-abs*
unfolding *full-unfold* **by** *blast*

The converse does *not* hold, since we cannot prove that $S = T$ given *state* S = *state* S.

lemma *full-abs-if-full*:
assumes *inv* (*state* *S*) **and** *full* *R* (*state* *S*) (*state* *T*)
shows *full* *R-abs* *S* *T* $\lor$ (*state* *S* $\sim$*m* *state* *T* $\land$ *no-step* *R* (*state* *S*))
using *assms full1-iff relation-compatible-abs* **unfolding** *full-unfold* **by** *auto*

lemma *full-exists-full-abs*:
  assumes *inv*: *inv* (*state S*) **and** *full*: *full R* (*state S*) *T*
  obtains *U* **where** *full R-abs S U* **and** *T* ∼*m state U*
proof −
  consider
    (*0*)   *state S* = *T* **and** *no-step R* (*state S*) |
    (*full1*) *full1 R* (*state S*) *T*
  using *full* **unfolding** *full-unfold* $cdcl_W$*-mset-state-eq-eq* **by** *fast*
  then show *?thesis*
    proof *cases*
      case *0*
      then show *?thesis* **using** *that*[*of S*] **unfolding** *full-def*
        using $cdcl_W$*-mset.state-eq-ref inv relation-compatible-state rtranclp.rtrancl-refl* **by** *blast*
    next
      case *full1*
      then obtain *U* **where**
        $R\text{-}abs^{++}$ *S U* **and** *T* ∼*m state U*
        using *tranclp-relation-tranclp-relation-abs-compatible inv* **unfolding** *full1-def*
        by *blast*
      then show *?thesis*
        using *full1 that*[*of U*] *full1-iff*[*OF inv*] *full1-is-full full-def*
        unfolding $cdcl_W$*-mset-state-eq-eq* **by** *blast*
    qed
qed

lemma *full1-exists-full1-abs*:
  assumes *inv*: *inv* (*state S*) **and** *full1*: *full1 R* (*state S*) *T*
  obtains *U* **where** *full1 R-abs S U* **and** *T* ∼*m state U*
proof −
  obtain *U* **where**
    $R\text{-}abs^{++}$ *S U* **and** *T* ∼*m state U*
    using *tranclp-relation-tranclp-relation-abs-compatible inv full1* **unfolding** *full1-def*
    by *blast*
  then show *?thesis*
    using *full1 that*[*of U*] *full1-iff*[*OF inv*] **unfolding** $cdcl_W$*-mset-state-eq-eq* **by** *blast*
qed

lemma *full1-right-compatible*:
  assumes *inv* (*state S*) **and**
    full1: *full1 R-abs S T* **and** *TV*: *state T* ∼*m state V*
  shows *full1 R-abs S V*
  by (*metis* (*full-types*) *TV assms*(*1*) $cdcl_W$*-mset-state-eq-eq full1 full1-iff*)

lemma *full-right-compatible*:
  assumes *inv*: *inv* (*state S*) **and**
    full-ST: *full R-abs S T* **and** *TU*: *state T* ∼*m state U*
  shows *full R-abs S U* ∨ (*S* = *T* ∧ *no-step R-abs S*)
proof −
  consider
    (*0*) *S* = *T* **and** *no-step R-abs T* |
    (*full1*) *full1 R-abs S T*
    using *full-ST* **unfolding** *full-unfold* **by** *blast*
  then show *?thesis*
    proof *cases*
      case *full1*
      then show *?thesis*

using *full1-right-compatible*[*OF inv*, *of T U*] *TU full-unfold* **by** *blast*
next
case *0*
then show *?thesis* **by** *fast*
qed
qed

end

locale *relation-relation-abs* =
fixes
$R :: 'v\ cdcl_W\text{-}mset \Rightarrow 'v\ cdcl_W\text{-}mset \Rightarrow bool$ **and**
$R\text{-}abs :: 'st \Rightarrow 'st \Rightarrow bool$ **and**
$state :: 'st \Rightarrow 'v\ cdcl_W\text{-}mset$ **and**
$inv :: 'v\ cdcl_W\text{-}mset \Rightarrow bool$
assumes
relation-compatible-state:
$inv\ (state\ S) \Longrightarrow R\ (state\ S)\ (state\ T) \longleftrightarrow R\text{-}abs\ S\ T$ **and**
relation-compatible-abs:
$\bigwedge S\ S'\ T.\ inv\ S \Longrightarrow S \sim m\ state\ S' \Longrightarrow R\ S\ T \Longrightarrow \exists\, U.\ R\text{-}abs\ S'\ U \wedge T \sim m\ state\ U$ **and**
relation-invariant:
$\bigwedge S\ T.\ R\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
begin

lemma *relation-compatible-eq*:
$inv\ (state\ S) \Longrightarrow R\text{-}abs\ S\ T \Longrightarrow state\ S \sim m\ state\ S' \Longrightarrow state\ T \sim m\ state\ T' \Longrightarrow R\text{-}abs\ S'\ T'$
by (*simp add*: $cdcl_W$*-mset-state-eq-eq relation-compatible-state*[*symmetric*])

lemma *relation-right-compatible*:
$inv\ (state\ S) \Longrightarrow R\text{-}abs\ S\ T \Longrightarrow state\ T \sim m\ state\ U \Longrightarrow R\text{-}abs\ S\ U$
by (*simp add*: $cdcl_W$*-mset-state-eq-eq relation-compatible-state*[*symmetric*])

sublocale *relation-implied-relation-abs*
apply *unfold-locales*
using *relation-compatible-eq relation-compatible-state relation-compatible-abs relation-invariant relation-right-compatible* **by** *blast+*

end

7.2.3 The State

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

locale *abs-state*$_W$*-ops* =
raw-clss cls-lit in-cls mset-cls
clss-cls in-clss mset-clss
+
raw-cls mset-ccls
for
— Clause:
$cls\text{-}lit :: 'cls \Rightarrow 'lit \Rightarrow 'v\ literal$ **and**
$in\text{-}cls :: 'lit \Rightarrow 'cls \Rightarrow bool$ **and**
$mset\text{-}cls :: 'cls \Rightarrow 'v\ clause$ **and**

— Multiset of Clauses:
clss-cls :: *'clss* ⇒ *'cls-it* ⇒ *'cls* **and**
in-clss :: *'cls-it* ⇒ *'clss* ⇒ *bool* **and**
mset-clss:: *'clss* ⇒ *'cls multiset* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause*
+
fixes
conc-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-conc-trail :: *'st* ⇒ (*'v*, *'cls-it*) *ann-lit* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
conc-backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conc-conflicting :: *'st* ⇒ *'ccls option* **and**

conc-learned-clss :: *'st* ⇒ *'v clauses* **and**

cons-conc-trail :: (*'v*, *'cls-it*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-conc-trail :: *'st* ⇒ *'st* **and**
add-conc-confl-to-learned-cls :: *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-conc-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
mark-conflicting :: *'cls-it* ⇒ *'st* ⇒ *'st* **and**
reduce-conc-trail-to :: (*'v*, *'v clause*) *ann-lits* ⇒ *'st* ⇒ *'st* **and**
resolve-conflicting :: *'v literal* ⇒ *'cls* ⇒ *'st* ⇒ *'st* **and**

conc-init-state :: *'clss* ⇒ *'st* **and**
restart-state :: *'st* ⇒ *'st*
begin

fun *mmset-of-mlit* :: *'clss* ⇒ (*'v*, *'cls-it*) *ann-lit* ⇒ (*'v*, *'v clause*) *ann-lit*
where
mmset-of-mlit Cs (*Propagated L C*) = *Propagated L* (*mset-cls* (*Cs* ⇓ *C*)) |
mmset-of-mlit - (*Decided L*) = *Decided L*

lemma *lit-of-mmset-of-mlit*[*simp*]:
lit-of (*mmset-of-mlit Cs a*) = *lit-of a*
by (*cases a*) *auto*

lemma *lit-of-mmset-of-mlit-set-lit-of-l*[*simp*]:
lit-of ‘ *mmset-of-mlit Cs* ‘ *set M'* = *lits-of-l M'*
by (*induction M'*) *auto*

lemma *map-mmset-of-mlit-true-annots-true-cls*[*simp*]:
map (*mmset-of-mlit Cs*) *M'* ⊨*as C* ⟷ *M'* ⊨*as C*
by (*simp add*: *true-annots-true-cls lits-of-def*)

definition *clauses-of-clss* **where**
clauses-of-clss N ≡ *image-mset mset-cls* (*mset-clss N*)

definition *conc-clauses* :: *'st* ⇒ *'v clauses* **where**
conc-clauses S ≡ *image-mset mset-cls* (*mset-clss* (*raw-clauses S*))

definition *conc-init-clss* :: *'st* ⇒ *'v literal multiset multiset* **where**
conc-init-clss = (λ*S*. *conc-clauses S* − *conc-learned-clss S*)

abbreviation *conc-conflicting* :: $'st \Rightarrow 'v$ *clause option* **where**
conc-conflicting $\equiv \lambda S.$ *map-option mset-ccls* (*raw-conc-conflicting* S)

definition *state* :: $'st \Rightarrow 'v$ $cdcl_W$*-mset* **where**
state = ($\lambda S.$ (*conc-trail* S, *conc-init-clss* S, *conc-learned-clss* S, *conc-backtrack-lvl* S,
conc-conflicting S))

fun *valid-annotation* :: $'st \Rightarrow ('a, 'cls\text{-}it)$ *ann-lit* $\Rightarrow$ *bool* **where**
valid-annotation S (*Propagated* - E) $\longleftrightarrow E \in\Downarrow$ *raw-clauses* S |
valid-annotation S (*Decided* -) $\longleftrightarrow$ *True*

end

We are using an abstract state to abstract away the detail of the implementation: we do not need to know how the clauses are represented internally, we just need to know that they can be converted to multisets.

Weidenbach state is a five-tuple composed of:

1. the trail is a list of decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause (*'ccls*, standing for conflicting clause) and one for the initial and learned clauses (*'cls*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to *'v CDCL-Abstract-Clause-Representation.clause* is enough (needed for function *hd-raw-conc-trail* below).
There are several axioms to state the independance of the different fields of the state: for example, adding a clause to the learned clauses does not change the trail.
We define the following operations on the elements

- trail: *cons-trail*, *tl-trail*, and *reduce-conc-trail-to*.
- initial set of clauses: a clause can be removed.
- learned clauses: *add-conc-confl-to-learned-cls* moves the conflicting clause to the learned clauses.
- backtrack level: it can be arbitrary set.
- conflicting clause: there is *resolve-conflicting* that does a resolve step, *mark-conflicting* setting a conflict, and *add-conc-confl-to-learned-cls* setting the conflicting clause to *None*.

To ease the representation, we consider the clauses all together, where some of them are learned. This eases representation like arrays where the initial set of clause is at the beginning and avoid having an explicit *op* $\cup$ operator.

locale *abs-state*$_W$ =
abs-state$_W$*-ops*
— functions for clauses:
cls-lit in-cls mset-cls
clss-cls in-clss mset-clss

— functions for the conflicting clause:
mset-ccls

— functions about the state:
— getter:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss
— setter:
cons-conc-trail tl-conc-trail add-conc-confl-to-learned-cls remove-cls update-conc-backtrack-lvl
mark-conflicting reduce-conc-trail-to resolve-conflicting

— Some specific states:
conc-init-state
restart-state
for
— Clause:
cls-lit :: *'cls* ⇒ *'lit* ⇒ *'v literal* **and**
in-cls :: *'lit* ⇒ *'cls* ⇒ *bool* **and**
mset-cls :: *'cls* ⇒ *'v clause* **and**

— Multiset of Clauses:
clss-cls :: *'clss* ⇒ *'cls-it* ⇒ *'cls* **and**
in-clss :: *'cls-it* ⇒ *'clss* ⇒ *bool* **and**
mset-clss:: *'clss* ⇒ *'cls multiset* **and**

— Conflicting clause:
mset-ccls :: *'ccls* ⇒ *'v clause* **and**

conc-trail :: *'st* ⇒ (*'v*, *'v clause*) *ann-lits* **and**
hd-raw-conc-trail :: *'st* ⇒ (*'v*, *'cls-it*) *ann-lit* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
conc-backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conc-conflicting :: *'st* ⇒ *'ccls option* **and**
conc-learned-clss :: *'st* ⇒ *'v clauses* **and**

cons-conc-trail :: (*'v*, *'cls-it*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-conc-trail :: *'st* ⇒ *'st* **and**
add-conc-confl-to-learned-cls :: *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-conc-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
mark-conflicting :: *'cls-it* ⇒ *'st* ⇒ *'st* **and**
reduce-conc-trail-to :: (*'v*, *'v clause*) *ann-lits* ⇒ *'st* ⇒ *'st* **and**
resolve-conflicting :: *'v literal* ⇒ *'cls* ⇒ *'st* ⇒ *'st* **and**

conc-init-state :: *'clss* ⇒ *'st* **and**
restart-state :: *'st* ⇒ *'st* +
assumes
— Definition of *hd-raw-trail*:
hd-raw-conc-trail:
conc-trail st ≠ [] ⟹

mmset-of-mlit (*raw-clauses st*) (*hd-raw-conc-trail st*) = *hd* (*conc-trail st*) **and**

cons-conc-trail:
$\bigwedge S'$. *undefined-lit* (*conc-trail st*) (*lit-of L*) $\Longrightarrow$
state st = (*M*, S') $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
state (*cons-conc-trail L st*) = (*mmset-of-mlit* (*raw-clauses st*) *L* # *M*, S') **and**

tl-conc-trail:
$\bigwedge S'$. *state st* = (*M*, S') $\Longrightarrow$ *state* (*tl-conc-trail st*) = (*tl M*, S') **and**

remove-cls:
$\bigwedge S'$. *state st* = (*M*, *N*, *U*, S') $\Longrightarrow$
state (*remove-cls C st*) =
(*M*, *removeAll-mset* (*mset-cls C*) *N*, *removeAll-mset* (*mset-cls C*) *U*, S') **and**

add-conc-confl-to-learned-cls:
no-dup (*conc-trail st*) $\Longrightarrow$ *state st* = (*M*, *N*, *U*, *k*, *Some F*) $\Longrightarrow$
state (*add-conc-confl-to-learned-cls st*) =
(*M*, *N*, {#*F*#} + *U*, *k*, *None*) **and**

update-conc-backtrack-lvl:
$\bigwedge S'$. *state st* = (*M*, *N*, *U*, *k*, S') $\Longrightarrow$
state (*update-conc-backtrack-lvl* k' *st*) = (*M*, *N*, *U*, k', S') **and**

mark-conflicting:
state st = (*M*, *N*, *U*, *k*, *None*) $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
state (*mark-conflicting E st*) = (*M*, *N*, *U*, *k*, *Some* (*mset-cls* (*raw-clauses st* $\Downarrow$ *E*))) **and**

resolve-conflicting:
state st = (*M*, *N*, *U*, *k*, *Some F*) $\Longrightarrow$ $-L'$ $\in$# *F* $\Longrightarrow$ L' $\in$# *mset-cls D* $\Longrightarrow$
state (*resolve-conflicting* L' *D st*) =
(*M*, *N*, *U*, *k*, *Some* ($cdcl_W$*-mset.resolve-cls* L' *F* (*mset-cls D*))) **and**

conc-init-state:
state (*conc-init-state Ns*) = ([], *clauses-of-clss Ns*, {#}, *0*, *None*) **and**

— Properties about restarting *restart-state*:
conc-trail-restart-state[*simp*]: *conc-trail* (*restart-state S*) = [] **and**
conc-init-clss-restart-state[*simp*]: *conc-init-clss* (*restart-state S*) = *conc-init-clss S* **and**
conc-learned-clss-restart-state[*intro*]:
conc-learned-clss (*restart-state S*) $\subseteq$# *conc-learned-clss S* **and**
conc-backtrack-lvl-restart-state[*simp*]: *conc-backtrack-lvl* (*restart-state S*) = *0* **and**
conc-conflicting-restart-state[*simp*]: *conc-conflicting* (*restart-state S*) = *None* **and**

— Properties about *reduce-conc-trail-to*:
reduce-conc-trail-to[*simp*]:
$\bigwedge S'$. *conc-trail st* = *M2* @ *M1* $\Longrightarrow$ *state st* = (*M*, S') $\Longrightarrow$
state (*reduce-conc-trail-to M1 st*) = (*M1*, S') **and**

learned-clauses:
conc-learned-clss S $\subseteq$# *conc-clauses S*
begin

lemma

conc-init-clss-tl-conc-trail[*simp*]:
conc-init-clss (*tl-conc-trail st*) = *conc-init-clss st* **and**
conc-init-clss-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* $\neq$ *None* $\Longrightarrow$
conc-init-clss (*add-conc-confl-to-learned-cls st*) = *conc-init-clss st* **and**
conc-init-clss-remove-cls[*simp*]:
conc-init-clss (*remove-cls C st*) = *removeAll-mset* (*mset-cls C*) (*conc-init-clss st*) **and**
conc-init-clss-update-conc-backtrack-lvl[*simp*]:
conc-init-clss (*update-conc-backtrack-lvl k st*) = *conc-init-clss st* **and**
conc-init-clss-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-init-clss (*mark-conflicting E st*) = *conc-init-clss st* **and**
conc-init-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ L' $\in\#$ *mset-cls D* $\Longrightarrow$
conc-init-clss (*resolve-conflicting L' D st*) = *conc-init-clss st* **and**
conc-init-clss-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-init-clss (*reduce-conc-trail-to M1 st*) = *conc-init-clss st*
using *tl-conc-trail*[*of st*]
add-conc-confl-to-learned-cls[*of st conc-trail st - - -*]
update-conc-backtrack-lvl[*of st - - - - - k*]
mark-conflicting[*of st - - - - E*]
remove-cls[*of st - - - - C*]
reduce-conc-trail-to[*of st M2 M1*]
resolve-conflicting[*of st - - - - F L' D*]
unfolding *state-def Product-Type.prod.inject*
by (*fastforce*; *fail*)+

lemma
— Properties about the trail *conc-trail*:
conc-trail-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-trail (*cons-conc-trail L st*) = *mmset-of-mlit* (*raw-clauses st*) *L* # *conc-trail st* **and**
conc-trail-tl-conc-trail[*simp*]:
conc-trail (*tl-conc-trail st*) = *tl* (*conc-trail st*) **and**
conc-trail-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* $\neq$ *None* $\Longrightarrow$
conc-trail (*add-conc-confl-to-learned-cls st*) = *conc-trail st* **and**
conc-trail-remove-cls[*simp*]:
conc-trail (*remove-cls C st*) = *conc-trail st* **and**
conc-trail-update-conc-backtrack-lvl[*simp*]:
conc-trail (*update-conc-backtrack-lvl k st*) = *conc-trail st* **and**
conc-trail-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-trail (*mark-conflicting E st*) = *conc-trail st* **and**
conc-trail-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ L' $\in\#$ *mset-cls D* $\Longrightarrow$
conc-trail (*resolve-conflicting L' D st*) = *conc-trail st* **and**

— Properties about the initial clauses *conc-init-clss*:
conc-init-clss-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-init-clss (*cons-conc-trail L st*) = *conc-init-clss st*
and

— Properties about the learned clauses *conc-learned-clss*:

conc-learned-clss-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-learned-clss (*cons-conc-trail L st*) = *conc-learned-clss st* **and**
conc-learned-clss-tl-conc-trail[*simp*]:
conc-learned-clss (*tl-conc-trail st*) = *conc-learned-clss st* **and**
conc-learned-clss-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* = *Some* C' $\Longrightarrow$
conc-learned-clss (*add-conc-confl-to-learned-cls st*) = $\{\#C'\#\}$ + *conc-learned-clss st* **and**
conc-learned-clss-remove-cls[*simp*]:
conc-learned-clss (*remove-cls C st*) = *removeAll-mset* (*mset-cls C*) (*conc-learned-clss st*) **and**
conc-learned-clss-update-conc-backtrack-lvl[*simp*]:
conc-learned-clss (*update-conc-backtrack-lvl k st*) = *conc-learned-clss st* **and**
conc-learned-clss-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ $E \in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-learned-clss (*mark-conflicting E st*) = *conc-learned-clss st* **and**
conc-learned-clss-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L' \in\#$ F $\Longrightarrow$ $L' \in\#$ *mset-cls D* $\Longrightarrow$
conc-learned-clss (*resolve-conflicting* L' *D st*) = *conc-learned-clss st* **and**

— Properties about the backtracking level *conc-backtrack-lvl*:
conc-backtrack-lvl-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-backtrack-lvl (*cons-conc-trail L st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-tl-conc-trail[*simp*]:
conc-backtrack-lvl (*tl-conc-trail st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* $\neq$ *None* $\Longrightarrow$
conc-backtrack-lvl (*add-conc-confl-to-learned-cls st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-remove-cls[*simp*]:
conc-backtrack-lvl (*remove-cls C st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-update-conc-backtrack-lvl[*simp*]:
conc-backtrack-lvl (*update-conc-backtrack-lvl k st*) = *k* **and**
conc-backtrack-lvl-mark-conflicting[*simp*]:
raw-conc-conflicting st = *None* $\Longrightarrow$ $E \in\Downarrow$ *raw-clauses st* $\Longrightarrow$
conc-backtrack-lvl (*mark-conflicting E st*) = *conc-backtrack-lvl st* **and**
conc-backtrack-lvl-clss-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L' \in\#$ F $\Longrightarrow$ $L' \in\#$ *mset-cls D* $\Longrightarrow$
conc-backtrack-lvl (*resolve-conflicting* L' *D st*) = *conc-backtrack-lvl st* **and**

— Properties about the conflicting clause *conc-conflicting*:
conc-conflicting-cons-conc-trail[*simp*]:
undefined-lit (*conc-trail st*) (*lit-of L*) $\Longrightarrow$ *valid-annotation st L* $\Longrightarrow$
conc-conflicting (*cons-conc-trail L st*) = *conc-conflicting st* **and**
conc-conflicting-tl-conc-trail[*simp*]:
conc-conflicting (*tl-conc-trail st*) = *conc-conflicting st* **and**
conc-conflicting-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* = *Some* C' $\Longrightarrow$
conc-conflicting (*add-conc-confl-to-learned-cls st*) = *None*
and
raw-conc-conflicting-add-conc-confl-to-learned-cls[*simp*]:
no-dup (*conc-trail st*) $\Longrightarrow$ *conc-conflicting st* = *Some* C' $\Longrightarrow$
raw-conc-conflicting (*add-conc-confl-to-learned-cls st*) = *None* **and**
conc-conflicting-remove-cls[*simp*]:
conc-conflicting (*remove-cls C st*) = *conc-conflicting st* **and**
conc-conflicting-update-conc-backtrack-lvl[*simp*]:
conc-conflicting (*update-conc-backtrack-lvl k st*) = *conc-conflicting st* **and**

conc-conflicting-clss-clss-resolve-conflicting[*simp*]:
conc-conflicting st = *Some F* $\Longrightarrow$ $-L'$ $\in\#$ *F* $\Longrightarrow$ L' $\in\#$ *mset-cls D* $\Longrightarrow$
conc-conflicting (*resolve-conflicting* L' *D st*) =
Some ($cdcl_W$*-mset.resolve-cls* L' *F* (*mset-cls D*)) **and**

— Properties about the initial state *conc-init-state*:
conc-init-state-conc-trail[*simp*]: *conc-trail* (*conc-init-state Ns*) = [] **and**
conc-init-state-clss[*simp*]: *conc-init-clss* (*conc-init-state Ns*) = *clauses-of-clss Ns* **and**
conc-init-state-conc-learned-clss[*simp*]: *conc-learned-clss* (*conc-init-state Ns*) = {#} **and**
conc-init-state-conc-backtrack-lvl[*simp*]: *conc-backtrack-lvl* (*conc-init-state Ns*) = *0* **and**
conc-init-state-conc-conflicting[*simp*]: *conc-conflicting* (*conc-init-state Ns*) = *None* **and**

— Properties about *reduce-conc-trail-to*:
trail-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$ *conc-trail* (*reduce-conc-trail-to M1 st*) = *M1* **and**
conc-learned-clss-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-learned-clss (*reduce-conc-trail-to M1 st*) = *conc-learned-clss st* **and**
conc-backtrack-lvl-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-backtrack-lvl (*reduce-conc-trail-to M1 st*) = *conc-backtrack-lvl st* **and**
conc-conflicting-reduce-conc-trail-to[*simp*]:
conc-trail st = *M2* @ *M1* $\Longrightarrow$
conc-conflicting (*reduce-conc-trail-to M1 st*) = *conc-conflicting st*
using *cons-conc-trail*[*of st L conc-trail st snd* (*state st*)] *tl-conc-trail*[*of st*]
add-conc-confl-to-learned-cls[*of st conc-trail st - - -*]
update-conc-backtrack-lvl[*of st - - - - - k*]
mark-conflicting[*of st - - - - E*]
remove-cls[*of st - - - - C*]
conc-init-state[*of Ns*]
reduce-conc-trail-to[*of st*]
resolve-conflicting[*of st - - - - F* L' *D*]
unfolding *state-def Product-Type.prod.inject* **by** *auto*

abbreviation *incr-lvl* :: *'st* $\Rightarrow$ *'st* **where**
incr-lvl S $\equiv$ *update-conc-backtrack-lvl* (*conc-backtrack-lvl S* + *1*) *S*

abbreviation *state-eq* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* (**infix** $\sim$ *36*) **where**
S $\sim$ *T* $\equiv$ *state S* $\sim$*m state T*

lemma *state-eq-sym*:
S $\sim$ *T* $\longleftrightarrow$ *T* $\sim$ *S*
using $cdcl_W$*-mset.state-eq-sym* **by** *blast*

lemma *state-eq-trans*:
S $\sim$ *T* $\Longrightarrow$ *T* $\sim$ *U* $\Longrightarrow$ *S* $\sim$ *U*
using $cdcl_W$*-mset.state-eq-trans* **by** *blast*

lemma *conc-clauses-init-learned*: *conc-clauses S* = *conc-init-clss S* + *conc-learned-clss S*
using *learned-clauses*[*of S*] **by** (*auto simp*: *conc-init-clss-def multiset-eq-iff subseteq-mset-def*)

lemma
init-clss-conc-init-clss[*simp*]:
init-clss (*state S*) = *conc-init-clss S* **and**
learned-clss-conc-learned-clss[*simp*]:

learned-clss (*state* S) = *conc-learned-clss* S
by (*auto simp*: *cdcl*$_W$*-mset-state state-def*)

lemma *clauses-conc-clauses*[*simp*]:
cdcl$_W$*-mset.clauses* (*state* S) = *conc-clauses* S
unfolding *conc-clauses-init-learned cdcl*$_W$*-mset.clauses-def* **by** *auto*

lemma
backtrack-lvl-conc-backtrack-lvl[*simp*]:
backtrack-lvl (*state* S) = *conc-backtrack-lvl* S **and**
trail-conc-trail[*simp*]:
trail (*state* S) = *conc-trail* S **and**
conflicting-conc-conflicting[*simp*]:
conflicting (*state* S) = *conc-conflicting* S
by (*auto simp*: *cdcl*$_W$*-mset-state state-def*)

lemma
shows
state-eq-conc-trail: $S \sim T \Longrightarrow$ *conc-trail* S = *conc-trail* T **and**
state-eq-conc-init-clss: $S \sim T \Longrightarrow$ *conc-init-clss* S = *conc-init-clss* T **and**
state-eq-conc-learned-clss: $S \sim T \Longrightarrow$ *conc-learned-clss* S = *conc-learned-clss* T **and**
state-eq-conc-backtrack-lvl: $S \sim T \Longrightarrow$ *conc-backtrack-lvl* S = *conc-backtrack-lvl* T **and**
state-eq-conc-conflicting: $S \sim T \Longrightarrow$ *conc-conflicting* S = *conc-conflicting* T **and**
state-eq-clauses: $S \sim T \Longrightarrow$ *conc-clauses* S = *conc-clauses* T **and**
state-eq-undefined-lit:
$S \sim T \Longrightarrow$ *undefined-lit* (*conc-trail* S) L = *undefined-lit* (*conc-trail* T) L
unfolding *state-def cdcl*$_W$*-mset.state-eq-def conc-clauses-init-learned*
by (*auto simp*: *cdcl*$_W$*-mset-state*)

We combine all simplification rules about *op* $\sim$ in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

lemmas *state-simp* = *state-eq-conc-trail state-eq-conc-init-clss state-eq-conc-learned-clss state-eq-conc-backtrack-lvl state-eq-conc-conflicting state-eq-clauses state-eq-undefined-lit*

lemma *atms-of-ms-conc-learned-clss-restart-state-in-atms-of-ms-conc-learned-clssI*[*intro*]:
$x \in$ *atms-of-mm* (*conc-learned-clss* (*restart-state* S)) $\Longrightarrow x \in$ *atms-of-mm* (*conc-learned-clss* S)
by (*meson atms-of-ms-mono conc-learned-clss-restart-state set-mset-mono subsetCE*)

lemma *clauses-reduce-conc-trail-to*[*simp*]:
conc-trail S = *M2* @ *M1* $\Longrightarrow$ *conc-clauses* (*reduce-conc-trail-to M1* S) = *conc-clauses* S
unfolding *conc-clauses-init-learned* **by** *auto*

lemma *in-get-all-ann-decomposition-clauses-reduce-conc-trail-to*[*simp*]:
(L # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*conc-trail* S)) $\Longrightarrow$
conc-clauses (*reduce-conc-trail-to M1* S) = *conc-clauses* S
unfolding *conc-clauses-init-learned* **by** *auto*

lemma *in-get-all-ann-decomposition-conc-trail-update-conc-trail*[*simp*]:
assumes H: (L # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*conc-trail* S))
shows *conc-trail* (*reduce-conc-trail-to M1* S) = *M1*
using *assms* **by** *auto*

lemma *raw-conc-conflicting-cons-conc-trail*[*simp*]:
assumes *undefined-lit* (*conc-trail* S) (*lit-of* L) **and** *valid-annotation* S L

shows
raw-conc-conflicting (*cons-conc-trail L S*) = *None* $\longleftrightarrow$ *raw-conc-conflicting S* = *None*
using *assms conc-conflicting-cons-conc-trail*[*of S L*] *map-option-is-None* **by** *fastforce+*

lemma *raw-conc-conflicting-update-backtracl-lvl*[*simp*]:
raw-conc-conflicting (*update-conc-backtrack-lvl k S*) = *None* $\longleftrightarrow$ *raw-conc-conflicting S* = *None*
using *map-option-is-None conc-conflicting-update-conc-backtrack-lvl*[*of k S*] **by** *fastforce+*

lemma *conc-conflicting-mark-conflicting*[*simp*]:
raw-conc-conflicting S = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses S* $\Longrightarrow$
conc-conflicting (*mark-conflicting E S*) = *Some* (*mset-cls* (*raw-clauses S* $\Downarrow$ *E*))
using *mark-conflicting* **unfolding** *state-def* **by** *fastforce*

lemma *conflicting-None-iff-raw-conc-conflicting*[*simp*]:
conflicting (*state S*) = *None* $\longleftrightarrow$ *raw-conc-conflicting S* = *None*
unfolding *state-def conflicting-def* **by** *simp*

lemma *trail-state-add-conc-confl-to-learned-cls*:
no-dup (*conc-trail S*) $\Longrightarrow$ *conc-conflicting S* $\neq$ *None* $\Longrightarrow$
trail (*state* (*add-conc-confl-to-learned-cls S*)) = *trail* (*state S*)
unfolding *trail-def state-def* **by** *simp*

lemma *trail-state-update-backtrack-lvl*:
trail (*state* (*update-conc-backtrack-lvl i S*)) = *trail* (*state S*)
unfolding *trail-def state-def* **by** *simp*

lemma *trail-state-update-conflicting*:
raw-conc-conflicting S = *None* $\Longrightarrow$ *E* $\in\Downarrow$ *raw-clauses S* $\Longrightarrow$
trail (*state* (*mark-conflicting E S*)) = *trail* (*state S*)
unfolding *trail-def state-def* **by** *simp*

lemma *tl-trail-state-tl-con-trail*[*simp*]:
tl-trail (*state S*) = *state* (*tl-conc-trail S*)
by (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *add-learned-cls-state-add-conc-confl-to-learned-cls*[*simp*]:
assumes *no-dup* (*conc-trail S*) **and** *raw-conc-conflicting S* = *Some D*
shows *update-conflicting None* (*add-learned-cls* (*mset-ccls D*) (*state S*)) =
state (*add-conc-confl-to-learned-cls S*)
using *assms* **by** (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *state-cons-cons-trail-cons-trail*[*simp*]:
undefined-lit (*trail* (*state S*)) (*lit-of L*) $\Longrightarrow$ *valid-annotation S L* $\Longrightarrow$
cons-trail (*mmset-of-mlit* (*raw-clauses S*) *L*) (*state S*) = *state* (*cons-conc-trail L S*)
by (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *state-cons-cons-trail-cons-trail-propagated*[*simp*]:
undefined-lit (*trail* (*state S*)) *K* $\Longrightarrow$ *C* $\in\Downarrow$ *raw-clauses S* $\Longrightarrow$
cons-trail (*Propagated K* (*mset-cls* (*raw-clauses S* $\Downarrow$ *C*))) (*state S*)
= *state* (*cons-conc-trail* (*Propagated K C*) *S*)
using *state-cons-cons-trail-cons-trail*[*of S Propagated K C*] **by** *simp*

lemma *state-cons-cons-trail-cons-trail-decided*[*simp*]:
undefined-lit (*trail* (*state S*)) *K* $\Longrightarrow$
cons-trail (*Decided K*) (*state S*) = *state* (*cons-conc-trail* (*Decided K*) *S*)
using *state-cons-cons-trail-cons-trail*[*of S Decided K*] **by** *simp*

lemma *state-mark-conflicting-update-conflicting*[*simp*]:
assumes *raw-conc-conflicting S* = *None* **and** $D \in\Downarrow$ *raw-clauses S*
shows
update-conflicting (*Some* (*mset-cls* (*raw-clauses S* $\Downarrow$ *D*))) (*state S*) =
state (*mark-conflicting* (*D*) *S*)
using *assms* **by** (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *update-backtrack-lvl-state*[*simp*]:
update-backtrack-lvl i (*state S*) = *state* (*update-conc-backtrack-lvl i S*)
by (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *update-conflicting-resolve-state-mark-conflicting*[*simp*]:
raw-conc-conflicting S = *Some* $D' \Longrightarrow -L \in\#$ *mset-ccls* $D' \Longrightarrow L \in\#$ *mset-cls* $E' \Longrightarrow$
update-conflicting (*Some* (*remove1-mset* $(- L)$ (*mset-ccls* D') $\#\cup$ *remove1-mset L* (*mset-cls* E')))
(*state* (*tl-conc-trail S*)) =
state (*resolve-conflicting L* E' (*tl-conc-trail S*))
by (*auto simp*: $cdcl_W$*-mset-state state-def simp del*:)

lemma *add-learned-update-backtrack-update-conflicting*[*simp*]:
no-dup (*conc-trail S*) $\Longrightarrow$ *raw-conc-conflicting S* = *Some D* $\Longrightarrow D' \in\Downarrow T \Longrightarrow$
mset-cls $(T \Downarrow D')$ = *mset-ccls D* $\Longrightarrow$
add-learned-cls (*mset-cls* $(T \Downarrow D')$)
(*update-backtrack-lvl i*
(*update-conflicting None*
(*state S*))) =
state (*add-conc-confl-to-learned-cls* (*update-conc-backtrack-lvl i S*))
by (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *state-state*:
$cdcl_W$*-mset.state* (*state S*) = (*trail* (*state S*), *init-clss* (*state S*), *learned-clss* (*state S*),
backtrack-lvl (*state S*), *conflicting* (*state S*))
by (*simp*)

lemma *state-reduce-conc-trail-to-reduce-conc-trail-to*[*simp*]:
assumes [*simp*]: *conc-trail S* = *M2* @ *M1*
shows $cdcl_W$*-mset.reduce-trail-to M1* (*state S*) = *state* (*reduce-conc-trail-to M1 S*) (**is** *?RS* = *?SR*)
proof −
have *1*: *trail ?SR* = *trail ?RS*
apply (*subst state-def*)
apply (*auto simp add*: $cdcl_W$*-mset.trail-reduce-trail-to-drop*)
apply (*auto simp*: *trail-def*)
done

have *2*: *init-clss ?SR* = *init-clss ?RS*
by *simp*

have *3*: *learned-clss ?SR* = *learned-clss ?RS*
by *simp*

have *4*: *backtrack-lvl ?SR* = *backtrack-lvl ?RS*
by *simp*

have *5*: *conflicting ?SR* = *conflicting ?RS*
by *simp*

show *?thesis*
  using *1 2 3 4 5* **apply** −
  apply (*subst* (*asm*) *trail-def*, *subst* (*asm*) *trail-def*)
  apply (*subst* (*asm*) *init-clss-def*, *subst* (*asm*) *init-clss-def*)
  apply (*subst* (*asm*) *learned-clss-def*, *subst* (*asm*) *learned-clss-def*)
  apply (*subst* (*asm*) *backtrack-lvl-def*, *subst* (*asm*) *backtrack-lvl-def*)
  apply (*subst* (*asm*) *conflicting-def*, *subst* (*asm*) *conflicting-def*)
  apply (*cases state* (*reduce-conc-trail-to M1 S*))
  apply (*cases* $cdcl_W$*-mset.reduce-trail-to M1* (*state S*))
  by *simp*
qed

lemma *state-conc-init-state*: *state* (*conc-init-state N*) = *init-state* (*clauses-of-clss N*)
  by (*auto simp*: $cdcl_W$*-mset-state state-def*)

lemma *conc-clauses-add-conc-confl-to-learned-cls*[*simp*]:
  conc-conflicting S = *Some C* $\Longrightarrow$ *no-dup* (*conc-trail S*) $\Longrightarrow$
    conc-clauses (*add-conc-confl-to-learned-cls S*) = {#*C*#} + *conc-clauses S*
  unfolding *conc-clauses-init-learned* **by** (*auto simp*: *ac-simps*)

lemma *raw-conc-conflicting-update-conc-backtrack-lvl*:
  raw-conc-conflicting (*update-conc-backtrack-lvl i S*) = *Some z'* $\Longrightarrow$
    (*raw-conc-conflicting S* ≠ *None* ∧ *conc-conflicting S* = *Some* (*mset-ccls z'*))

  apply *auto*
  apply (*metis not-Some-eq raw-conc-conflicting-update-backtracl-lvl*)
  apply (*metis conc-conflicting-update-conc-backtrack-lvl is-none-code*(*2*) *option.exhaust-sel*
    option.sel raw-conc-conflicting-update-backtracl-lvl the-map-option)
  done

More robust version of theorem *in-mset-clss-exists-preimage*:

lemma *in-clauses-preimage*:
  assumes *b*: *b* ∈# $cdcl_W$*-mset.clauses* (*state C*)
  shows ∃ *b'*. *b'* ∈⇓ *raw-clauses C* ∧ *mset-cls* ((*raw-clauses C*) ⇓ *b'*) = *b*
proof −
  have *b* ∈# *conc-clauses C*
    using *b* **by** *auto*
  then show *?thesis*
    using *in-mset-clss-exists-preimage* **unfolding** *conc-clauses-def* **by** *fastforce*
qed

lemma *state-reduce-conc-trail-to-reduce-conc-trail-to-decomp*[*simp*]:
  assumes (*P* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail S*))
  shows $cdcl_W$*-mset.reduce-trail-to M1* (*state S*) = *state* (*reduce-conc-trail-to M1 S*)
  using *assms* **by** *auto*

end — end of $abs\text{-}state_W$ locale

7.2.4 CDCL Rules

locale $abs\text{-}conflict\text{-}driven\text{-}clause\text{-}learning_W$ =
  $abs\text{-}state_W$
    — functions for clauses:
    cls-lit in-cls mset-cls
    clss-cls in-clss mset-clss

— functions for the conflicting clause:
mset-ccls

— functions about the state:
— getter:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss
— setter:
cons-conc-trail tl-conc-trail add-conc-confl-to-learned-cls remove-cls update-conc-backtrack-lvl
mark-conflicting reduce-conc-trail-to resolve-conflicting

— Some specific states:
conc-init-state
restart-state
for
— Clause:
$cls\text{-}lit :: 'cls \Rightarrow 'lit \Rightarrow 'v\ literal$ **and**
$in\text{-}cls :: 'lit \Rightarrow 'cls \Rightarrow bool$ **and**
$mset\text{-}cls :: 'cls \Rightarrow 'v\ clause$ **and**

— Multiset of Clauses:
$clss\text{-}cls :: 'clss \Rightarrow 'cls\text{-}it \Rightarrow 'cls$ **and**
$in\text{-}clss :: 'cls\text{-}it \Rightarrow 'clss \Rightarrow bool$ **and**
$mset\text{-}clss :: 'clss \Rightarrow 'cls\ multiset$ **and**

— Conflicting clause:
$mset\text{-}ccls :: 'ccls \Rightarrow 'v\ clause$ **and**

$conc\text{-}trail :: 'st \Rightarrow ('v, 'v\ clause)\ ann\text{-}lits$ **and**
$hd\text{-}raw\text{-}conc\text{-}trail :: 'st \Rightarrow ('v, 'cls\text{-}it)\ ann\text{-}lit$ **and**
$raw\text{-}clauses :: 'st \Rightarrow 'clss$ **and**
$conc\text{-}backtrack\text{-}lvl :: 'st \Rightarrow nat$ **and**
$raw\text{-}conc\text{-}conflicting :: 'st \Rightarrow 'ccls\ option$ **and**
$conc\text{-}learned\text{-}clss :: 'st \Rightarrow 'v\ clauses$ **and**

$cons\text{-}conc\text{-}trail :: ('v, 'cls\text{-}it)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$tl\text{-}conc\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}conc\text{-}confl\text{-}to\text{-}learned\text{-}cls :: 'st \Rightarrow 'st$ **and**
$remove\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}conc\text{-}backtrack\text{-}lvl :: nat \Rightarrow 'st \Rightarrow 'st$ **and**
$mark\text{-}conflicting :: 'cls\text{-}it \Rightarrow 'st \Rightarrow 'st$ **and**
$reduce\text{-}conc\text{-}trail\text{-}to :: ('v, 'v\ clause)\ ann\text{-}lits \Rightarrow 'st \Rightarrow 'st$ **and**
$resolve\text{-}conflicting :: 'v\ literal \Rightarrow 'cls \Rightarrow 'st \Rightarrow 'st$ **and**

$conc\text{-}init\text{-}state :: 'clss \Rightarrow 'st$ **and**
$restart\text{-}state :: 'st \Rightarrow 'st$
begin

inductive $propagate\text{-}abs :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
$propagate\text{-}abs\text{-}rule$: $conc\text{-}conflicting\ S = None \Longrightarrow$
$E \in\Downarrow raw\text{-}clauses\ S \Longrightarrow$
$L \in\#\ mset\text{-}cls\ (raw\text{-}clauses\ S \Downarrow E) \Longrightarrow$
$conc\text{-}trail\ S \models as\ CNot\ (mset\text{-}cls\ (raw\text{-}clauses\ S \Downarrow E) - \{\#L\#\}) \Longrightarrow$
$undefined\text{-}lit\ (conc\text{-}trail\ S)\ L \Longrightarrow$
$T \sim cons\text{-}conc\text{-}trail\ (Propagated\ L\ E)\ S \Longrightarrow$

propagate-abs S T

inductive-cases *propagate-absE*: *propagate-abs S T*

lemma *propagate-propagate-abs*:
  $cdcl_W$*-mset.propagate* (*state S*) (*state T*) ⟷ *propagate-abs S T* (**is** *?mset* ⟷ *?abs*)
proof
  assume *?abs*
  then obtain *E L* **where**
    confl: *conc-conflicting S* = *None* **and**
    E: *E* ∈⇓ *raw-clauses S* **and**
    L: *L* ∈# *mset-cls* (*raw-clauses S* ⇓ *E*) **and**
    tr-E: *conc-trail S* ⊨*as CNot* (*mset-cls* (*raw-clauses S* ⇓ *E*) − {#*L*#}) **and**
    undef: *undefined-lit* (*conc-trail S*) *L* **and**
    T: *T* ∼ *cons-conc-trail* (*Propagated L E*) *S*
    by (*auto elim*: *propagate-absE*)

  show *?mset*
    apply (*rule* $cdcl_W$*-mset.propagate-rule*)
        using *confl* **apply** (*auto*; *fail*)[]
       using *E* **apply** (*auto simp*: *conc-clauses-def*; *fail*)[]
      using *L* **apply** (*auto*; *fail*)[]
     using *tr-E* **apply** (*auto*; *fail*)[]
     using *undef* **apply** (*auto*; *fail*)[]
    using *undef T E* **unfolding** $cdcl_W$*-mset-state-eq-eq state-def cons-trail-def* **by** *simp*
next
  assume *?mset*
  then obtain *E L* **where**
    conc-conflicting S = *None* **and**
    E ∈⇓ *raw-clauses S* **and**
    L ∈# *mset-cls* (*raw-clauses S* ⇓ *E*) **and**
    conc-trail S ⊨*as CNot* (*mset-cls* (*raw-clauses S* ⇓ *E*) − {#*L*#}) **and**
    undefined-lit (*conc-trail S*) *L* **and**
    state T ∼*m cons-trail* (*Propagated L* (*mset-cls* (*raw-clauses S* ⇓ *E*))) (*state S*)
    by (*auto elim*!: $cdcl_W$*-mset.propagateE dest*!: *in-clauses-preimage*
      simp: $cdcl_W$*-mset.clauses-def*)
  then show *?abs*
    by (*auto intro*!: *propagate-abs-rule*)
qed

lemma *propagate-compatible-abs*:
  assumes *SS'*: *S* ∼*m state S'* **and** *abs*: $cdcl_W$*-mset.propagate S T*
  obtains *U* **where** *propagate-abs S' U* **and** *T* ∼*m state U*
proof −
  obtain *E L* **where**
    confl: *conflicting S* = *None* **and**
    E: *E* ∈# $cdcl_W$*-mset.clauses S* **and**
    L: *L* ∈# *E* **and**
    tr: *trail S* ⊨*as CNot* (*E* − {#*L*#}) **and**
    undef: *undefined-lit* (*trail S*) *L* **and**
    T: *T* ∼*m cons-trail* (*Propagated L E*) *S*
    using *abs* **by** (*auto elim*!: $cdcl_W$*-mset.propagateE dest*!: *in-clauses-preimage*
      simp: $cdcl_W$*-mset.clauses-def*)
  then obtain *E'* **where**
    E': *E'* ∈⇓ *raw-clauses S'* **and** [*simp*]: *E* = *mset-cls* (*raw-clauses S'* ⇓ *E'*)
    by (*metis SS'* $cdcl_W$*-mset.state-eq-clauses in-clauses-preimage*)

let *?U* = *cons-conc-trail* (*Propagated L E'*) *S'*
have *propagate-abs S' ?U*
apply (*rule propagate-abs-rule*)
using *confl SS'* **apply** *simp*
using *E' SS'* **apply** *simp*
using *L* **apply** *simp*
using *tr SS'* **apply** *simp*
using *undef SS'* **apply** *simp*
using *undef SS'* **by** *simp*
moreover have *T* $\sim$*m state ?U*
using *T SS' undef E'* **by** (*auto simp*: $cdcl_W$*-mset-state-eq-eq*)
ultimately show *thesis* **using** *that* **by** *blast*
qed

interpretation *propagate-abs*: *relation-relation-abs* $cdcl_W$*-mset.propagate propagate-abs state*
λ-. *True*
apply *unfold-locales*
apply (*simp add*: *propagate-propagate-abs*)
using *propagate-compatible-abs* **by** *blast*

inductive *conflict-abs* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
conflict-abs-rule:
conc-conflicting S = *None* $\Longrightarrow$
D $\in\Downarrow$ *raw-clauses S* $\Longrightarrow$
conc-trail S $\models$*as CNot* (*mset-cls* (*raw-clauses S* $\Downarrow$ *D*)) $\Longrightarrow$
T $\sim$ *mark-conflicting D S* $\Longrightarrow$
conflict-abs S T

inductive-cases *conflict-absE*: *conflict-abs S T*

lemma *conflict-conflict-abs*:
$cdcl_W$*-mset.conflict* (*state S*) (*state T*) $\longleftrightarrow$ *conflict-abs S T* (**is** *?mset* $\longleftrightarrow$ *?abs*)
proof
assume *?abs*
then obtain *D* **where**
confl: *conc-conflicting S* = *None* **and**
D: *D* $\in\Downarrow$ *raw-clauses S* **and**
tr-D: *conc-trail S* $\models$*as CNot* (*mset-cls* (*raw-clauses S* $\Downarrow$ *D*)) **and**
T: *T* $\sim$ *mark-conflicting D S*
by (*auto elim*!: *conflict-absE*)
show *?mset*
apply (*rule* $cdcl_W$*-mset.conflict-rule*)
using *confl* **apply** *simp*
using *D* **apply** (*auto simp*: *conc-clauses-def*; *fail*)[]
using *tr-D* **apply** *simp*
using *T confl D* **apply** *auto*
done
next
assume *?mset*
then obtain *D* **where**
confl: *conflicting* (*state S*) = *None* **and**
D: *D* $\in$# $cdcl_W$*-mset.clauses* (*state S*) **and**
tr-D: *trail* (*state S*) $\models$*as CNot D* **and**
T: *state T* $\sim$*m update-conflicting* (*Some D*) (*state S*)
by (*cases state S*) (*auto elim*: $cdcl_W$*-mset.conflictE*)
obtain *D'* **where** *D'*: *D'* $\in\Downarrow$ *raw-clauses S* **and** *DD'*[*simp*]: *D* = *mset-cls* (*raw-clauses S* $\Downarrow$ *D'*)

using D **by** (*auto dest*!: *in-mset-clss-exists-preimage simp*: *conc-clauses-def*)[]
show *?abs*
apply (*rule conflict-abs-rule*)
using *confl* **apply** *simp*
using D' **apply** *simp*
using *tr-D* **apply** *simp*
using T *confl* D' **by** *auto*
qed

lemma *conflict-compatible-abs*:
assumes SS': $S \sim m$ *state* S' **and** *conflict*: $cdcl_W$*-mset.conflict* S T
obtains U **where** *conflict-abs* S' U **and** $T \sim m$ *state* U
proof −
obtain D **where**
confl: *conflicting* S = *None* **and**
D: $D \in\#$ $cdcl_W$*-mset.clauses* S **and**
tr-D: *trail* $S \models as$ *CNot* D **and**
T: $T \sim m$ *update-conflicting* (*Some* D) S
using *conflict* **by** (*auto elim*: $cdcl_W$*-mset.conflictE*)
obtain D' **where** D': $D' \in\Downarrow$ *raw-clauses* S' **and** DD'[*simp*]: D = *mset-cls* (*raw-clauses* $S' \Downarrow D'$)
using D SS' **by** (*auto dest*!: *in-mset-clss-exists-preimage simp*: *conc-clauses-def*)[]
let *?U* = *mark-conflicting* D' S'
have *conflict-abs* S' *?U*
apply (*rule conflict-abs-rule*)
using *confl* SS' **apply** *simp*
using D' SS' **apply** *simp*
using *tr-D* SS' **apply** *simp*
using T **by** *auto*
moreover have $T \sim m$ *state ?U*
using T SS' *confl* D' **by** (*auto simp*: $cdcl_W$*-mset-state-eq-eq*)
ultimately show *thesis* **using** *that*[*of ?U*] **by** *fast*
qed

interpretation *conflict-abs*: *relation-relation-abs* $cdcl_W$*-mset.conflict conflict-abs state*
λ-. *True*
apply *unfold-locales*
apply (*simp add*: *conflict-conflict-abs*)
using *conflict-compatible-abs* **by** *metis*

In the backtrack rule, we assume the existence of an index D' such that the clause is equal to the one use to backtrack.

1. the clause D was added to the state by *add-conc-confl-to-learned-cls*
2. therefore, the index D' exists.

inductive *backtrack-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
backtrack-abs-rule:
conc-conflicting S = *Some* $D \Longrightarrow$
$L \in\#$ $D \Longrightarrow$
(*Decided* K # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail* S)) $\Longrightarrow$
get-level (*conc-trail* S) L = *conc-backtrack-lvl* $S \Longrightarrow$
get-level (*conc-trail* S) L = *get-maximum-level* (*conc-trail* S) $D \Longrightarrow$
get-maximum-level (*conc-trail* S) (D − {#L#}) ≡ $i \Longrightarrow$
get-level (*conc-trail* S) K = i + *1* $\Longrightarrow$
mset-cls

($raw\text{-}clauses$ ($reduce\text{-}conc\text{-}trail\text{-}to$ $M1$ ($add\text{-}conc\text{-}confl\text{-}to\text{-}learned\text{-}cls$
($update\text{-}conc\text{-}backtrack\text{-}lvl$ i S))) $\Downarrow$ D') = D $\Longrightarrow$
D' $\in\Downarrow$ $raw\text{-}clauses$ ($reduce\text{-}conc\text{-}trail\text{-}to$ $M1$ ($add\text{-}conc\text{-}confl\text{-}to\text{-}learned\text{-}cls$
($update\text{-}conc\text{-}backtrack\text{-}lvl$ i S))) $\Longrightarrow$
T $\sim$ $cons\text{-}conc\text{-}trail$ ($Propagated$ L D')
($reduce\text{-}conc\text{-}trail\text{-}to$ $M1$
($add\text{-}conc\text{-}confl\text{-}to\text{-}learned\text{-}cls$
($update\text{-}conc\text{-}backtrack\text{-}lvl$ i S))) $\Longrightarrow$
$backtrack\text{-}abs$ S T

inductive-cases *backtrack-absE*: *backtrack-abs S T*

lemma *backtrack-backtrack-abs*:
assumes *inv*: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*)
shows $cdcl_W$*-mset.backtrack* (*state S*) (*state T*) $\longleftrightarrow$ *backtrack-abs S T* (**is** *?conc* $\longleftrightarrow$ *?abs*)
proof
assume *?abs*
then obtain *D D' L K M1 M2 i* **where**
D: *conc-conflicting S* = *Some D* **and**
L: $L \in\# D$ **and**
decomp: (*Decided K* # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*conc-trail S*)) **and**
lev-L: *get-level* (*conc-trail S*) *L* = *conc-backtrack-lvl S* **and**
lev-Max: *get-level* (*conc-trail S*) *L* = *get-maximum-level* (*conc-trail S*) *D* **and**
i: *get-maximum-level* (*conc-trail S*) (*D* − {#*L*#}) ≡ *i* **and**
lev-K: *get-level* (*conc-trail S*) *K* = *i* + *1* **and**
D': *mset-cls* (*raw-clauses* (*reduce-conc-trail-to M1* (*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S*))) $\Downarrow$ *D'*) = *D* **and**
D'T: *D'* $\in\Downarrow$ *raw-clauses* (*reduce-conc-trail-to M1* (*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S*))) **and**
T: *T* $\sim$ *cons-conc-trail* (*Propagated L D'*)
(*reduce-conc-trail-to M1*
(*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S*)))
by (*auto elim*!: *backtrack-absE*)
have *n-d*: *no-dup* (*trail* (*state S*))
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
by *simp*
have *atm-of L* $\notin$ *atm-of* ' *lits-of-l M1*
apply (*rule* $cdcl_W$*-mset.backtrack-lit-skiped*[*of - state S*])
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *decomp* **apply** *simp*
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *lev-K* **apply** *simp*
done
then have *undef*: *undefined-lit M1 L*
by (*auto simp add*: *defined-lit-map lits-of-def*)
obtain *c* **where** *tr*: *conc-trail S* = *c* @ *M2* @ *Decided K* # *M1*
using *decomp* **by** *auto*
show *?conc*
apply (*rule* $cdcl_W$*-mset.backtrack-rule*)
using *D* **apply** *simp*
using *L* **apply** *simp*

using *decomp* **apply** *simp*
using *lev-L* **apply** *simp*
using *lev-Max* **apply** *simp*
using *i* **apply** *simp*
using *lev-K* **apply** *simp*
using *T undef n-d tr D D′ D′T* **unfolding** *Product-Type.prod.inject* **by** *auto*
next
assume *?conc*
then obtain *L D K M1 M2 i* **where**
confl: *conflicting* (*state S*) = *Some D* **and**
L: *L* ∈# *D* **and**
decomp: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail* (*state S*))) **and**
lev-L: *get-level* (*trail* (*state S*)) *L* = *backtrack-lvl* (*state S*) **and**
lev-max: *get-level* (*trail* (*state S*)) *L* = *get-maximum-level* (*trail* (*state S*)) (*D*) **and**
i: *get-maximum-level* (*trail* (*state S*)) (*D* − {#*L*#}) ≡ *i* **and**
lev-K: *get-level* (*trail* (*state S*)) *K* = *i* + *1* **and**
T: *state T* ∼*m* *cons-trail* (*Propagated L* (*D*))
($cdcl_W$*-mset.reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None* (*state S*)))))
by (*auto elim*: $cdcl_W$*-mset.backtrackE*)
let *?S′* = *reduce-conc-trail-to M1* (*add-conc-confl-to-learned-cls* (*update-conc-backtrack-lvl i S*))
have *n-d*: *no-dup* (*trail* (*state S*))
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
by *simp*
have *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
apply (*rule* $cdcl_W$*-mset.backtrack-lit-skiped*[*of - state S*])
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *decomp* **apply** *simp*
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *lev-L inv* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *lev-K* **apply** *simp*
done
then have *undef*: *undefined-lit M1 L*
by (*auto simp add*: *defined-lit-map lits-of-def*)
have ∃ *z′*. *raw-conc-conflicting* (*update-conc-backtrack-lvl i S*) = *Some z′* ∧ *mset-ccls z′* = *D*
using *confl decomp n-d conc-conflicting-update-conc-backtrack-lvl*[*of i S*]
by (*auto simp del*: *conc-conflicting-update-conc-backtrack-lvl*)
then have *D* ∈# *conc-clauses ?S′*
using *confl decomp n-d* **by** *auto*
with *in-clauses-preimage*[*of D ?S′*] **obtain** *D′* **where**
confl′: *D′* ∈⇓ *raw-clauses ?S′* **and** *D*[*simp*]: *D* = *mset-cls* (*raw-clauses ?S′* ⇓ *D′*)
by (*auto simp*:)

show *?abs*
apply (*rule backtrack-abs-rule*)
using *confl* **apply** (*simp*; *fail*)
using *L* **apply** (*simp*; *fail*)
using *decomp* **apply** (*simp*; *fail*)
using *lev-L* **apply** (*simp*; *fail*)
using *lev-max* **apply** (*simp*; *fail*)
using *i* **apply** (*simp*; *fail*)

using *lev-K* **apply** (*simp*; *fail*)
using *T undef n-d decomp confl′ confl* **by** *auto*
qed

lemma *backtrack-exists-backtrack-abs-step*:
assumes *bt*: $cdcl_W$*-mset.backtrack S T* **and** *inv*: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv S* **and**
SS′: *S* ∼*m state S′*
obtains *U* **where** *backtrack-abs S′ U* **and** *T* ∼*m state U*
proof −
from *bt* **obtain** *L D K M1 M2 i* **where**
confl: *conflicting S* = *Some D* **and**
L: *L* ∈# *D* **and**
decomp: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
lev-L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
lev-max: *get-level* (*trail S*) *L* = *get-maximum-level* (*trail S*) (*D*) **and**
i: *get-maximum-level* (*trail S*) (*D* − {#*L*#}) ≡ *i* **and**
lev-K: *get-level* (*trail S*) *K* = *i* + *1* **and**
T: *T* ∼*m cons-trail* (*Propagated L D*)
($cdcl_W$*-mset.reduce-trail-to M1*
(*add-learned-cls D*
(*update-backtrack-lvl i*
(*update-conflicting None S*))))
by (*auto elim*: $cdcl_W$*-mset.backtrackE*)
obtain *D′* **where**
confl′: *raw-conc-conflicting S′* = *Some D′* **and** *D*[*simp*]: *D* = *mset-ccls D′*
using *confl SS′* **by** *auto*
have *n-d*: *no-dup* (*trail* (*state S′*))
using *lev-L inv SS′* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
by *simp*
have *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
apply (*rule* $cdcl_W$*-mset.backtrack-lit-skiped*[*of* - *state S′*])
using *lev-L inv SS′* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *decomp SS′* **apply** *simp*
using *lev-L inv SS′* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *lev-L inv SS′* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-mset.*$cdcl_W$*-M-level-inv-def*
apply *simp*
using *lev-K SS′* **apply** *simp*
done
then have *undef*: *undefined-lit M1 L*
by (*auto simp add*: *defined-lit-map lits-of-def*)
let *?S* = *reduce-conc-trail-to M1*
(*add-conc-confl-to-learned-cls*
(*update-conc-backtrack-lvl i S′*))
have *D* ∈# *conc-clauses ?S*
using *confl decomp n-d SS′* **by** *auto*
then obtain *D′′* **where**
D′′: *D′′* ∈⇓ *raw-clauses ?S* **and** [*simp*]: *mset-ccls D′* = *mset-cls* (*raw-clauses ?S* ⇓ *D′′*)
using *in-clauses-preimage*[*of D ?S*] **by** *auto*
let *?U* = *cons-conc-trail* (*Propagated L D′′*) *?S*
have *backtrack-abs S′ ?U*
apply (*rule backtrack-abs-rule*)
using *confl′* **apply** (*simp*; *fail*)
using *L* **apply** (*simp*; *fail*)
using *decomp SS′* **apply** (*simp*; *fail*)

using *lev-L SS'* **apply** (*simp*; *fail*)
using *lev-max SS'* **apply** (*simp*; *fail*)
using *i SS'* **apply** (*simp*; *fail*)
using *lev-K SS'* **apply** (*simp*; *fail*)
using *T undef n-d D'' decomp* **by** *auto*
moreover have *T* $\sim$*m state ?U*
using *undef decomp T n-d SS'*[*unfolded* $cdcl_W$*-mset-state-eq-eq*] *confl' D''*
by *auto*
ultimately show *thesis* **using** *that*[*of ?U*] **by** *fast*
qed

interpretation *backtrack-abs*: *relation-relation-abs* $cdcl_W$*-mset.backtrack backtrack-abs state*
$cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv*
apply *unfold-locales*
apply (*simp add*: *backtrack-backtrack-abs*)
using *backtrack-exists-backtrack-abs-step* **apply** *metis*
using $cdcl_W$*-mset.backtrack* $cdcl_W$*-mset.bj* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-inv* **by** *blast*

inductive *decide-abs* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
decide-abs-rule:
conc-conflicting S = None $\Longrightarrow$
undefined-lit (*conc-trail S*) *L* $\Longrightarrow$
atm-of L $\in$ *atms-of-mm* (*conc-init-clss S*) $\Longrightarrow$
T $\sim$ *cons-conc-trail* (*Decided L*) (*incr-lvl S*) $\Longrightarrow$
decide-abs S T

inductive-cases *decide-absE*: *decide-abs S T*

lemma *decide-decide-abs*:
$cdcl_W$*-mset.decide* (*state S*) (*state T*) $\longleftrightarrow$ *decide-abs S T*
by (*auto elim*!: $cdcl_W$*-mset.decideE decide-absE intro*!: $cdcl_W$*-mset.decide-rule decide-abs-rule*)

interpretation *decide-abs*: *relation-relation-abs* $cdcl_W$*-mset.decide decide-abs state*
λ*-. True*
apply *unfold-locales*
apply (*simp add*: *decide-decide-abs*)
apply (*metis* (*full-types*) $cdcl_W$*-mset.decide.cases* $cdcl_W$*-mset-state-eq-eq*
conc-trail-update-conc-backtrack-lvl decide-decide-abs
state-cons-cons-trail-cons-trail-decided trail-conc-trail update-backtrack-lvl-state)
using $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-mset.decide* $cdcl_W$*-mset.other* **by** *blast*

inductive *skip-abs* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
skip-abs-rule:
conc-trail S = Propagated L C' # M $\Longrightarrow$
raw-conc-conflicting S = Some E $\Longrightarrow$
$-L \notin\#$ *mset-ccls E* $\Longrightarrow$
mset-ccls E $\neq$ {#} $\Longrightarrow$
T $\sim$ *tl-conc-trail S* $\Longrightarrow$
skip-abs S T

inductive-cases *skip-absE*: *skip-abs S T*

lemma *skip-skip-abs*:
$cdcl_W$*-mset.skip* (*state S*) (*state T*) $\longleftrightarrow$ *skip-abs S T* (**is** *?conc* $\longleftrightarrow$ *?abs*)
proof
assume *?abs*

then show *?conc*
  by (*auto elim*!: *skip-absE intro*!: *cdcl*$_W$*-mset.skip-rule*)
next
  assume *?conc*
  then obtain L C' E M **where**
    tr: *trail* (*state* S) = *Propagated* L C' # M **and**
    confl: *conflicting* (*state* S) = *Some* E **and**
    L: $-L \notin\#$ E **and**
    E: $E \neq \{\#\}$ **and**
    T: *state* T $\sim m$ *tl-trail* (*state* S)
    by (*auto elim*: *cdcl*$_W$*-mset.skipE*)
  obtain E' **where**
    confl': *raw-conc-conflicting* S = *Some* E' **and** [*simp*]: E = *mset-ccls* E'
    using *confl* **by** *auto*
  show *?abs*
    apply (*rule skip-abs-rule*)
        using *tr* **apply** *simp*
      using *confl'* **apply** *simp*
     using L **apply** *simp*
    using E **apply** *simp*
    using T **by** *simp*
qed

lemma *skip-exists-skip-abs*:
  assumes *skip*: *cdcl*$_W$*-mset.skip* S T **and** *SS'*: S $\sim m$ *state* S'
  obtains U **where** *skip-abs* S' U **and** T $\sim m$ *state* U
proof −
  obtain L C' E M **where**
    tr: *trail* S = *Propagated* L C' # M **and**
    confl: *conflicting* S = *Some* E **and**
    L: $-L \notin\#$ E **and**
    E: $E \neq \{\#\}$ **and**
    T: T $\sim m$ *tl-trail* S
    using *skip* **by** (*auto elim*: *cdcl*$_W$*-mset.skipE*)
  obtain E' **where**
    confl': *raw-conc-conflicting* S' = *Some* E' **and** [*simp*]: E = *mset-ccls* E'
    using *confl SS'* **by** *auto*
  have *skip-abs* S' (*tl-conc-trail* S')
    apply (*rule skip-abs-rule*)
        using *tr SS'* **apply** *simp*
      using *confl' SS'* **apply** *simp*
     using L *SS'* **apply** *simp*
    using E **apply** *simp*
    using T **by** *simp*
  then show *?thesis*
    using *that*[*of tl-conc-trail* S'] T *SS'*[*unfolded cdcl*$_W$*-mset-state-eq-eq*] **by** *auto*
qed

interpretation *skip-abs*: *relation-relation-abs cdcl*$_W$*-mset.skip skip-abs state*
  λ-. *True*
  apply *unfold-locales*
    apply (*simp add*: *skip-skip-abs*)
    using *skip-exists-skip-abs* **apply** *metis*
  using *cdcl*$_W$*-mset.cdcl*$_W$*-all-struct-inv-inv cdcl*$_W$*-mset.skip cdcl*$_W$*-mset.other* **by** *blast*

inductive *resolve-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**

resolve-abs-rule: *conc-trail S* $\neq$ [] $\Longrightarrow$
  hd-raw-conc-trail S = *Propagated L E* $\Longrightarrow$
  L $\in\#$ *mset-cls* (*raw-clauses S* $\Downarrow$ *E*) $\Longrightarrow$
  raw-conc-conflicting S = *Some D'* $\Longrightarrow$
  $-L$ $\in\#$ *mset-ccls D'* $\Longrightarrow$
  get-maximum-level (*conc-trail S*) (*remove1-mset* ($-L$) (*mset-ccls D'*)) = *conc-backtrack-lvl S* $\Longrightarrow$
  T $\sim$ *resolve-conflicting L* (*raw-clauses S* $\Downarrow$ *E*) (*tl-conc-trail S*) $\Longrightarrow$
  resolve-abs S T

inductive-cases *resolve-absE*: *resolve-abs S T*

lemma *resolve-resolve-abs*:
  cdcl$_W$*-mset.resolve* (*state S*) (*state T*) $\longleftrightarrow$ *resolve-abs S T* (**is** *?conc* $\longleftrightarrow$ *?abs*)
proof
  assume *?conc*
  then obtain *L E D* **where**
    tr: *trail* (*state S*) $\neq$ [] **and**
    hd: *cdcl*$_W$*-mset.hd-trail* (*state S*) = *Propagated L E* **and**
    LE: *L* $\in\#$ *E* **and**
    confl: *conflicting* (*state S*) = *Some D* **and**
    LD: $-L$ $\in\#$ *D* **and**
    lvl-max: *get-maximum-level* (*trail* (*state S*)) ((*remove1-mset* ($-L$) *D*)) = *backtrack-lvl* (*state S*) **and**
    T: *state T* $\sim$*m update-conflicting* (*Some* (*cdcl*$_W$*-mset.resolve-cls L D E*)) (*tl-trail* (*state S*))
    by (*auto elim*!: *cdcl*$_W$*-mset.resolveE*)
  obtain *E'* **where**
    hd': *hd-raw-conc-trail S* = *Propagated L E'* **and**
    [*simp*]: *E* = *mset-cls* (*raw-clauses S* $\Downarrow$ *E'*)
    apply (*cases hd-raw-conc-trail S*)
    using *hd-raw-conc-trail*[*of S*] *tr hd* **by** *simp-all*
  obtain *D'* **where**
    confl': *raw-conc-conflicting S* = *Some D'* **and**
    [*simp*]: *D* = *mset-ccls D'*
    using *confl* **by** *auto*
  show *?abs*
    apply (*rule resolve-abs-rule*)
          using *tr* **apply** *simp*
         using *hd'* **apply** *simp*
        using *LE* **apply** *simp*
       using *confl'* **apply** *simp*
      using *LD* **apply** *simp*
     using *lvl-max* **apply** *simp*
    using *T confl' LE LD* **by** *simp*
next
  assume *?abs*
  then show *?conc*
    using *hd-raw-conc-trail*[*of S*] **by** (*auto elim*!: *resolve-absE intro*!: *cdcl*$_W$*-mset.resolve-rule*)
qed

lemma *resolve-exists-resolve-abs*:
  assumes
    res: *cdcl*$_W$*-mset.resolve S T* **and**
    SS': *S* $\sim$*m state S'*
  obtains *U* **where** *resolve-abs S' U* **and** *T* $\sim$*m state U*
proof –
  obtain *L E D* **where**
    tr: *trail S* $\neq$ [] **and**

hd: $cdcl_W$*-mset.hd-trail S* = *Propagated L E* **and**
LE: *L* ∈# *E* **and**
confl: *conflicting S* = *Some D* **and**
LD: −*L* ∈# *D* **and**
lvl-max: *get-maximum-level* (*trail S*) ((*remove1-mset* (−*L*) *D*)) = *backtrack-lvl S* **and**
T: *T* ∼*m update-conflicting* (*Some* ($cdcl_W$*-mset.resolve-cls L D E*)) (*tl-trail S*)
using *res*
by (*auto elim*!: $cdcl_W$*-mset.resolveE*)
obtain *E'* **where**
hd': *hd-raw-conc-trail S'* = *Propagated L E'* **and**
[*simp*]: *E* = *mset-cls* (*raw-clauses S'* ⇓ *E'*)
apply (*cases hd-raw-conc-trail S'*)
using *hd-raw-conc-trail*[*of S'*] *tr hd SS'* **by** *simp-all*
obtain *D'* **where**
confl': *raw-conc-conflicting S'* = *Some D'* **and**
[*simp*]: *D* = *mset-ccls D'*
using *confl SS'* **by** *auto*
let *?U* = *resolve-conflicting L* (*raw-clauses S'* ⇓ *E'*) (*tl-conc-trail S'*)
have *resolve-abs S' ?U*
apply (*rule resolve-abs-rule*)
using *tr SS'* **apply** *simp*
using *hd'* **apply** *simp*
using *LE* **apply** *simp*
using *confl'* **apply** *simp*
using *LD* **apply** *simp*
using *lvl-max SS'* **apply** *simp*
using *T* **by** *simp*
moreover have *T* ∼*m state ?U*
using *T SS' confl LE LD* **unfolding** $cdcl_W$*-mset.state-eq-def* **by** *fastforce*
ultimately show *thesis* **using** *that*[*of ?U*] **by** *fast*
qed

interpretation *resolve-abs*: *relation-relation-abs* $cdcl_W$*-mset.resolve resolve-abs state*
λ-. *True*
apply *unfold-locales*
apply (*simp add*: *resolve-resolve-abs*)
using *resolve-exists-resolve-abs* **apply** *metis*
using $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-mset.resolve* $cdcl_W$*-mset.other* **by** *blast*

inductive *restart* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
restart: *conc-conflicting S* = *None* ⟹
¬*conc-trail S* ⊨*asm conc-clauses S* ⟹
T ∼ *restart-state S* ⟹
restart S T

inductive-cases *restartE*: *restart S T*

We add the condition *C* ∉# *conc-init-clss S*, to maintain consistency even without the strategy.

inductive *forget* :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget-rule:
conc-conflicting S = *None* ⟹
C ∈⇓ *raw-conc-learned-clss S* ⟹
¬(*conc-trail S*) ⊨*asm clauses S* ⟹
mset-cls (*raw-clauses S* ⇓ *C*) ∉ *set* (*get-all-mark-of-propagated* (*conc-trail S*)) ⟹
mset-cls (*raw-clauses S* ⇓ *C*) ∉# *conc-init-clss S* ⟹
T ∼ *remove-cls* (*raw-clauses S* ⇓ *C*) *S* ⟹

forget S T

inductive-cases *forgetE*: *forget S T*

inductive *cdcl$_W$-abs-rf* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
restart: *restart-abs S T* ⟹ *cdcl$_W$-abs-rf S T* |
forget: *forget-abs S T* ⟹ *cdcl$_W$-abs-rf S T*

inductive *cdcl$_W$-abs-bj* :: *'st* ⇒ *'st* ⇒ *bool* **where**
skip: *skip-abs S S'* ⟹ *cdcl$_W$-abs-bj S S'* |
resolve: *resolve-abs S S'* ⟹ *cdcl$_W$-abs-bj S S'* |
backtrack: *backtrack-abs S S'* ⟹ *cdcl$_W$-abs-bj S S'*

inductive-cases *cdcl$_W$-abs-bjE*: *cdcl$_W$-abs-bj S T*

lemma *cdcl$_W$-abs-bj-cdcl$_W$-abs-bj*:
  cdcl$_W$-mset.cdcl$_W$-all-struct-inv (*state S*) ⟹
    cdcl$_W$-mset.cdcl$_W$-bj (*state S*) (*state T*) ⟷ *cdcl$_W$-abs-bj S T*
  by (*auto simp*: *cdcl$_W$-mset.cdcl$_W$-bj.simps cdcl$_W$-abs-bj.simps*
    backtrack-backtrack-abs skip-skip-abs resolve-resolve-abs)

interpretation *cdcl$_W$-abs-bj*: *relation-relation-abs cdcl$_W$-mset.cdcl$_W$-bj cdcl$_W$-abs-bj state*
  cdcl$_W$-mset.cdcl$_W$-all-struct-inv
  apply *unfold-locales*
    apply (*simp add*: *cdcl$_W$-abs-bj-cdcl$_W$-abs-bj*)
   apply (*metis* (*no-types*, *hide-lams*) *backtrack-exists-backtrack-abs-step cdcl$_W$-abs-bj.simps*
     cdcl$_W$-mset.cdcl$_W$-bj.simps resolve-exists-resolve-abs skip-abs.relation-compatible-abs)
  using *cdcl$_W$-mset.bj cdcl$_W$-mset.cdcl$_W$-all-struct-inv-inv cdcl$_W$-mset.other* **by** *blast*

inductive *cdcl$_W$-abs-o* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
decide: *decide-abs S S'* ⟹ *cdcl$_W$-abs-o S S'* |
bj: *cdcl$_W$-abs-bj S S'* ⟹ *cdcl$_W$-abs-o S S'*

inductive *cdcl$_W$-abs* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate: *propagate-abs S S'* ⟹ *cdcl$_W$-abs S S'* |
conflict: *conflict-abs S S'* ⟹ *cdcl$_W$-abs S S'* |
other: *cdcl$_W$-abs-o S S'* ⟹ *cdcl$_W$-abs S S'*|
rf: *cdcl$_W$-abs-rf S S'* ⟹ *cdcl$_W$-abs S S'*

7.2.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have add a strategy and show the inclusion in the multiset version.

inductive *cdcl$_W$-merge-abs-cp* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
conflict': *conflict-abs S T* ⟹ *full cdcl$_W$-abs-bj T U* ⟹ *cdcl$_W$-merge-abs-cp S U* |
propagate': *propagate-abs^{++} S S'* ⟹ *cdcl$_W$-merge-abs-cp S S'*

lemma *cdcl$_W$-merge-cp-cdcl$_W$-abs-merge-cp*:
  assumes
    cp: *cdcl$_W$-merge-abs-cp S T* **and**
    inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*)
  shows *cdcl$_W$-mset.cdcl$_W$-merge-cp* (*state S*) (*state T*)
  using *cp*
proof (*induction rule*: *cdcl$_W$-merge-abs-cp.induct*)
  case (*conflict' T U*) **note** *confl* = *this*(*1*) **and** *bj* = *this*(*2*)

then have *cdcl$_W$-mset.conflict* (*state S*) (*state T*)
by (*auto simp*: *conflict-conflict-abs propagate-propagate-abs cdcl$_W$-abs-bj-cdcl$_W$-abs-bj*)
moreover
have *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state T*)
using *cdcl$_W$-mset.conflict cdcl$_W$-mset.cdcl$_W$-all-struct-inv-inv confl inv*
unfolding *conflict-conflict-abs*[*symmetric*] **by** *blast*
then have *full cdcl$_W$-mset.cdcl$_W$-bj* (*state T*) (*state U*)
using *bj* **by** (*auto simp*: *cdcl$_W$-abs-bj.full-if-full-abs*)
ultimately show *?case* **by** (*auto intro*: *cdcl$_W$-mset.cdcl$_W$-merge-cp.intros*)
next
case (*propagate′ T*)
then show *?case*
by (*auto simp*: *propagate-abs.tranclp-abs-tranclp intro*: *cdcl$_W$-mset.cdcl$_W$-merge-cp.propagate′*)
qed

lemma *cdcl$_W$-merge-cp-abs-exists-cdcl$_W$-merge-cp*:
assumes
cp: *cdcl$_W$-mset.cdcl$_W$-merge-cp* (*state S*) *T* **and**
inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*)
obtains *U* **where** *cdcl$_W$-merge-abs-cp S U* **and** *T* ∼*m state U*
using *cp*
proof (*induction rule*: *cdcl$_W$-mset.cdcl$_W$-merge-cp.induct*)
case (*conflict′ T U*) **note** *confl* = *this*(*1*) **and** *bj* = *this*(*2*) **and** *that* = *this*(*3*)

obtain *V* **where** *SV*: *conflict-abs S V* **and** *TV*: *T* ∼*m state V*
using *conflict-abs.relation-compatible-abs*[*of state S S*] *confl* **by** *blast*
have *inv-V*: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state V*) **and**
inv-T: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv T*
using *TV bj cdcl$_W$-mset.cdcl$_W$-stgy.simps cdcl$_W$-mset.cdcl$_W$-stgy-cdcl$_W$-all-struct-inv*
cdcl$_W$-mset.conflict-is-full1-cdcl$_W$-cp confl inv **unfolding** *cdcl$_W$-mset-state-eq-eq* **by** *blast+*
then obtain *T′* **where** *full cdcl$_W$-abs-bj V T′* **and** *U* ∼*m state T′*
using *TV bj cdcl$_W$-abs-bj.full-exists-full-abs*[*of V U*] **unfolding** *cdcl$_W$-mset-state-eq-eq*
by *blast*
then show *?thesis* **using** *that cdcl$_W$-merge-abs-cp.conflict′*[*of S V T′*] *SV* **by** *fast*
next
case (*propagate′ T*)
then show *?case*
using *cdcl$_W$-merge-abs-cp.propagate′*
propagate-abs.tranclp-relation-tranclp-relation-abs-compatible **by** *blast*
qed

lemma *no-step-cdcl$_W$-merge-cp-no-step-cdcl$_W$-abs-merge-cp*:
assumes
inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*)
shows *no-step cdcl$_W$-merge-abs-cp S* ⟷ *no-step cdcl$_W$-mset.cdcl$_W$-merge-cp* (*state S*)
(**is** *?abs* ⟷ *?conc*)
proof
assume *?abs*
show *?conc*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *T* **where** *cdcl$_W$-mset.cdcl$_W$-merge-cp* (*state S*) *T*
by *blast*
then show *False*
using *cdcl$_W$-merge-cp-abs-exists-cdcl$_W$-merge-cp*[*of S T*] ‹*?abs*› *inv* **by** *auto*
qed

next
  assume *?conc*
  then show *?abs*
    using $cdcl_W$*-merge-cp-*$cdcl_W$*-abs-merge-cp inv* **by** *blast*
qed

lemma $cdcl_W$*-merge-abs-cp-right-compatible*:
  $cdcl_W$*-merge-abs-cp S V* $\Longrightarrow$ $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv (state S)* $\Longrightarrow$
  $V \sim W \Longrightarrow$ $cdcl_W$*-merge-abs-cp S W*
proof (*induction rule*: $cdcl_W$*-merge-abs-cp.induct*)
  case (*conflict' T U*) **note** *confl = this*(*1*) **and** *full = this*(*2*) **and** *inv = this*(*3*) **and** *UW = this*(*4*)
  have *inv-T*: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state T*)
    using $cdcl_W$*-mset.*$cdcl_W$*-stgy.simps* $cdcl_W$*-mset.*$cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*
    $cdcl_W$*-mset.conflict-is-full1-*$cdcl_W$*-cp confl conflict-conflict-abs inv* **by** *blast*
  then have *full* $cdcl_W$*-abs-bj T W* $\vee$ (*T = U* $\wedge$ *no-step* $cdcl_W$*-abs-bj T*)
    using $cdcl_W$*-abs-bj.full-right-compatible*[*OF - full UW*] *full* **by** *blast*
  then consider
      (*full*) *full* $cdcl_W$*-abs-bj T W* |
      (*0*) *T = U* **and** *no-step* $cdcl_W$*-abs-bj T*
    by *blast*
  then show *?case*
    proof *cases*
      case *full*
      then show *?thesis* **using** *confl* **by** (*blast intro*: $cdcl_W$*-merge-abs-cp.intros*)
    next
      case *0*
      then have *conflict-abs S W* **and** *no-step* $cdcl_W$*-abs-bj W*
        using *confl UW conflict-abs.relation-right-compatible* **apply** *blast*
       using *full* **unfolding** *full-def*
       by (*metis* (*mono-tags*, *lifting*) *0*(*1*) *UW inv-T* $cdcl_W$*-abs-bj-*$cdcl_W$*-abs-bj*
         $cdcl_W$*-mset-state-eq-eq*)
      moreover then have *full* $cdcl_W$*-abs-bj W W*
       unfolding *full-def* **by** *auto*
      ultimately show *?thesis* **by** (*blast intro*: $cdcl_W$*-merge-abs-cp.intros*)
    qed
next
  case (*propagate'*)
  then show *?case* **using** *propagate-abs.tranclp-relation-compatible-eq*
    by (*blast intro*: $cdcl_W$*-merge-abs-cp.propagate'*)
qed

interpretation $cdcl_W$*-merge-abs-cp*: *relation-implied-relation-abs*
  $cdcl_W$*-mset.*$cdcl_W$*-merge-cp* $cdcl_W$*-merge-abs-cp state* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv*
  apply *unfold-locales*
     using $cdcl_W$*-merge-cp-*$cdcl_W$*-abs-merge-cp* **apply** *blast*
    using $cdcl_W$*-merge-cp-abs-exists-*$cdcl_W$*-merge-cp* **unfolding** $cdcl_W$*-mset-state-eq-eq* **apply** *blast*
   using $cdcl_W$*-mset.rtranclp-*$cdcl_W$*-all-struct-inv-inv*
   $cdcl_W$*-mset.rtranclp-*$cdcl_W$*-merge-cp-rtranclp-*$cdcl_W$ **apply** *blast*
  using $cdcl_W$*-merge-abs-cp-right-compatible* **unfolding** $cdcl_W$*-mset-state-eq-eq* **by** *blast*

inductive $cdcl_W$*-merge-abs-stgy* **for** *S* :: *'st* **where**
fw-s-cp: *full1* $cdcl_W$*-merge-abs-cp S T* $\Longrightarrow$ $cdcl_W$*-merge-abs-stgy S T* |
fw-s-decide: *decide-abs S T* $\Longrightarrow$ *no-step* $cdcl_W$*-merge-abs-cp S* $\Longrightarrow$ *full* $cdcl_W$*-merge-abs-cp T U*
  $\Longrightarrow$ $cdcl_W$*-merge-abs-stgy S U*

lemma *cdcl$_W$-cp-cdcl$_W$-abs-cp*:
  assumes *stgy*: *cdcl$_W$-merge-abs-stgy S T* **and**
    inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state S*)
  shows *cdcl$_W$-mset.cdcl$_W$-merge-stgy* (*state S*) (*state T*)
  using *stgy*
proof (*induction rule*: *cdcl$_W$-merge-abs-stgy.induct*)
  case (*fw-s-cp T*)
  show *?case*
    apply (*rule cdcl$_W$-mset.cdcl$_W$-merge-stgy.fw-s-cp*)
    using *fw-s-cp inv* **by** (*simp add*: *cdcl$_W$-merge-abs-cp.full1-iff*)
next
  case (*fw-s-decide T U*) **note** *dec* = *this*(*1*) **and** *ns* = *this*(*2*) **and** *full* = *this*(*3*)
  have *dec'*: *cdcl$_W$-mset.decide* (*state S*) (*state T*)
    using *dec decide-decide-abs* **by** *blast*
  then have *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state T*)
    using *inv cdcl$_W$-mset.cdcl$_W$-all-struct-inv-inv*
    by (*blast dest*: *cdcl$_W$-mset.cdcl$_W$.other cdcl$_W$-mset.cdcl$_W$-o.decide*)
  then have *full cdcl$_W$-mset.cdcl$_W$-merge-cp* (*state T*) (*state U*)
    using *full cdcl$_W$-merge-abs-cp.full-if-full-abs* **by** *blast*
  then show *?case*
    using *dec' cdcl$_W$-mset.cdcl$_W$-merge-stgy.fw-s-decide*[*of state S state T state U*] *ns inv*
    by (*simp add*: *no-step-cdcl$_W$-merge-cp-no-step-cdcl$_W$-abs-merge-cp*)
qed

lemma *cdcl$_W$-merge-abs-stgy-exists-cdcl$_W$-merge-stgy*:
  assumes
    inv: *cdcl$_W$-mset.cdcl$_W$-all-struct-inv S* **and**
    SS': *S* $\sim$*m state S'* **and**
    st: *cdcl$_W$-mset.cdcl$_W$-merge-stgy S T*
  shows $\exists$ *U*. *cdcl$_W$-merge-abs-stgy S' U* $\wedge$ *T* $\sim$*m state U*
  using *st*
proof (*induction rule*: *cdcl$_W$-mset.cdcl$_W$-merge-stgy.induct*)
  case (*fw-s-cp T*)
  then show *?case* **using** *cdcl$_W$-merge-abs-cp.full1-exists-full1-abs*[*of S' T*] *inv*
    unfolding *SS'*[*unfolded cdcl$_W$-mset-state-eq-eq*] **by** (*metis cdcl$_W$-merge-abs-stgy.fw-s-cp*)
next
  case (*fw-s-decide T U*) **note** *dec* = *this*(*1*) **and** *n-s* = *this*(*2*) **and** *full* = *this*(*3*)
  have *SS'*: *S* = *state S'*
    using *SS'* **unfolding** *cdcl$_W$-mset-state-eq-eq* **.**
  obtain *T'* **where** *decide-abs S' T'* **and** *TT'*: *T* $\sim$*m state T'*
    using *dec decide-abs.relation-compatible-abs*[*of S S' T*] *SS'* **by** *auto*
  moreover
    have *cdcl$_W$-mset.cdcl$_W$-all-struct-inv* (*state T'*)
      using *SS' calculation*(*1*) *cdcl$_W$-mset.cdcl$_W$.intros*(*3*) *cdcl$_W$-mset.cdcl$_W$-all-struct-inv-inv*
      cdcl$_W$-mset.decide decide-decide-abs inv **by** *blast*
    then obtain *U'* **where** *full cdcl$_W$-merge-abs-cp T' U'* **and** *U* $\sim$*m state U'*
      using *full cdcl$_W$-merge-abs-cp.full-exists-full-abs* **unfolding** *TT'*[*unfolded cdcl$_W$-mset-state-eq-eq*]
      by *blast*
  moreover have *no-step cdcl$_W$-merge-abs-cp S'*
    using *n-s cdcl$_W$-merge-abs-cp.no-step-iff inv* **unfolding** *SS'* **by** *blast*
  ultimately show *?case*
    using *cdcl$_W$-merge-abs-stgy.fw-s-decide*[*of S' T' U'*] **by** *fast*
qed

lemma *cdcl$_W$-merge-abs-stgy-right-compatible*:
  assumes

inv: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state S*) **and**
st: $cdcl_W$*-merge-abs-stgy S T* **and**
TU: $T \sim V$
shows $cdcl_W$*-merge-abs-stgy S V*
using *st TU*
proof (*induction rule*: $cdcl_W$*-merge-abs-stgy.induct*)
case (*fw-s-cp T*)
then show *?thesis*
using $cdcl_W$*-merge-abs-cp.full1-right-compatible* $cdcl_W$*-merge-abs-stgy.fw-s-cp inv* **by** *blast*
next
case (*fw-s-decide T U*) **note** *dec = this*(*1*) **and** *n-s = this*(*2*) **and** *full = this*(*3*) **and** *UV = this*(*4*)
have *inv-T*: $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state T*)
using *dec inv* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-inv*[*of state S state T*]
by (*auto dest*!: $cdcl_W$*-mset.*$cdcl_W$*-o.decide* $cdcl_W$*-mset.*$cdcl_W$*.other*
simp: *decide-decide-abs*[*symmetric*])
then have *full* $cdcl_W$*-merge-abs-cp T V* ∨ (*T = U* ∧ *no-step* $cdcl_W$*-merge-abs-cp T*)
using $cdcl_W$*-merge-abs-cp.full-right-compatible*[*of T U V*] *full UV* **by** *blast*
then consider
(*full*) *full* $cdcl_W$*-merge-abs-cp T V* |
(*0*) *T = U* **and** *no-step* $cdcl_W$*-merge-abs-cp T*
by *blast*
then show *?case*
proof *cases*
case *full*
then show *?thesis*
using *n-s dec* **by** (*blast intro*: $cdcl_W$*-merge-abs-stgy.intros*)
next
case *0* **note** *TU = this*(*1*) **and** *n-s' = this*(*2*)
have *decide-abs S V*
using *TU dec UV decide-abs.relation-abs-right-compatible* **by** *auto*
moreover
have $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state V*)
using *inv-T* **by** (*metis* (*full-types*) *TU* $cdcl_W$*-mset-state-eq-eq fw-s-decide.prems*)
then have *full* $cdcl_W$*-merge-abs-cp V V*
using *n-s' TU UV*[*unfolded* $cdcl_W$*-mset-state-eq-eq*]
unfolding *full-def* **by** (*metis* $cdcl_W$*-merge-abs-cp.no-step-iff rtranclp-unfold*)
ultimately show *?thesis* **using** *n-s* **by** (*blast intro*: $cdcl_W$*-merge-abs-stgy.intros*)
qed
qed

interpretation $cdcl_W$*-merge-abs-stgy*: *relation-implied-relation-abs*
$cdcl_W$*-mset.*$cdcl_W$*-merge-stgy* $cdcl_W$*-merge-abs-stgy state* $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv*
apply *unfold-locales*
using $cdcl_W$*-cp-*$cdcl_W$*-abs-cp* **apply** *blast*
using $cdcl_W$*-merge-abs-stgy-exists-*$cdcl_W$*-merge-stgy* **apply** *blast*
using $cdcl_W$*-mset.*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$ $cdcl_W$*-mset.rtranclp-*$cdcl_W$*-all-struct-inv-inv*
apply *blast*
using $cdcl_W$*-merge-abs-stgy-right-compatible* **by** *blast*

lemma $cdcl_W$*-merge-abs-stgy-final-State-conclusive*:
fixes *T* :: *'st*
assumes
full: *full* $cdcl_W$*-merge-abs-stgy* (*conc-init-state N*) *T* **and**
n-d: *distinct-mset-mset* (*clauses-of-clss N*)
shows (*conc-conflicting T = Some* {#} ∧ *unsatisfiable* (*set-mset* (*clauses-of-clss N*)))
∨ (*conc-conflicting T = None* ∧ *conc-trail T* ⊨*asm clauses-of-clss N*

$\wedge$ *satisfiable* (*set-mset* (*clauses-of-clss* N)))
proof −
have $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv* (*state* (*conc-init-state* N))
using *n-d* **unfolding** $cdcl_W$*-mset.*$cdcl_W$*-all-struct-inv-def* **by** (*auto simp*: *state-conc-init-state*)
then show *?thesis*
using $cdcl_W$*-mset.full-*$cdcl_W$*-merge-stgy-final-state-conclusive′*[*of clauses-of-clss N state T*]
$cdcl_W$*-merge-abs-stgy.full-if-full-abs*[*of conc-init-state N T*] *full*
by (*auto simp*: *state-conc-init-state n-d*)
qed

end

end

7.3 2-Watched-Literal

theory *CDCL-Two-Watched-Literals*
imports *CDCL-W-Abstract-State*
begin

First we define here the core of the two-watched literal data structure:

1. A clause is composed of (at most) two watched literals.
2. It is sufficient to find the candidates for propagation and conflict from the clauses such that the new literal is watched.

While this it the principle behind the two-watched literals, an implementation have to remember the candidates that have been found so far while updating the data structure.

We will directly on the two-watched literals data structure with lists: it could be also seen as a state over some abstract clause representation we would later refine as lists. However, as we need a way to select element from a clause, working on lists is better.

7.3.1 Essence of 2-WL

Data structure and Access Functions

Only the 2-watched literals have to be verified here: the backtrack level and the trail that appear in the state are not related to the 2-watched algoritm.

datatype *′v twl-clause* =
TWL-Clause (*watched*: *′v literal list*) (*unwatched*: *′v literal list*)

datatype *′v twl-state* =
TWL-State (*raw-trail*: (*′v*, *′v twl-clause*) *ann-lits*)
(*raw-init-clss*: *′v twl-clause list*)
(*raw-learned-clss*: *′v twl-clause list*) (*backtrack-lvl*: *nat*)
(*raw-conflicting*: *′v literal list option*)

fun *mmset-of-mlit* :: (*′v*, *′v twl-clause*) *ann-lit* ⇒ (*′v*, *′v clause*) *ann-lit*
where
mmset-of-mlit (*Propagated L C*) = *Propagated L* (*mset* (*watched C* @ *unwatched C*)) |
mmset-of-mlit (*Decided L*) = *Decided L*

lemma *lit-of-mmset-of-mlit*[*simp*]: *lit-of* (*mmset-of-mlit x*) = *lit-of x*
  by (*cases x*) *auto*

lemma *lits-of-mmset-of-mlit*[*simp*]: *lits-of* (*mmset-of-mlit* ‘ *S*) = *lits-of S*
  by (*auto simp*: *lits-of-def image-image*)

abbreviation *trail* **where**
trail S ≡ *map mmset-of-mlit* (*raw-trail S*)

abbreviation *clauses-of-l* **where**
  clauses-of-l ≡ λ*L*. *mset* (*map mset L*)

definition *raw-clause* :: ′*v twl-clause* ⇒ ′*v literal list* **where**
  raw-clause C ≡ *watched C* @ *unwatched C*

definition *clause* :: ′*v twl-clause* ⇒ ′*v clause* **where**
  clause C ≡ *mset* (*raw-clause C*)

lemma *clause-def-lambda*:
  clause = (λ*C*. *mset* (*raw-clause C*))
  by (*auto simp*: *clause-def*)

abbreviation *raw-clss-l* :: ′*a twl-clause list* ⇒ ′*a clauses* **where**
  raw-clss-l C ≡ *mset* (*map clause C*)

abbreviation *raw-clauses* :: ′*v twl-state* ⇒ ′*v twl-clause list* **where**
  raw-clauses S ≡ *raw-init-clss S* @ *raw-learned-clss S*

interpretation *raw-cls clause* **.**

lemma *mset-map-clause-remove1-cond*:
  raw-clss-l (*remove1-cond* (λ*D*. *clause D* = *clause a*) *Cs*) = *remove1-mset* (*clause a*) (*raw-clss-l Cs*)
   apply (*induction Cs*)
     apply *simp*
   by (*auto simp*: *ac-simps remove1-mset-single-add raw-clause-def clause-def*)
term *raw-clss*
interpretation *raw-clss*
  λ- *L*. *L*
  λ*L C*. *L* ∈ *set* (*raw-clause C*)
  clause

  λ- *C*. *C*
  λ*C Cs*. *C* ∈ *set Cs*
  mset
  apply (*unfold-locales*)
  using *mset-map-clause-remove1-cond* **by** (*auto simp*: *hd-map comp-def map-tl ac-simps raw-clause-def*
    union-mset-list mset-map-mset-remove1-cond ex-mset clause-def-lambda)

lemma *ex-mset-unwatched-watched*:
  ∃ *a*. *mset* (*unwatched a*) + *mset* (*watched a*) = *E*
proof −
  obtain *e* **where** *mset e* = *E*
    using *ex-mset* **by** *blast*
  then have *mset* (*unwatched* (*TWL-Clause* [] *e*)) + *mset* (*watched* (*TWL-Clause* [] *e*)) = *E*

by *auto*
then show *?thesis* **by** *fast*
qed

abbreviation *conc-learned-clss* **where**
conc-learned-clss ≡ λ*S*. *mset* (*map clause* (*raw-learned-clss S*))

interpretation *twl*: *abs-state*$_W$*-ops*
λ- *L*. *L*
λ*L C*. *L* ∈ *set* (*raw-clause C*)
clause

λ- *C*. *C*
λ*C Cs*. *C* ∈ *set Cs*
mset

mset
trail λ*S*. *hd* (*raw-trail S*)
(λ*S*. *raw-init-clss S* @ *raw-learned-clss S*) *backtrack-lvl raw-conflicting*
conc-learned-clss
rewrites
twl.mmset-of-mlit S = *mmset-of-mlit*
proof *goal-cases*
case *1*
show *H*: *?case*
apply *unfold-locales*
done

case *2*
show *?case*
apply (*rule ext*)
apply (*rename-tac x*)
apply (*case-tac x*)
apply (*simp-all add*: *abs-state*$_W$*-ops.mmset-of-mlit.simps*[*OF H*] *raw-clause-def clause-def*)
done
qed

definition
candidates-propagate :: ′*v twl-state* ⇒ (′*v literal* × ′*v twl-clause*) *set*
where
candidates-propagate S =
{(*L*, *C*) | *L C*.
C ∈ *set* (*raw-clauses S*) ∧
set (*watched C*) − (*uminus* ‘ *lits-of-l* (*trail S*)) = {*L*} ∧
undefined-lit (*raw-trail S*) *L*}

definition *candidates-conflict* :: ′*v twl-state* ⇒ ′*v twl-clause set* **where**
candidates-conflict S =
{*C*. *C* ∈ *set* (*raw-clauses S*) ∧
set (*watched C*) ⊆ *uminus* ‘ *lits-of-l* (*raw-trail S*)}

primrec (*nonexhaustive*) *index* :: ′*a list* ⇒′*a* ⇒ *nat* **where**
index (*a* # *l*) *c* = (*if a* = *c then 0 else 1*+*index l c*)

lemma *index-nth*:
a ∈ *set l* ⟹ *l* ! (*index l a*) = *a*

by (*induction l*) *auto*

Invariants

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

primrec *struct-wf-twl-cls* :: *'v twl-clause* ⇒ *bool* **where**
struct-wf-twl-cls (*TWL-Clause W UW*) ⟷
  distinct W ∧ *length W* ≤ *2* ∧ (*length W* < *2* ⟶ *set UW* ⊆ *set W*)

We need the following property about updates: if there is a literal L with $-L$ in the trail, and L is not watched, then it stays unwatched; i.e., while updating with *rewatch*, L does not get swapped with a watched literal L' such that $-L'$ is in the trail. This corresponds to the laziness of the data structure.

Remark that M is a trail: literals at the end were the first to be added to the trail.

primrec *watched-only-lazy-updates* :: (*'v*, *'mark*) *ann-lits* ⇒
 'v twl-clause ⇒ *bool*
 where
watched-only-lazy-updates M (*TWL-Clause W UW*) ⟷
 (∀ *L'*∈ *set W*. ∀ *L*∈ *set UW*.
  −*L'* ∈ *lits-of-l M* ⟶ −*L* ∈ *lits-of-l M* ⟶ *L* ∉ *set W* ⟶
   index (*map lit-of M*) (−*L'*) ≤ *index* (*map lit-of M*) (−*L*))

If the negation of a watched literal is included in the trail, then the negation of every unwatched literals is also included in the trail. Otherwise, the data-structure has to be updated.

primrec *watched-wf-twl-cls* :: (*'a*, *'b*) *ann-lits* ⇒ *'a twl-clause* ⇒
 bool **where**
watched-wf-twl-cls M (*TWL-Clause W UW*) ⟷
 (∀ *L* ∈ *set W*. −*L* ∈ *lits-of-l M* ⟶ (∀ *L'* ∈ *set UW*. *L'* ∉ *set W* ⟶ −*L'* ∈ *lits-of-l M*))

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-twl-cls* :: (*'v*, *'mark*) *ann-lits* ⇒ *'v twl-clause* ⇒ *bool* **where**
 wf-twl-cls M (*TWL-Clause W UW*) ⟷
  struct-wf-twl-cls (*TWL-Clause W UW*) ∧ *watched-wf-twl-cls M* (*TWL-Clause W UW*) ∧
  watched-only-lazy-updates M (*TWL-Clause W UW*)

lemma *wf-twl-cls-annotation-independant*:
 assumes *M*: *map lit-of M* = *map lit-of M'*
 shows *wf-twl-cls M* (*TWL-Clause W UW*) ⟷ *wf-twl-cls M'* (*TWL-Clause W UW*)
proof −
 have *lits-of-l M* = *lits-of-l M'*
  using *arg-cong*[*OF M*, *of set*] **by** (*simp add*: *lits-of-def*)
 then show *?thesis*
  by (*simp add*: *lits-of-def M*)
qed

lemma *wf-twl-cls-wf-twl-cls-tl*:
 assumes *wf*: *wf-twl-cls M C* **and** *n-d*: *no-dup M*
 shows *wf-twl-cls* (*tl M*) *C*
proof (*cases M*)
 case *Nil*
 then show *?thesis* **using** *wf*

by (*cases C*) (*simp add*: *wf-twl-cls.simps*[*of tl* -])
next
case (*Cons l M'*) **note** $M = this(1)$
obtain *W UW* **where** *C*: $C = TWL\text{-}Clause\ W\ UW$
by (*cases C*)
{ fix *L L'*
assume
LW: $L \in set\ W$ **and**
LM: $-\ L \in lits\text{-}of\text{-}l\ M'$ **and**
L'UW: $L' \in set\ UW$ **and**
$L' \notin set\ W$
then have
L'M: $-\ L' \in lits\text{-}of\text{-}l\ M$
using *wf* **by** (*auto simp*: *C M*)
have *watched-only-lazy-updates M C*
using *wf* **by** (*auto simp*: *C*)
then have
index (*map lit-of M*) $(-L) \leq$ *index* (*map lit-of M*) $(-L')$
using *LM L'M L'UW LW* ‹$L' \notin set\ W$› *C M* **unfolding** *lits-of-def*
by (*fastforce simp*: *lits-of-def*)
then have $-\ L' \in lits\text{-}of\text{-}l\ M'$
using ‹$L' \notin set\ W$› *LW L'M* **by** (*auto simp*: *C M split*: *if-split-asm*)
}
moreover
{
fix *L' L*
assume
$L' \in set\ W$ **and**
$L \in set\ UW$ **and**
L'M: $-\ L' \in lits\text{-}of\text{-}l\ M'$ **and**
$-\ L \in lits\text{-}of\text{-}l\ M'$ **and**
$L \notin set\ W$
moreover
have $lit\text{-}of\ l \neq -\ L'$
using *n-d* **unfolding** *M*
by (*metis* (*no-types*) *L'M M Decided-Propagated-in-iff-in-lits-of-l defined-lit-map*
distinct.simps(*2*) *list.simps*(*9*) *set-map*)
moreover have *watched-only-lazy-updates M C*
using *wf* **by** (*auto simp*: *C*)
ultimately have *index* (*map lit-of M'*) $(-\ L') \leq$ *index* (*map lit-of M'*) $(-\ L)$
by (*fastforce simp*: *M C split*: *if-split-asm*)
}
moreover have *distinct W* **and** $length\ W \leq 2$ **and** ($length\ W < 2 \longrightarrow set\ UW \subseteq set\ W$)
using *wf* **by** (*auto simp*: *C M*)
ultimately show *?thesis* **by** (*auto simp add*: *M C*)
qed

lemma *wf-twl-cls-append*:
assumes
n-d: *no-dup* (*M'* @ *M*) **and**
wf: *wf-twl-cls* (*M'* @ *M*) *C*
shows *wf-twl-cls M C*
using *wf n-d* **apply** (*induction M'*)
apply *simp*
using *wf-twl-cls-wf-twl-cls-tl* **by** *fastforce*

definition *wf-twl-state* :: *'v twl-state* ⇒ *bool* **where**
wf-twl-state S ⟷
(∀ *C* ∈ *set* (*raw-clauses S*). *wf-twl-cls* (*raw-trail S*) *C*) ∧ *no-dup* (*raw-trail S*)

lemma *wf-candidates-propagate-sound*:
assumes *wf*: *wf-twl-state S* **and**
cand: (*L*, *C*) ∈ *candidates-propagate S*
shows *raw-trail S* ⊨*as CNot* (*mset* (*removeAll L* (*raw-clause C*))) ∧ *undefined-lit* (*raw-trail S*) *L*
(**is** *?Not* ∧ *?undef*)
proof
def *M* ≡ *raw-trail S*
def *N* ≡ *raw-init-clss S*
def *U* ≡ *raw-learned-clss S*

note *MNU-defs* [*simp*] = *M-def N-def U-def*

have *cw*:
C ∈ *set* (*N* @ *U*)
set (*watched C*) − *uminus* ‘ *lits-of-l M* = {*L*}
undefined-lit M L
using *cand* **unfolding** *candidates-propagate-def MNU-defs* **by** *auto*

obtain *W UW* **where** *cw-eq*: *C* = *TWL-Clause W UW*
by (*cases C*)

have *l-w*: *L* ∈ *set W*
using *cw*(*2*) *cw-eq* **by** *auto*

have *wf-c*: *wf-twl-cls M C*
using *wf cw*(*1*) **unfolding** *wf-twl-state-def* **by** *simp*

have *w-nw*:
distinct W
length W < *2* ⟹ *set UW* ⊆ *set W*
⋀*L L'*. *L* ∈ *set W* ⟹ −*L* ∈ *lits-of-l M* ⟹ *L'* ∈ *set UW* ⟹ *L'* ∉ *set W* ⟹ −*L'* ∈ *lits-of-l M*
using *wf-c* **unfolding** *cw-eq* **by** (*auto simp*: *image-image*)

have ∀ *L'* ∈ *set* (*raw-clause C*) − {*L*}. −*L'* ∈ *lits-of-l M*
proof (*cases length W* < *2*)
case *True*
moreover have *size W* ≠ *0*
using *cw*(*2*) *cw-eq* **by** *auto*
ultimately have *size W* = *1*
by *linarith*
then have *w*: *W* = [*L*]
using *l-w* **by** (*auto simp*: *length-list-Suc-0*)
from *True* **have** *set UW* ⊆ *set W*
using *w-nw*(*2*) **by** *blast*
then show *?thesis*
using *w cw*(*1*) *cw-eq* **by** (*auto simp*: *raw-clause-def*)
next
case *sz2*: *False*
show *?thesis*
proof
fix *L'*
assume *l'*: *L'* ∈ *set* (*raw-clause C*) − {*L*}

```
      have ex-la: ∃ La. La ≠ L ∧ La ∈ set W
      proof (cases W)
        case w: Nil
        then show ?thesis
          using l-w by auto
      next
        case lb: (Cons Lb W′)
        show ?thesis
        proof (cases W′)
          case Nil
          then show ?thesis
            using lb sz2 by simp
        next
          case lc: (Cons Lc W′′)
          then show ?thesis
            by (metis distinct-length-2-or-more lb list.set-intros(1) list.set-intros(2) w-nw(1))
        qed
      qed
      then obtain La where la: La ≠ L La ∈ set W
        by blast
      then have La ∈ uminus ‘ lits-of-l M
        using cw(2)[unfolded cw-eq, simplified, folded M-def] ‹La ∈ set W› ‹La ≠ L› by auto
      then have nla: −La ∈ lits-of-l M
        by (auto simp: image-image)
      then show −L′ ∈ lits-of-l M

      proof −
        have f1: L′ ∈ set (raw-clause C)
          using l′ by blast
        have f2: L′ ∉ {L}
          using l′ by fastforce
        have ⋀l L. − (l::′a literal) ∈ L ∨ l ∉ uminus ‘ L
          by force
        then show ?thesis
          using cw(1) cw-eq w-nw(3) raw-clause-def by (metis DiffI Un-iff cw(2) f1 f2 la(2) nla
            set-append twl-clause.sel(1) twl-clause.sel(2))
      qed
    qed
  qed
  then show ?Not
    unfolding true-annots-def by (auto simp: image-image Ball-def CNot-def)

  show ?undef
    using cw(3) unfolding M-def by blast
qed

lemma wf-candidates-propagate-complete:
  assumes wf: wf-twl-state S and
    c-mem: C ∈ set (raw-clauses S) and
    l-mem: L ∈ set (raw-clause C) and
    unsat: trail S ⊨as CNot (mset-set (set (raw-clause C) − {L})) and
    undef: undefined-lit (raw-trail S) L
  shows (L, C) ∈ candidates-propagate S
proof −
  def M ≡ raw-trail S
  def N ≡ raw-init-clss S
```

def $U \equiv$ *raw-learned-clss S*

note *MNU-defs* [*simp*] = *M-def N-def U-def*

obtain *W UW* **where** *cw-eq*: C = *TWL-Clause W UW*
  by (*cases C*, *blast*)

have *wf-c*: *wf-twl-cls M C*
  using *wf c-mem* **unfolding** *wf-twl-state-def* **by** *simp*

have *w-nw*:
  distinct W
  length $W < 2 \Longrightarrow$ *set* $UW \subseteq$ *set* W
  $\bigwedge L\ L'.\ L \in$ *set* $W \Longrightarrow -L \in$ *lits-of-l* $M \Longrightarrow L' \in$ *set* $UW \Longrightarrow L' \notin$ *set* $W \Longrightarrow -L' \in$ *lits-of-l* M
 using *wf-c* **unfolding** *cw-eq* **by** (*auto simp*: *image-image*)

have *unit-set*: *set* $W -$ (*uminus* ‘ *lits-of-l M*) = $\{L\}$ (**is** *?W* = *?L*)
proof
  show *?W* $\subseteq \{L\}$
  proof
    fix L'
    assume l': $L' \in$ *?W*
    then have l'*-mem-w*: $L' \in$ *set W*
      by (*simp add*: *in-diffD*)
    have $L' \notin$ *uminus* ‘ *lits-of-l M*
      using l' **by** *blast*
    then have $\neg\ M \models a\ \{\#-L'\#\}$
      by (*auto simp*: *lits-of-def uminus-lit-swap image-image*)
    moreover have $L' \in$ *set* (*raw-clause C*)
      using *c-mem cw-eq* l'*-mem-w* **by** (*auto simp*: *raw-clause-def*)
    ultimately have $L' = L$
      using *unsat*[*unfolded CNot-def true-annots-def*, *simplified*]
      unfolding *M-def* **by** *fastforce*
    then show $L' \in \{L\}$
      by *simp*
  qed
next
  show $\{L\} \subseteq$ *?W*
  proof *clarify*
    have $L \in$ *set W*
    proof (*cases W*)
      case *Nil*
      then show *?thesis*
        using *w-nw*(*2*) *cw-eq l-mem* **by** (*auto simp*: *raw-clause-def*)
    next
      case (*Cons La W'*)
      then show *?thesis*
      proof (*cases La* = *L*)
        case *True*
        then show *?thesis*
          using *Cons* **by** *simp*
      next
        case *False*
        have $-La \in$ *lits-of-l M*
          using *False Cons cw-eq unsat*[*unfolded CNot-def true-annots-def*, *simplified*]
          by (*fastforce simp*: *raw-clause-def*)

then show *?thesis*
using *Cons cw-eq l-mem undef w-nw*(*3*)
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l raw-clause-def*)
qed
qed
moreover have $L \notin\#$ *mset-set* (*uminus* ‘ *lits-of-l M*)
using *undef* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l image-image*)
ultimately show $L \in$ *?W*
by *simp*
qed
qed

show *?thesis*
unfolding *candidates-propagate-def* **using** *unit-set undef c-mem* **unfolding** *cw-eq M-def*
by (*auto simp*: *image-image cw-eq intro*!: *exI*[*of - C*])
qed

lemma *wf-candidates-conflict-sound*:
assumes *wf*: *wf-twl-state S* **and**
cand: $C \in$ *candidates-conflict S*
shows *trail S* $\models$*as CNot* (*clause C*) $\wedge$ $C \in$ *set* (*raw-clauses S*)
proof
def $M \equiv$ *raw-trail S*
def $N \equiv$ *raw-init-clss S*
def $U \equiv$ *raw-learned-clss S*

note *MNU-defs* [*simp*] = *M-def N-def U-def*

have *cw*:
$C \in$ *set* (N @ U)
set (*watched C*) $\subseteq$ *uminus* ‘ *lits-of-l* (*trail S*)
using *cand*[*unfolded candidates-conflict-def*, *simplified*] **by** *auto*

obtain *W UW* **where** *cw-eq*: C = *TWL-Clause W UW*
by (*cases C*, *blast*)

have *wf-c*: *wf-twl-cls M C*
using *wf cw*(*1*) **unfolding** *wf-twl-state-def* **by** *simp*

have *w-nw*:
distinct W
length W < *2* $\Longrightarrow$ *set UW* $\subseteq$ *set W*
$\bigwedge L\ L'.\ L \in set\ W \Longrightarrow -L \in lits\text{-}of\text{-}l\ M \Longrightarrow L' \in set\ UW \Longrightarrow L' \notin set\ W \Longrightarrow -L' \in lits\text{-}of\text{-}l\ M$
using *wf-c* **unfolding** *cw-eq* **by** (*auto simp*: *image-image*)

have $\forall L \in$ *set* (*raw-clause C*). $-L \in$ *lits-of-l M*
proof (*cases W*)
case *Nil*
then have *raw-clause C* = []
using *cw*(*1*) *cw-eq w-nw*(*2*) **by** (*auto simp*: *raw-clause-def*)
then show *?thesis*
by *simp*
next
case (*Cons La W'*) **note** *W'* = *this*(*1*)
show *?thesis*
proof

fix L
assume l: $L \in set$ $(raw\text{-}clause\ C)$
show $-L \in lits\text{-}of\text{-}l\ M$
proof $(cases\ L \in set\ W)$
case *True*
then show *?thesis*
using $cw(2)$ *cw-eq* **by** *fastforce*
next
case *False*
then show *?thesis*
using W' $cw(2)$ *cw-eq* l $w\text{-}nw(3)$ **unfolding** *M-def raw-clause-def*
by (*metis* (*no-types*, *lifting*) *UnE imageE list.set-intros*(*1*)
lits-of-mmset-of-mlit rev-subsetD set-append set-map twl-clause.sel(*1*)
twl-clause.sel(*2*) *uminus-of-uminus-id*)
qed
qed
qed
then show *trail* $S \models as$ *CNot* (*clause* C)
unfolding *CNot-def true-annots-def clause-def* **by** *auto*

show $C \in set$ (*raw-clauses* S)
using *cw* **by** *auto*
qed

lemma *wf-candidates-conflict-complete*:
assumes *wf*: *wf-twl-state* S **and**
c-mem: $C \in set$ (*raw-clauses* S) **and**
unsat: *trail* $S \models as$ *CNot* (*clause* C)
shows $C \in$ *candidates-conflict* S
proof −
def $M \equiv$ *raw-trail* S
def $N \equiv$ *twl.conc-init-clss* S
def $U \equiv$ *conc-learned-clss* S

note *MNU-defs* [*simp*] = *M-def N-def U-def*

obtain W UW **where** *cw-eq*: $C = TWL\text{-}Clause\ W\ UW$
by (*cases* C, *blast*)

have *wf-c*: *wf-twl-cls* M C
using *wf c-mem* **unfolding** *wf-twl-state-def* **by** *simp*

have *w-nw*:
distinct W
$length\ W < 2 \Longrightarrow set\ UW \subseteq set\ W$
$\bigwedge L\ L'.\ L \in set\ W \Longrightarrow -L \in lits\text{-}of\text{-}l\ M \Longrightarrow L' \in set\ UW \Longrightarrow L' \notin set\ W \Longrightarrow -L' \in lits\text{-}of\text{-}l\ M$
using *wf-c* **unfolding** *cw-eq* **by** (*auto simp*: *image-image*)

have $\bigwedge L.\ L \in set\ (raw\text{-}clause\ C) \Longrightarrow -L \in lits\text{-}of\text{-}l\ M$
unfolding *M-def* **using** *unsat*[*unfolded CNot-def true-annots-def*, *simplified*]
by (*auto simp*: *clause-def*)
then have $set\ (raw\text{-}clause\ C) \subseteq uminus$ ‘ $lits\text{-}of\text{-}l\ M$
by (*metis imageI subsetI uminus-of-uminus-id*)
then have $set\ W \subseteq uminus$ ‘ $lits\text{-}of\text{-}l\ M$
using *cw-eq* **by** (*auto simp*: *raw-clause-def*)
then have *subset*: $set\ W \subseteq uminus$ ‘ $lits\text{-}of\text{-}l\ M$

by (*simp add*: *w-nw*(*1*))

have W = *watched C*
using *cw-eq twl-clause.sel*(*1*) **by** *simp*
then show *?thesis*
using *MNU-defs c-mem subset candidates-conflict-def* **by** *blast*
qed

typedef *'v wf-twl* = {*S*::*'v twl-state. wf-twl-state S*}
morphisms *rough-state-of-twl twl-of-rough-state*
proof −
have *TWL-State* ([]::(*'v*, *'v twl-clause*) *ann-lits*)
[] [] *0 None* ∈ {*S*:: *'v twl-state. wf-twl-state S*}
by (*auto simp*: *wf-twl-state-def*)
then show *?thesis* **by** *auto*
qed

lemma [*code abstype*]:
twl-of-rough-state (*rough-state-of-twl S*) = *S*
by (*fact CDCL-Two-Watched-Literals.wf-twl.rough-state-of-twl-inverse*)

lemma *wf-twl-state-rough-state-of-twl*[*simp*]: *wf-twl-state* (*rough-state-of-twl S*)
using *rough-state-of-twl* **by** *auto*

abbreviation *candidates-conflict-twl* :: *'v wf-twl* ⇒ *'v twl-clause set* **where**
candidates-conflict-twl S ≡ *candidates-conflict* (*rough-state-of-twl S*)

abbreviation *candidates-propagate-twl* :: *'v wf-twl* ⇒ (*'v literal* × *'v twl-clause*) *set* **where**
candidates-propagate-twl S ≡ *candidates-propagate* (*rough-state-of-twl S*)

abbreviation *raw-trail-twl* :: *'a wf-twl* ⇒ (*'a*, *'a twl-clause*) *ann-lits* **where**
raw-trail-twl S ≡ *raw-trail* (*rough-state-of-twl S*)

abbreviation *trail-twl* :: *'a wf-twl* ⇒ (*'a*, *'a literal multiset*) *ann-lits* **where**
trail-twl S ≡ *trail* (*rough-state-of-twl S*)

abbreviation *raw-clauses-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-clauses-twl S ≡ *raw-clauses* (*rough-state-of-twl S*)

abbreviation *raw-init-clss-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-init-clss-twl S ≡ *raw-init-clss* (*rough-state-of-twl S*)

abbreviation *raw-learned-clss-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-learned-clss-twl S ≡ *raw-learned-clss* (*rough-state-of-twl S*)

abbreviation *conc-learned-clss-twl* :: *'a wf-twl* ⇒ *'a clauses* **where**
conc-learned-clss-twl S ≡ *conc-learned-clss* (*rough-state-of-twl S*)

abbreviation *backtrack-lvl-twl* **where**
backtrack-lvl-twl S ≡ *backtrack-lvl* (*rough-state-of-twl S*)

abbreviation *raw-conflicting-twl* **where**
raw-conflicting-twl S ≡ *raw-conflicting* (*rough-state-of-twl S*)

lemma *wf-candidates-twl-conflict-complete*:
assumes

c-mem: $C \in$ *set* (*raw-clauses-twl S*) **and**
unsat: *trail-twl S* $\models$*as CNot* (*clause C*)
shows $C \in$ *candidates-conflict-twl S*
using *c-mem unsat wf-candidates-conflict-complete wf-twl-state-rough-state-of-twl* **by** *blast*

abbreviation *update-backtrack-lvl* **where**
update-backtrack-lvl k S $\equiv$
TWL-State (*raw-trail S*) (*raw-init-clss S*) (*raw-learned-clss S*) *k* (*raw-conflicting S*)

abbreviation *update-conflicting* **where**
update-conflicting C S $\equiv$
TWL-State (*raw-trail S*) (*raw-init-clss S*) (*raw-learned-clss S*) (*backtrack-lvl S*) *C*

Abstract 2-WL

definition *tl-trail* **where**
tl-trail S =
TWL-State (*tl* (*raw-trail S*)) (*raw-init-clss S*) (*raw-learned-clss S*) (*backtrack-lvl S*)
(*raw-conflicting S*)

locale *abstract-twl* =
fixes
watch :: *'v twl-state* ⇒ *'v literal list* ⇒ *'v twl-clause* **and**
rewatch :: *'v literal* ⇒ *'v twl-state* ⇒
'v twl-clause ⇒ *'v twl-clause* **and**
restart-learned :: *'v twl-state* ⇒ *'v twl-clause list*
assumes
clause-watch: *no-dup* (*raw-trail S*) ⟹ *clause* (*watch S C*) = *mset C* **and**
wf-watch: *no-dup* (*raw-trail S*) ⟹ *wf-twl-cls* (*raw-trail S*) (*watch S C*) **and**
clause-rewatch: *clause* (*rewatch L' S C'*) = *clause C'* **and**
wf-rewatch:
no-dup (*raw-trail S*) ⟹ *undefined-lit* (*raw-trail S*) (*lit-of L*) ⟹
wf-twl-cls (*raw-trail S*) *C'* ⟹
wf-twl-cls (*L* # *raw-trail S*) (*rewatch* (*lit-of L*) *S C'*)
and
restart-learned: *mset* (*restart-learned S*) ⊆# *mset* (*raw-learned-clss S*) — We need *mset* and not *set* to take care of duplicates.
begin

definition
cons-trail :: (*'v*, *'v twl-clause*) *ann-lit* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
cons-trail L S =
TWL-State (*L* # *raw-trail S*) (*map* (*rewatch* (*lit-of L*) *S*) (*raw-init-clss S*))
(*map* (*rewatch* (*lit-of L*) *S*) (*raw-learned-clss S*)) (*backtrack-lvl S*) (*raw-conflicting S*)

definition
add-init-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
add-init-cls C S =
TWL-State (*raw-trail S*) (*watch S C* # *raw-init-clss S*) (*raw-learned-clss S*) (*backtrack-lvl S*)
(*raw-conflicting S*)

definition
add-learned-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where

add-learned-cls C S =
TWL-State (*raw-trail S*) (*raw-init-clss S*) (*watch S C* # *raw-learned-clss S*) (*backtrack-lvl S*)
(*raw-conflicting S*)

definition
remove-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
remove-cls C S =
TWL-State (*raw-trail S*)
(*removeAll-cond* (λ*D*. *clause D* = *mset C*) (*raw-init-clss S*))
(*removeAll-cond* (λ*D*. *clause D* = *mset C*) (*raw-learned-clss S*))
(*backtrack-lvl S*)
(*raw-conflicting S*)

definition *init-state* :: *'v literal list list* ⇒ *'v twl-state* **where**
init-state N = *fold add-init-cls N* (*TWL-State* [] [] [] *0 None*)

lemma *unchanged-fold-add-init-cls*:
raw-trail (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *M*
raw-learned-clss (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *U*
backtrack-lvl (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *k*
raw-conflicting (*fold add-init-cls Cs* (*TWL-State M N U k C*)) = *C*
by (*induct Cs arbitrary*: *N*) (*auto simp*: *add-init-cls-def*)

lemma *unchanged-init-state*[*simp*]:
raw-trail (*init-state N*) = []
raw-learned-clss (*init-state N*) = []
backtrack-lvl (*init-state N*) = *0*
raw-conflicting (*init-state N*) = *None*
unfolding *init-state-def* **by** (*rule unchanged-fold-add-init-cls*)+

lemma *raw-clss-l-raw-clss*[*simp*]: *CDCL-Two-Watched-Literals.raw-clss* = *raw-clss-l*
apply (*rule sym*)
using *mset-map* **by** *blast*

lemma *conc-init-clss*[*simp*]:
twl.conc-init-clss (*TWL-State M N U k C*) = *raw-clss-l N*
proof −
have ⋀*t*. *twl.conc-clauses* (*t*::*'a twl-state*) − *conc-learned-clss t* = *raw-clss-l* (*raw-init-clss t*)
by (*metis* (*no-types*) *diff-union-cancelR map-append raw-clss-l-raw-clss twl.conc-clauses-def*
union-code)
then show *?thesis*
by (*simp add*: *twl.conc-init-clss-def*)
qed

lemma *clauses-init-fold-add-init*:
no-dup M ⟹
twl.conc-init-clss (*fold add-init-cls Cs* (*TWL-State M N U k C*)) =
clauses-of-l Cs + *raw-clss-l N*
by (*induct Cs arbitrary*: *N*) (*auto simp*: *add-init-cls-def clause-watch comp-def ac-simps*
clause-def[*symmetric*])

lemma *init-clss-init-state*[*simp*]: *twl.conc-init-clss* (*init-state N*) = *clauses-of-l N*
unfolding *init-state-def* **by** (*subst clauses-init-fold-add-init*) *simp-all*

definition *restart'* **where**

```
  restart′ S = TWL-State [] (raw-init-clss S) (restart-learned S) 0 None

end
```

Instanciation of the previous locale

```
definition watch-nat :: ′v twl-state ⇒ ′v literal list ⇒ ′v twl-clause where
  watch-nat S C =
   (let
      C′ = remdups C;
      neg-not-assigned = filter (λL. −L ∉ lits-of-l (raw-trail S)) C′;
      neg-assigned-sorted-by-trail = filter (λL. L ∈ set C) (map (λL. −lit-of L) (raw-trail S));
      W = take 2 (neg-not-assigned @ neg-assigned-sorted-by-trail);
      UW = foldr remove1 W C
    in TWL-Clause W UW)

lemma list-cases2:
  fixes l :: ′a list
  assumes
    l = [] ⟹ P and
    ⋀x. l = [x] ⟹ P and
    ⋀x y xs. l = x # y # xs ⟹ P
  shows P
  by (metis assms list.collapse)

lemma filter-in-list-prop-verifiedD:
  assumes [L←P . Q L] = l
  shows ∀x ∈ set l. x ∈ set P ∧ Q x
  using assms by auto

lemma no-dup-filter-diff:
  assumes n-d: no-dup M and H: [L←map (λL. − lit-of L) M. L ∈ set C] = l
  shows distinct l
  unfolding H[symmetric]
  apply (rule distinct-filter)
  using n-d by (induction M) auto

lemma watch-nat-lists-disjointD:
  assumes
    l: [L←remdups C. − L ∉ lits-of-l (raw-trail S)] = l and
    l′: [L←map (λL. − lit-of L) (raw-trail S) . L ∈ set C] = l′
  shows ∀x ∈ set l. ∀y ∈ set l′. x ≠ y
  by (auto simp: l[symmetric] l′[symmetric] lits-of-def image-image)

lemma watch-nat-list-cases-witness[consumes 2, case-names Nil-Nil Nil-single Nil-other
  single-Nil single-other other]:
  fixes
    C :: ′v literal list and
    S :: ′v twl-state
  defines
    xs ≡ [L←remdups C. − L ∉ lits-of-l (raw-trail S)] and
    ys ≡ [L←map (λL. − lit-of L) (raw-trail S) . L ∈ set C]
  assumes
    n-d: no-dup (raw-trail S) and
    Nil-Nil: xs = [] ⟹ ys = [] ⟹ P and
    Nil-single:
```

$\bigwedge a.\ xs = [] \Longrightarrow ys = [a] \Longrightarrow a \in set\ C \Longrightarrow P$ **and**
Nil-other: $\bigwedge a\ b\ ys'.\ xs = [] \Longrightarrow ys = a\ \#\ b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
single-Nil: $\bigwedge a.\ xs = [a] \Longrightarrow ys = [] \Longrightarrow P$ **and**
single-other: $\bigwedge a\ b\ ys'.\ xs = [a] \Longrightarrow ys = b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
other: $\bigwedge a\ b\ xs'.\ xs = a\ \#\ b\ \#\ xs' \Longrightarrow a \neq b \Longrightarrow P$
shows P
proof −
note *xs-def*[*simp*] **and** *ys-def*[*simp*]
have *dist*: $\bigwedge P.$ *distinct* [L←*remdups* C . $P\ L$]
by *auto*
then have H: $\bigwedge a\ b\ P\ xs.$ [L←*remdups* C . $P\ L$] $= a\ \#\ b\ \#\ xs \Longrightarrow a \neq b$
by (*metis distinct-length-2-or-more*)
show *?thesis*
apply (*cases* [L←*remdups* C. $- L \notin$ *lits-of-l* (*raw-trail* S)]
rule: *list-cases2*;
cases [L←*map* ($\lambda L.\ -$ *lit-of* L) (*raw-trail* S) . $L \in$ *set* C] *rule*: *list-cases2*)
using *Nil-Nil* **apply** *simp*
using *Nil-single* **apply** (*force dest*: *filter-in-list-prop-verifiedD*)
using *Nil-other no-dup-filter-diff*[*OF n-d*, *of C*]
apply *fastforce*
using *single-Nil* **apply** *simp*
using *single-other xs-def ys-def* **apply** (*metis list.set-intros*(*1*) *watch-nat-lists-disjointD*)
using *single-other* **unfolding** *xs-def ys-def* **apply** (*metis list.set-intros*(*1*)
watch-nat-lists-disjointD)
using *other xs-def ys-def* **by** (*metis H*)+
qed

lemma *watch-nat-list-cases* [*consumes 1*, *case-names Nil-Nil Nil-single Nil-other single-Nil single-other other*]:
fixes
C :: $'v$ *literal list* **and**
S :: $'v$ *twl-state*
defines
$xs \equiv$ [L←*remdups* C . $- L \notin$ *lits-of-l* (*raw-trail* S)] **and**
$ys \equiv$ [L←*map* ($\lambda L.\ -$ *lit-of* L) (*raw-trail* S) . $L \in$ *set* C]
assumes
n-d: *no-dup* (*raw-trail* S) **and**
Nil-Nil: $xs = [] \Longrightarrow ys = [] \Longrightarrow P$ **and**
Nil-single:
$\bigwedge a.\ xs = [] \Longrightarrow ys = [a] \Longrightarrow a \in set\ C \Longrightarrow P$ **and**
Nil-other: $\bigwedge a\ b\ ys'.\ xs = [] \Longrightarrow ys = a\ \#\ b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
single-Nil: $\bigwedge a.\ xs = [a] \Longrightarrow ys = [] \Longrightarrow P$ **and**
single-other: $\bigwedge a\ b\ ys'.\ xs = [a] \Longrightarrow ys = b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
other: $\bigwedge a\ b\ xs'.\ xs = a\ \#\ b\ \#\ xs' \Longrightarrow a \neq b \Longrightarrow P$
shows P
using *watch-nat-list-cases-witness*[*OF n-d*, *of C P*]
Nil-Nil Nil-single Nil-other single-Nil single-other other
unfolding *xs-def*[*symmetric*] *ys-def*[*symmetric*] **by** *auto*

lemma *watch-nat-lists-set-union-witness*:
fixes
C :: $'v$ *literal list* **and**
S :: $'v$ *twl-state*
defines
$xs \equiv$ [L←*remdups* C. $- L \notin$ *lits-of-l* (*raw-trail* S)] **and**
$ys \equiv$ [L←*map* ($\lambda L.\ -$ *lit-of* L) (*raw-trail* S) . $L \in$ *set* C]

assumes *n-d*: *no-dup* (*raw-trail S*)
shows *set C* = *set xs* ∪ *set ys*
using *n-d* **unfolding** *xs-def ys-def* **by** (*auto simp*: *lits-of-def comp-def uminus-lit-swap*)

lemma *mset-intersection-inclusion*: $A + (B - A) = B \longleftrightarrow A \subseteq\# B$
apply (*rule iffI*)
apply (*metis mset-le-add-left*)
by (*auto simp*: *ac-simps multiset-eq-iff subseteq-mset-def*)

lemma *clause-watch-nat*:
assumes *no-dup* (*raw-trail S*)
shows *clause* (*watch-nat S C*) = *mset C*
using *assms*
apply (*cases rule*: *watch-nat-list-cases*[*OF assms*(*1*), *of C*])
by (*auto dest*: *filter-in-list-prop-verifiedD simp*: *watch-nat-def multiset-eq-iff raw-clause-def clause-def*)

lemma *index-uminus-index-map-uminus*:
$-a \in$ *set L* $\Longrightarrow$ *index L* $(-a)$ = *index* (*map uminus L*) (*a*::'*a literal*)
by (*induction L*) *auto*

lemma *index-filter*:
$a \in$ *set L* $\Longrightarrow b \in$ *set L* $\Longrightarrow P\ a \Longrightarrow P\ b \Longrightarrow$
index L a $\leq$ *index L b* $\longleftrightarrow$ *index* (*filter P L*) *a* $\leq$ *index* (*filter P L*) *b*
by (*induction L*) *auto*

lemma *foldr-remove1-W-Nil*[*simp*]: *foldr remove1 W* [] = []
by (*induct W*) *auto*

lemma *image-lit-of-mmset-of-mlit*[*simp*]:
lit-of ‘ *mmset-of-mlit* ‘ *A* = *lit-of* ‘ *A*
unfolding *comp-def*
using [[*simp-trace*]]**by** (*simp add*: *image-image comp-def*)

lemma *distinct-filter-eq*:
assumes *distinct xs*
shows [*L*← *xs*. *L* = *a*] = (*if a* ∈ *set xs then* [*a*] *else* [])
using *assms* **by** (*induction xs*) *auto*

lemma *no-dup-distinct-map-uminus-lit-of*:
no-dup xs $\Longrightarrow$ *distinct* (*map* (λ*L*. − *lit-of L*) *xs*)
by (*induction xs*) *auto*

lemma *wf-watch-witness*:
fixes *C* :: '*v literal list* **and**
S :: '*v twl-state*
defines
ass: *neg-not-assigned* ≡ *filter* (λ*L*. −*L* ∉ *lits-of-l* (*raw-trail S*)) (*remdups C*) **and**
tr: *neg-assigned-sorted-by-trail* ≡ *filter* (λ*L*. *L* ∈ *set C*) (*map* (λ*L*. −*lit-of L*) (*raw-trail S*))
defines
W: *W* ≡ *take 2* (*neg-not-assigned* @ *neg-assigned-sorted-by-trail*)
assumes
n-d[*simp*]: *no-dup* (*raw-trail S*)
shows *wf-twl-cls* (*raw-trail S*) (*TWL-Clause W* (*foldr remove1 W C*))
unfolding *wf-twl-cls.simps struct-wf-twl-cls.simps*
proof (*intro conjI*, *goal-cases*)

case *1*
then show *?case* **using** *n-d W* **unfolding** *ass tr*
apply (*cases rule*: *watch-nat-list-cases-witness*[*of S C*, *OF n-d*])
by (*auto simp*: *distinct-mset-add-single*)
next
case *2*
then show *?case* **unfolding** *W* **by** *simp*
next
case *3*
show *?case* **using** *n-d*
proof (*cases rule*: *watch-nat-list-cases-witness*[*of S C*])
case *Nil-Nil*
then have $set\ C = set\ [] \cup set\ []$
using *watch-nat-lists-set-union-witness n-d* **by** *metis*
then show *?thesis*
by *simp*
next
case (*Nil-single a*)
moreover have $\bigwedge x.\ set\ C = \{a\} \Longrightarrow -\ a \in lits\text{-}of\text{-}l\ (trail\ S) \Longrightarrow x \in set\ (remove1\ a\ C) \Longrightarrow$
$x = a$
using *notin-set-remove1* **by** *auto*
ultimately show *?thesis*
using *watch-nat-lists-set-union-witness*[*of S C*] *3* **by** (*auto simp*: *W ass tr comp-def*)
next
case *Nil-other*
then show *?thesis*
using *3* **by** (*auto simp*: *W ass tr*)
next
case (*single-Nil a*)
show *?thesis*
using *watch-nat-lists-set-union-witness*[*of S C*] *3*
by (*fastforce simp add*: *W ass tr single-Nil comp-def distinct-filter-eq*
no-dup-distinct-map-uminus-lit-of min-def)
next
case *single-other*
then show *?thesis*
using *3* **by** (*auto simp*: *W ass tr*)
next
case *other*
then show *?thesis*
using *3* **by** (*auto simp*: *W ass tr*)
qed
next
case *4* **note** -[*simp*] = *this*
show *?case*
using *n-d* **apply** (*cases rule*: *watch-nat-list-cases-witness*[*of S C*])
apply (*auto dest*: *filter-in-list-prop-verifiedD*
simp: *W ass tr lits-of-def filter-empty-conv*)[*4*]
using *watch-nat-lists-set-union-witness*[*of S C*]
by (*force dest*: *filter-in-list-prop-verifiedD simp*: *W ass tr lits-of-def*)+
next
case *5*
from *n-d* **show** *?case*
proof (*cases rule*: *watch-nat-list-cases-witness*[*of S C*])
case *Nil-Nil*
then show *?thesis* **by** (*auto simp*: *W ass tr*)

next
  case *Nil-single*
  then show *?thesis*
    using *watch-nat-lists-set-union-witness*[*of S C*] *tr* **by** (*fastforce simp*: *W ass*)
next
  case *Nil-other*
  then show *?thesis*
    unfolding *watched-only-lazy-updates.simps Ball-def*
    apply (*intro allI impI*)
    apply (*subst index-uminus-index-map-uminus*,
      simp add: *index-uminus-index-map-uminus lits-of-def o-def*)
    apply (*subst index-uminus-index-map-uminus*,
      simp add: *index-uminus-index-map-uminus lits-of-def o-def*)

    apply (*subst index-filter*[*of* - - - $\lambda L.\ L \in$ *set C*])
    by (*auto dest*: *filter-in-list-prop-verifiedD*
      simp: *uminus-lit-swap lits-of-def o-def W ass tr dest*: *in-diffD*)
next
  case *single-Nil*
  then show *?thesis*
    using *watch-nat-lists-set-union-witness*[*of S C*] *tr* **by** (*fastforce simp*: *W ass*)
next
  case *single-other*
  then show *?thesis*
    unfolding *watched-only-lazy-updates.simps Ball-def*
    apply (*clarify*)
    apply (*subst index-uminus-index-map-uminus*,
      simp add: *index-uminus-index-map-uminus lits-of-def image-image o-def*)
    apply (*subst index-uminus-index-map-uminus*,
      simp add: *index-uminus-index-map-uminus lits-of-def o-def*)

    apply (*subst index-filter*[*of* - - - $\lambda L.\ L \in$ *set C*])
    by (*auto dest*: *filter-in-list-prop-verifiedD*
      simp: *W ass tr uminus-lit-swap lits-of-def o-def dest*: *in-diffD*)
next
  case *other*
  then show *?thesis*
    unfolding *watched-only-lazy-updates.simps*
    apply *clarify*
    apply (*subst index-uminus-index-map-uminus*,
      simp add: *index-uminus-index-map-uminus lits-of-def o-def*)[*1*]
    apply (*subst index-uminus-index-map-uminus*,
      simp add: *index-uminus-index-map-uminus lits-of-def o-def*)[*1*]

    apply (*subst index-filter*[*of* - - - $\lambda L.\ L \in$ *set C*])
    by (*auto dest*: *filter-in-list-prop-verifiedD*
      simp: *index-uminus-index-map-uminus lits-of-def o-def uminus-lit-swap*
        W ass tr)
qed
qed

lemma *wf-watch-nat*: *no-dup* (*raw-trail S*) $\Longrightarrow$ *wf-twl-cls* (*raw-trail S*) (*watch-nat S C*)
  using *wf-watch-witness*[*of S C*] *watch-nat-def* **by** *metis*

definition
  rewatch-nat ::

$'v$ *literal* ⇒ $'v$ *twl-state* ⇒ $'v$ *twl-clause* ⇒ $'v$ *twl-clause*
where
rewatch-nat L S C =
(*if* $-$ $L \in$ *set* (*watched* C) *then*
case filter ($\lambda L'$. $L' \notin$ *set* (*watched* C) $\wedge$ $-$ $L' \notin$ *insert* L (*lits-of-l* (*trail* S)))
(*unwatched* C) *of*
[] ⇒ C
| L' # - ⇒
TWL-Clause (L' # *remove1* ($-L$) (*watched* C)) ($-L$ # *remove1* L' (*unwatched* C))
else
C)

lemma *clause-rewatch-nat*:
fixes UW :: $'v$ *literal list* **and**
S :: $'v$ *twl-state* **and**
L :: $'v$ *literal* **and** C :: $'v$ *twl-clause*
shows *clause* (*rewatch-nat* L S C) = *clause* C
using *List.set-remove1-subset*[*of* $-L$ *watched* C]
apply (*cases* C)
by (*auto simp*: *raw-clause-def rewatch-nat-def ac-simps multiset-eq-iff clause-def*
split: *list.split*
dest: *filter-in-list-prop-verifiedD*)

lemma *filter-sorted-list-of-multiset-Nil*:
[$x \leftarrow$ *sorted-list-of-multiset* M. p x] = [] $\longleftrightarrow$ ($\forall x \in\#$ M. $\neg$ p x)
by *auto* (*metis empty-iff filter-set list.set*(*1*) *member-filter set-sorted-list-of-multiset*)

lemma *filter-sorted-list-of-multiset-ConsD*:
[$x \leftarrow$ *sorted-list-of-multiset* M. p x] = x # xs $\Longrightarrow$ p x
by (*metis filter-set insert-iff list.set*(*2*) *member-filter*)

lemma *mset-minus-single-eq-mempty*:
a $-$ {#b#} = {#} $\longleftrightarrow$ a = {#b#} $\vee$ a = {#}
by (*metis Multiset.diff-cancel add.right-neutral diff-single-eq-union*
diff-single-trivial zero-diff)

lemma *size-mset-le-2-cases*:
assumes *size* $W \leq 2$
shows W = {#} $\vee$ ($\exists$ a. W = {#a#}) $\vee$ ($\exists$ a b. W = {#a,b#})
proof $-$
have *size* W = *0* $\vee$ *size* W = *1* $\vee$ *size* W = *2*
using *assms* **by** *linarith*
then show *?thesis*
using *assms* **by** (*fastforce elim*!: *size-mset-SucE simp*: *Num.numeral-2-eq-2*)
qed

lemma *filter-sorted-list-of-multiset-eqD*:
assumes [$x \leftarrow$ *sorted-list-of-multiset* A. p x] = x # xs (**is** *?comp* = -)
shows $x \in\#$ A
proof $-$
have $x \in$ *set ?comp*
using *assms* **by** *simp*
then have $x \in$ *set* (*sorted-list-of-multiset* A)
by *simp*
then show $x \in\#$ A
by *simp*

```
qed

lemma clause-rewatch-witness':
  assumes
    wf: wf-twl-cls (raw-trail S) C and
    undef: undefined-lit (raw-trail S) (lit-of L)
  shows wf-twl-cls (L # raw-trail S) (rewatch-nat (lit-of L) S C)
proof (cases − lit-of L ∈ set (watched C))
  case False
  then show ?thesis
    apply (cases C)
    using wf undef unfolding rewatch-nat-def
    by (auto simp: uminus-lit-swap Decided-Propagated-in-iff-in-lits-of-l comp-def)
next
  case falsified: True

  let ?unwatched-nonfalsified =
    [L'← unwatched C. L' ∉ set (watched C) ∧ − L' ∉ insert (lit-of L) (lits-of-l (trail S))]
  obtain W UW where C: C = TWL-Clause W UW
    by (cases C)

  show ?thesis
  proof (cases ?unwatched-nonfalsified)
    case Nil
    show ?thesis
      using falsified Nil
      apply (simp only: wf-twl-cls.simps if-True list.cases C rewatch-nat-def
        struct-wf-twl-cls.simps)
      apply (intro conjI)
      proof goal-cases
        case 1
        then show ?case using wf C by simp
      next
        case 2
        then show ?case using wf C by simp
      next
        case 3
        then show ?case using wf C by simp
      next
        case 4
        have ⋀p l. filter p (unwatched C) ≠ [] ∨ l ∉ set UW ∨ ¬ p l
          unfolding C by (metis (no-types) filter-empty-conv twl-clause.sel(2))
        then show ?case
          using 4(2) C by auto
      next
        case 5
        then show ?case
          using wf by (fastforce simp add: C comp-def uminus-lit-swap)
      qed
  next
    case (Cons L' Ls)
    show ?thesis
      unfolding rewatch-nat-def
      using falsified Cons
      apply (simp only: wf-twl-cls.simps if-True list.cases C struct-wf-twl-cls.simps)
      apply (intro conjI)
```

proof *goal-cases*
  case *1*
  have *distinct* (*watched* (*TWL-Clause W UW*))
    using *wf* **unfolding** *C* **by** *auto*
  moreover have $L' \notin$ *set* (*remove1* (−*lit-of L*) (*watched* (*TWL-Clause W UW*)))
    using *1*(*2*) *not-gr0* **by** (*fastforce dest*: *filter-in-list-prop-verifiedD in-diffD*)
  ultimately show *?case*
    by (*auto simp*: *distinct-mset-single-add*)
next
  case *2*
  have *f2*: [*l*←*unwatched* (*TWL-Clause W UW*) . $l \notin$ *set* (*watched* (*TWL-Clause W UW*))
    $\wedge - l \notin$ *insert* (*lit-of L*) (*lits-of-l* (*trail S*))] $\neq$ []
    using *2*(*2*) **by** *simp*
  then have ¬ *set UW* ⊆ *set W*
     using *2* **by** (*auto simp add*: *filter-empty-conv*)
  then show *?case*
    using *wf C 2*(*1*) **by** (*auto simp*: *length-remove1*)
next
  case *3*
  have *W*: *length W* ≤ *Suc 0* ⟷ *length W* = *0* ∨ *length W* = *Suc 0*
    by *linarith*
  show *?case*
    using *wf C 3* **by** (*auto simp*: *length-remove1 W length-list-Suc-0 dest*!: *subset-singletonD*)
next
  case *4*
  have *H*: ∀ *L*∈ *set W*. − *L* ∈ *lits-of-l* (*trail S*) ⟶
    (∀ *L'*∈ *set UW*. $L' \notin$ *set W* ⟶ − *L'* ∈ *lits-of-l* (*trail S*))
    using *wf* **by** (*auto simp*: *C*)
  have *W*: *length W* ≤ *2* **and** *W-UW*: *length W* < *2* ⟶ *set UW* ⊆ *set W*
    using *wf* **by** (*auto simp*: *C*)
  have *distinct*: *distinct W*
    using *wf* **by** (*auto simp*: *C*)
  show *?case*
    using *4*
    unfolding *C watched-only-lazy-updates.simps Ball-def twl-clause.sel*
      watched-wf-twl-cls.simps
    apply (*intro allI impI*)
    apply (*rename-tac xW xUW*)
    apply (*case-tac* − *lit-of L* = *xW*; *case-tac xW* = *xUW*; *case-tac L'* = *xW*)
            apply (*auto simp*: *uminus-lit-swap*)[*2*]
          apply (*force dest*: *filter-in-list-prop-verifiedD*)
         using *H distinct* **apply** (*fastforce*)
       using *distinct* **apply** (*fastforce*)
      using *distinct* **apply** (*fastforce*)
     apply (*force dest*: *filter-in-list-prop-verifiedD*)
    using *H* **by** (*auto simp*: *uminus-lit-swap*)
next
  case *5*
  have *H*: ∀ *x*. *x* ∈ *set W* ⟶ − *x* ∈ *lits-of-l* (*trail S*) ⟶ (∀ *x*. *x* ∈ *set UW* ⟶ $x \notin$ *set W*
    ⟶ − *x* ∈ *lits-of-l* (*trail S*))
    using *wf* **by** (*auto simp*: *C*)
  show *?case*
    unfolding *C watched-only-lazy-updates.simps Ball-def*
    proof (*intro allI impI conjI*, *goal-cases*)
      case (*1 xW x*)
      show *?case*

```
            proof (cases − lit-of L = xW)
              case True
              then show ?thesis
                by (cases xW = x) (auto simp: uminus-lit-swap)
            next
              case False note LxW = this
              have f9: L′ ∈ set [l←unwatched C. l ∉ set (watched (TWL-Clause W UW))
                  ∧ − l ∉ lits-of-l (L # raw-trail S)]
                using 1(2) 5 C by auto
              moreover then have f11: − xW ∈ lits-of-l (trail S)
                using 1(3) LxW by (auto simp: uminus-lit-swap)
              moreover then have xW ∉ set W
                using f9 1(2) H by (auto simp: C)
              ultimately have False
                using 1 by auto
              then show ?thesis
                by fast
            qed
          qed
        qed
    qed
qed


interpretation twl: abstract-twl watch-nat rewatch-nat raw-learned-clss
  apply unfold-locales
  apply (rule clause-watch-nat; simp add: image-image comp-def)
  apply (rule wf-watch-nat; simp add: image-image comp-def)
  apply (rule clause-rewatch-nat)
  apply (rule clause-rewatch-witness′; simp add: image-image comp-def)
  apply (simp)
  done

interpretation twl2: abstract-twl watch-nat rewatch-nat λ-. []
  apply unfold-locales
  apply (rule clause-watch-nat; simp add: image-image comp-def)
  apply (rule wf-watch-nat; simp add: image-image comp-def)
  apply (rule clause-rewatch-nat)
  apply (rule clause-rewatch-witness′; simp add: image-image comp-def)
  apply (simp)
  done

end
```