Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

March 15, 2016

Contents

theory *Wellfounded-More*
imports *Main*

begin

1 Transitions

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

1.1 More theorems about Closures

This is the equivalent of $?r \leq ?s \Longrightarrow ?r^{**} \leq ?s^{**}$ for *tranclp*

lemma *tranclp-mono-explicit*:

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

lemma *tranclp-mono*:
assumes *mono*: $r \le s$
shows $r^{++} \le s^{++}$
  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 $?r^{****} = ?r^{**}$

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

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

This theorem already exists as $?r^{**}\ ?a\ ?b \equiv\ ?a =\ ?b \vee\ ?r^{++}\ ?a\ ?b$ (and sledgehammer uses it), but it makes sense to duplicate it, because it is unclear how stable the lemmas in Nitpick 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*)

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 arbitrary*:)
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 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 $= (\lambda S\ S'.\ rtranclp\ transf\ S\ S' \wedge (\forall\, S''.\ \neg\ transf\ S'\ S''))$

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*)

1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:
assumes $wf{:}wf\ \{(x, y).\ R\ y\ x\}$
shows $\exists b.\ R^{**}\ a\ b \land no\text{-}step\ R\ b$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have $H: \bigwedge b.\ \neg\ R^{**}\ a\ b \lor \neg no\text{-}step\ R\ b$
by *blast*
def $F \equiv rec\text{-}nat\ a\ (\lambda i\ b.\ SOME\ c.\ R\ b\ c)$
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 $\forall j < Suc\ i.\ R\ (F\ j)\ (F\ (Suc\ j))$
using *H Suc* **by** (*simp add*: *less-Suc-eq*)
then show *?case* **by** *fast*
qed
}
then have $\forall j.\ R\ (F\ j)\ (F\ (Suc\ j))$ **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 $\exists b.\ full\ R\ a\ b$
using *wf-exists-normal-form*[*OF assms*] **unfolding** *full-def* **by** *blast*

1.4 More Well-Foundedness

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

- link between *wf* and infinite chains: $wf\ ?r = (\nexists f.\ \forall i.\ (f\ (Suc\ i), f\ i) \in ?r)$, $[\![wf\ ?r; \bigwedge k.\ (?f\ (Suc\ k), ?f\ k) \notin ?r \Longrightarrow ?thesis]\!] \Longrightarrow ?thesis$

lemma *wf-if-measure-in-wf*:
$wf\ R \Longrightarrow (\bigwedge a\ b.\ (a, b) \in S \Longrightarrow (\nu\ a, \nu\ b) \in R) \Longrightarrow wf\ S$
by (*metis in-inv-image wfE-min wfI-min wf-inv-image*)

lemma *wfP-if-measure*: **fixes** $f :: 'a \Rightarrow nat$
shows $(\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow f\ y < f\ x) \Longrightarrow wf\ \{(y,x).\ P\ x \wedge g\ x\ y\}$
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 *wf-if-measure-f*:
assumes $wf\ r$
shows $wf\ \{(b, a).\ (f\ b, f\ a) \in r\}$
using *assms* **by** (*metis inv-image-def wf-inv-image*)

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

2 Various Lemmas

Close to $(\bigwedge n. \forall m<n. ?P\ m \Longrightarrow ?P\ n) \Longrightarrow ?P\ ?n$, but with a separation between zero and non-zero, and case names.

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 *?P* (*if ?Q then ?x else ?y*) = (¬ (*?Q* ∧ ¬ *?P ?x* ∨ ¬ *?Q* ∧ ¬ *?P ?y*)) can blow up goals (because of other if expression).

lemma *if-0-1-ge-0*[*simp*]:
  0 < (*if P then a else* (*0*::*nat*)) ⟷ *P* ∧ *0* < *a*
  by *auto*

Bounded function have not been defined in Isabelle.

definition *bounded* **where**
bounded f ⟷ (∃ *b*. ∀ *n*. *f n* ≤ *b*)

abbreviation *unbounded* :: (*'a* ⇒ *'b*::*ord*) ⇒ *bool* **where**
unbounded f ≡ ¬ *bounded f*

lemma *not-bounded-nat-exists-larger*:
  fixes *f* :: *nat* ⇒ *nat*
  assumes *unbound*: *unbounded f*
  shows ∃ *n*. *f n* > *m* ∧ *n* > n_0
proof (*rule ccontr*)
  assume *H*: ¬ *?thesis*
  have *finite* {*f n*|*n*. *n* ≤ n_0}
    by *auto*
  have ⋀*n*. *f n* ≤ *Max* ({*f n*|*n*. *n* ≤ n_0} ∪ {*m*})
    apply (*case-tac n* ≤ n_0)
    apply (*metis* (*mono-tags*, *lifting*) *Max-ge Un-insert-right* ‹*finite* {*f n* |*n*. *n* ≤ 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* ≤ n_0}›
      finite-insert insertI1 insert-not-empty leI sup-bot.right-neutral)
  then show *False*
    using *unbound* **unfolding** *bounded-def* **by** *auto*
qed

lemma *bounded-const-product*:
  fixes *k* :: *nat* **and** *f* :: *nat* ⇒ *nat*
  assumes *k* > *0*
  shows *bounded f* ⟷ *bounded* (λ*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 a property that can be expected from *bounded* holds.

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

3 More List

3.1 *upt*

The simplification rules are not very handy, because $[?i..<Suc\ ?j] = (if\ ?i \leq ?j\ then\ [?i..<?j]$ @ $[?j]\ else\ [])$ 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*]

lemma
  assumes $i \leq n - m$
  shows *take* i $[m..<n] = [m..<m+i]$
  by (*metis Nat.le-diff-conv2 add.commute assms diff-is-0-eq′ linear take-upt upt-conv-Nil*)

The counterpart for this lemma when $n - m < i$ is *length* $?xs \leq ?n \Longrightarrow$ *take* $?n\ ?xs = ?xs$. It is close to $?i + ?m \leq ?n \Longrightarrow$ *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

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

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

The converse of $?A$ @ $?B = [?m..<?n] \Longrightarrow ?A = [?m..<?m + length\ ?A]$

$?A$ @ $?B = [?m..<?n] \Longrightarrow ?B = [?m + length\ ?A..<?n]$ does not hold, for example if B 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

3.2 Lexicographic ordering

We are working a lot on lexicographic ordering over pairs.

lemma *list-length2-append-cons*:
  $[c,\ d] = ys\ @\ y\ \#\ ys' \longleftrightarrow (ys = [] \wedge y = c \wedge ys' = [d]) \vee (ys = [c] \wedge y = d \wedge ys' = [])$
  by (*cases ys*; *cases ys'*) *auto*

lemma *lexn2-conv*:
  $([a,\ b],\ [c,\ d]) \in lexn\ r\ 2 \longleftrightarrow (a,\ c) \in r \vee (a = c \wedge (b,\ d) \in r)$
  unfolding *lexn-conv* **by** (*auto simp add*: *list-length2-append-cons*)

3.3 Remove and Multiset equality

lemma *remove1-mset-single-add*:
  $a \neq b \Longrightarrow remove1\text{-}mset\ a\ (\{\#b\#\} + C) = \{\#b\#\} + remove1\text{-}mset\ a\ C$
  $remove1\text{-}mset\ a\ (\{\#a\#\} + C) = C$
  by (*auto simp*: *multiset-eq-iff*)

This is the sams as *remove1* under the assumptions of non-duplication inside a clause.

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

lemma *mset-map-mset-remove1-cond*:
  $mset\ (map\ mset\ (remove1\text{-}cond\ (\lambda L.\ mset\ L = mset\ a)\ C)) =$
    $remove1\text{-}mset\ (mset\ a)\ (mset\ (map\ mset\ C))$
  by (*induction C*) (*auto simp*: *ac-simps remove1-mset-single-add*)

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

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*)

abbreviation *union-mset-list* **where**
union-mset-list xs ys ≡ *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
end
theory *Prop-Logic*

imports *Main*

begin

4 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.

4.1 Definition and abstraction

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

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

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

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

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

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

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

lemma *propo-induct-arity*[*case-names nullary unary binary*]:
  fixes φ ψ :: *'v propo*
  assumes *nullary*: ($\bigwedge\varphi$ *x*. φ = *FF* ∨ φ = *FT* ∨ φ = *FVar x* ⟹ *P* φ)
  and *unary*: ($\bigwedge\psi$. *P* ψ ⟹ *P* (*FNot* ψ))
  and *binary*: ($\bigwedge\varphi$ ψ*1* ψ*2*. *P* ψ*1* ⟹ *P* ψ*2* ⟹ φ = *FAnd* ψ*1* ψ*2* ∨ φ = *FOr* ψ*1* ψ*2* ∨ φ = *FImp* ψ*1* ψ*2*

$\vee\ \varphi = FEq\ \psi 1\ \psi 2 \Longrightarrow P\ \varphi)$
shows $P\ \psi$
apply (*induct rule*: *propo.induct*)
using *assms* **by** *metis+*

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

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

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

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

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

inductive *wf-conn* :: *'v connective* ⇒ *'v propo list* ⇒ *bool* **for** *c* :: *'v connective* **where**
wf-conn-nullary[*simp*]: $(c = CT \vee c = CF \vee c = CVar\ v) \Longrightarrow wf\text{-}conn\ c\ []$ |
wf-conn-unary[*simp*]: $c = CNot \Longrightarrow wf\text{-}conn\ c\ [\psi]$ |
wf-conn-binary[*simp*]: $c \in binary\text{-}connectives \Longrightarrow wf\text{-}conn\ c\ (\psi\ \#\ \psi'\ \#\ [])$
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*)

4.2 properties of the abstraction

First we can define simplification rules.

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

lemma *wf-conn-list*:
  wf-conn c l $\Longrightarrow$ *conn c l* = *FT* $\longleftrightarrow$ (*c* = *CT* $\wedge$ *l* = [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FF* $\longleftrightarrow$ (*c* = *CF* $\wedge$ *l* = [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FVar x* $\longleftrightarrow$ (*c* = *CVar x* $\wedge$ *l* = [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FAnd a b* $\longleftrightarrow$ (*c* = *CAnd* $\wedge$ *l* = *a* # *b* # [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FOr a b* $\longleftrightarrow$ (*c* = *COr* $\wedge$ *l* = *a* # *b* # [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FEq a b* $\longleftrightarrow$ (*c* = *CEq* $\wedge$ *l* = *a* # *b* # [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FImp a b* $\longleftrightarrow$ (*c* = *CImp* $\wedge$ *l* = *a* # *b* # [])
  wf-conn c l $\Longrightarrow$ *conn c l* = *FNot a* $\longleftrightarrow$ (*c* = *CNot* $\wedge$ *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-connectives*
  shows *length l* = *2* $\longleftrightarrow$ *wf-conn c l*
proof
  assume *length l* = *2*
  then show *wf-conn c l* **using** *wf-conn-binary list-length2-decomp* **using** *conn* **by** *metis*
next
  assume *wf-conn c l*
  then show *length l* = *2* (**is** *?P l*)
    proof (*cases rule*: *wf-conn.induct*)
      case *wf-conn-nullary*
      then show *?P* [] **using** *conn binary-connectives-def*
        using *connective.distinct*(*11*) *connective.distinct*(*13*) *connective.distinct*(*9*) **by** *blast*
    next

```
      fix ψ :: 'v propo
      case wf-conn-unary
      then show ?P [ψ] using conn binary-connectives-def
        using connective.distinct  by blast
    next
      fix ψ ψ':: 'v propo
      show ?P [ψ, ψ'] by auto
    qed
qed
```

```
lemma wf-conn-not-list-length[iff]:
  fixes l :: 'v propo list
  shows wf-conn CNot l ⟷ 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 ∃ 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' ⟹ wf-conn c l ⟷ wf-conn c l'
proof –
  {
    fix l l'
    have length l = length l' ⟹ wf-conn c l ⟹ 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' ⟹ 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** $l\ \psi s :: 'v\ propo\ list$
  assumes *corr*: $wf\text{-}conn\ ca\ l$
  and *corr'*: $wf\text{-}conn\ c\ \psi s$
  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* $\psi'\ \psi''$)
  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

4.3 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* :: $'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$ (**infix** $\preceq$ *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*]:
$\varphi \preceq FAnd\ \psi\ \psi' \longleftrightarrow (\varphi = FAnd\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$ (**is** *?P FAnd*)
$\varphi \preceq FOr\ \psi\ \psi' \longleftrightarrow (\varphi = FOr\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$
$\varphi \preceq FEq\ \psi\ \psi' \longleftrightarrow (\varphi = FEq\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$
$\varphi \preceq FImp\ \psi\ \psi' \longleftrightarrow (\varphi = FImp\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$
proof –
have *wf-conn CAnd* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn CAnd* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn CAnd* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FAnd* **by** *auto*
next
have *wf-conn COr* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn COr* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn COr* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FOr* **by** *auto*
next
have *wf-conn CEq* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn CEq* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn CEq* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FEq* **by** *auto*
next
have *wf-conn CImp* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)

then have $\varphi \preceq$ *conn CImp* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn CImp* $[\psi, \psi'] \lor (\exists \psi''. \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FImp* **by** *auto*
qed

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

lemma *exists-c-conn*: $\exists$ *c l*. $\varphi =$ *conn c l* $\land$ *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* $\longleftrightarrow (\varphi =$ *conn c l* $\lor (\exists \psi \in$ *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 $\xi\ \xi'$ :: *'v propo list* **and** ψ :: *'v propo* **and** c :: *'v connective*
  assumes *corr*: *wf-conn c l* **and** *incl*: $\psi \in$ *set l*
  shows *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
proof (*cases c rule*: *connective-cases-arity-2*)
  case *nullary*
  then have *False* **using** *corr incl* **by** *auto*
  then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*) **by** *blast*
next
  case *binary* **note** c = *this*
  then obtain a b **where** *ab*: $l = [a,\ b]$
    using *wf-conn-bin-list-length list-length2-decomp corr* **by** *metis*
  then have $\psi = a \lor \psi = b$ **using** *incl* **by** *auto*
  then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
    using *ab c* **unfolding** *binary-connectives-def* **by** *auto*
next
  case *unary* **note** c = *this*
  fix φ :: *'v propo*
  have $l = [\psi]$ **using** *corr c incl split-list* **by** *force*
  then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*) **using** c **by** *auto*
qed

The set of variables is compatible with the subformula order.

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

4.4 Positions

Instead of 1 or 2 we use L or R

datatype *sign* = L | R

We use *nil* instead of ε.

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

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

lemma *finite-inj-comp-set*:
  fixes s :: *'v set*
  assumes *finite*: *finite s*
  and *inj*: *inj f*
  shows *card* ({*f p* |*p*. $p \in s$}) = *card s*
  using *finite*

```
proof (induct s rule: finite-induct)
  show card {f p |p. p ∈ {}} = card {} by auto
next
  fix x :: 'v and s:: 'v set
  assume f: finite s and notin: x ∉ s
  and IH: card {f p |p. p ∈ s} = card s
  have f': finite {f p |p. p ∈ insert x s} using f by auto
  have notin': f x ∉ {f p |p. p ∈ s} using notin inj injD by fastforce
  have {f p |p. p ∈ insert x s} = insert (f x) {f p |p. p∈ s} by auto
  then have card {f p |p. p ∈ insert x s} = 1 + card {f p |p. p ∈ s}
    using finite card-insert-disjoint f' notin' by auto
  moreover have ... = card (insert x s) using notin f IH by auto
  finally show card {f p |p. p ∈ insert x s} = card (insert x s) .
qed

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

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


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

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

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

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

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

by (*simp add*: *cons-inject finite-inj-comp-set finite-pos*)
moreover have ... $\geq$ *card* (*vars-of-prop* $\varphi 1$) + *card* (*vars-of-prop* $\varphi 2$) **using** *IH1 IH2* **by** *arith*
then have ... $\geq$ *card* (*vars-of-prop* $\varphi 1$ $\cup$ *vars-of-prop* $\varphi 2$) **using** *card-Un-le le-trans* **by** *blast*
ultimately
show *card* (*vars-of-prop* $\varphi 1$ $\cup$ *vars-of-prop* $\varphi 2$) $\leq$ *Suc* (*card* (*?L* $\cup$ *?R*))
card (*vars-of-prop* $\varphi 1$ $\cup$ *vars-of-prop* $\varphi 2$) $\leq$ *Suc* (*card* (*?L* $\cup$ *?R*))
card (*vars-of-prop* $\varphi 1$ $\cup$ *vars-of-prop* $\varphi 2$) $\leq$ *Suc* (*card* (*?L* $\cup$ *?R*))
card (*vars-of-prop* $\varphi 1$ $\cup$ *vars-of-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* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **where**
path-to-refl[*intro*]: *path-to* [] φ φ |
path-to-l: *c*$\in$*binary-connectives* $\vee$ *c* = *CNot* $\Longrightarrow$ *wf-conn c* (φ#*l*) $\Longrightarrow$ *path-to p* φ $\varphi'$$\Longrightarrow$
path-to (*L*#*p*) (*conn c* (φ#*l*)) φ' |
path-to-r: *c*$\in$*binary-connectives* $\Longrightarrow$ *wf-conn c* (ψ#φ#[]) $\Longrightarrow$ *path-to p* φ φ' $\Longrightarrow$
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' $\Longrightarrow$ $\varphi' \preceq \varphi$
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 $\varphi' \preceq \varphi$ $\Longrightarrow$ $\exists$ *p*. *path-to p* φ φ'
proof (*induct rule*: *subformula.induct*)
case *subformula-refl*
have *path-to* [] φ' φ' **by** *auto*
then show $\exists$ *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* $\vee$ *c* = *CF* $\vee$ *c* = *CVar x*
then have *False* **using** *subformula-into-subformula* **by** *auto*
then have $\exists$ *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 $\exists$ *p*. *path-to p* (*conn c l*) φ' **by** *blast*
}
moreover {
assume *c*: *c*$\in$ *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 $a = \psi \lor b = \psi$ **using** ψ **by** *auto*
then have *path-to* $(L \# p)$ $(conn\ c\ l)$ $\varphi' \lor$ *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 $\exists p.$ *path-to* p $(conn\ c\ l)$ φ' **by** *blast*
}
ultimately show $\exists p.$ *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 $(L \# l)$ $(FAnd\ \varphi\ \varphi')$ ψ = *FAnd* (*replace-at* $l\ \varphi\ \psi$) φ'|
replace-at $(R \# l)$ $(FAnd\ \varphi\ \varphi')$ ψ = *FAnd* φ (*replace-at* $l\ \varphi'\ \psi$) |
replace-at $(L \# l)$ $(FOr\ \varphi\ \varphi')$ ψ = *FOr* (*replace-at* $l\ \varphi\ \psi$) φ' |
replace-at $(R \# l)$ $(FOr\ \varphi\ \varphi')$ ψ = *FOr* φ (*replace-at* $l\ \varphi'\ \psi$) |
replace-at $(L \# l)$ $(FEq\ \varphi\ \varphi')$ ψ = *FEq* (*replace-at* $l\ \varphi\ \psi$) φ'|
replace-at $(R \# l)$ $(FEq\ \varphi\ \varphi')$ ψ = *FEq* φ (*replace-at* $l\ \varphi'\ \psi$) |
replace-at $(L \# l)$ $(FImp\ \varphi\ \varphi')$ ψ = *FImp* (*replace-at* $l\ \varphi\ \psi$) φ'|
replace-at $(R \# l)$ $(FImp\ \varphi\ \varphi')$ ψ = *FImp* φ (*replace-at* $l\ \varphi'\ \psi$) |
replace-at $(L \# l)$ $(FNot\ \varphi)$ ψ = *FNot* (*replace-at* $l\ \varphi\ \psi$)

5 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**
$\mathcal{A} \models FT = True$ |
$\mathcal{A} \models FF = False$ |
$\mathcal{A} \models FVar\ v = (\mathcal{A}\ v)$ |
$\mathcal{A} \models FNot\ \varphi = (\neg(\mathcal{A} \models \varphi))$ |
$\mathcal{A} \models FAnd\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \land \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FOr\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \lor \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FImp\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FEq\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \longleftrightarrow \mathcal{A} \models \varphi_2)$

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.

6 Rewrite systems and properties

6.1 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-rew-step* $r\ \varphi\ \psi$ |
propo-rew-one-step-lift: *propo-rew-step* $r\ \varphi\ \varphi' \Longrightarrow$ *wf-conn* $c\ (\psi s\ @\ \varphi\ \#\ \psi s')$
$\Longrightarrow$ *propo-rew-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-rew-step r* φ $\varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$
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 ψ ψ' φ :: *'v propo*
shows $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists \varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step r* φ $\varphi')$
proof (*induct* φ *rule*: *subformula.induct*)
case *subformula-refl*
hence *propo-rew-step r* ψ ψ' **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* ψ φ' **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* ψ'' φ' **by** *metis*
moreover obtain ξ ξ' :: *'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* (ξ @ φ' # ξ'))
using *propo-rew-step.intros*(*2*) *wf* **by** *metis*
moreover have $\psi' \preceq$ *conn c* (ξ @ φ' # ξ')
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* φ $\varphi')$
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*$) = (A \models$ *conn c l'*$)$
proof (*cases c rule*: *connective-cases-arity-2*)
case *nullary*
thus $(A \models$ *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*)
thus $A \models$ *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* φ φ' means that we rewrite ψ inside φ (ie at a path p) into ψ'.

lemma *propo-rew-step-rewrite*:
  fixes φ φ' :: *'v propo* **and** r :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool*
  assumes *propo-rew-step r* φ φ'
  shows $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \land$ *path-to p* φ ψ $\land$ *replace-at p* φ ψ' $= \varphi'$
  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* [] φ $\psi = \psi$ **by** *auto*
  ultimately show *?case* **by** *metis*
next
  case (*propo-rew-one-step-lift* φ φ' c ξ ξ') **note** *rel* = *this*(*1*) **and** *IH0* = *this*(*2*) **and** *corr* = *this*(*3*)
  obtain ψ ψ' p **where** *IH*: $r\ \psi\ \psi' \land$ *path-to p* φ ψ $\land$ *replace-at p* φ $\psi' = \varphi'$ **using** *IH0* **by** *metis*

  {
    fix x :: *'v*
    assume c = *CT* $\lor$ c = *CF* $\lor$ c = *CVar x*
    hence *False* **using** *corr* **by** *auto*
    hence $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \land$ *path-to p* (*conn c* (ξ@ (φ # ξ'))) ψ
        $\land$ *replace-at p* (*conn c* (ξ@ (φ # ξ'))) ψ' = *conn c* (ξ@ (φ' # ξ'))
      by *fast*
  }
  moreover {
    assume c: c = *CNot*
    hence *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 $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \land$ *path-to p* (*conn c* (ξ@ (φ # ξ'))) ψ
        $\land$ *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*
    hence *length* ξ + *length* ξ' = *1* **by** *auto*
    hence *ld*: (*length* ξ = *1* $\land$ *length* ξ' = *0*) $\lor$ (*length* ξ = *0* $\land$ *length* ξ' = *1*) **by** *arith*
    obtain a b **where** *ab*: (ξ=[] $\land$ ξ'=[b]) $\lor$ (ξ=[a] $\land$ ξ'=[])
      using *ld* **by** (*case-tac* ξ, *case-tac* ξ', *auto*)
    {
      assume φ: ξ=[] $\land$ 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 $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \land$ *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'=[]
hence *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

6.2 Consistency preservation

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

definition *preserves-un-sat* **where**
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*:
propo-rew-step r $\varphi\ \psi \Longrightarrow$ *preserves-un-sat r* $\Longrightarrow$ *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*
thus *?case* **by** *simp*
next
case (*propo-rew-one-step-lift* φ φ' *c* ξ ξ') **note** *rel* = *this*(*1*) **and** *wf* = *this*(*2*)
and *IH* = *this*(*3*)[*OF this*(*4*) *this*(*1*)] **and** *consistent* = *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*)
hence ($A \models$ *conn c* (ξ @ φ # ξ')) = ($A \models$ *conn c* (ξ @ φ' # ξ'))
by (*meson consistency-decompose-into-list wf wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
}
thus $\forall\,A.\ A \models$ *conn c* (ξ @ φ # ξ') $\longleftrightarrow A \models$ *conn c* (ξ @ φ' # ξ') **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*]:
preserves-un-sat f $\Longrightarrow$ *preserves-un-sat g* $\Longrightarrow$ *preserves-un-sat* (*f OO g*)
unfolding *preserves-un-sat-def* **by** *auto*

lemma *star-consistency-preservation-explicit*:
  assumes $(propo\text{-}rew\text{-}step\ r)^{\frown}{**}\ \varphi\ \psi$ **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*:
preserves-un-sat r $\Longrightarrow$ *preserves-un-sat* $(propo\text{-}rew\text{-}step\ r)^{\frown}{**}$
  by (*simp add*: *star-consistency-preservation-explicit preserves-un-sat-def*)

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

7 Transformation testing

7.1 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 $\varphi \Longrightarrow$ *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*)
thus $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
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)
hence $\neg$ *all-subformula-st test-symb* φ
using *all-subformula-st-decomp-explicit*(*3*) *nst subf* **by** *blast*
hence $\exists \psi\ \psi'.\ \psi \preceq$ *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 $(binary\ \varphi\ \varphi 1\ \varphi 2)$
note $IH\varphi 1\text{-}0 = this(1)[OF\ this(4)]$ **and** $IH\varphi 2\text{-}0 = this(2)[OF\ this(4)]$ **and** $r = this(4)$
and $\varphi = this(3)$ **and** $le = this(5)$ **and** $nst = this(6)$

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

hence *corr*: $wf\text{-}conn\ c\ [\varphi 1, \varphi 2]$ **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 $r\ IH\varphi 1\text{-}0$ **have** $IH\varphi 1$: $\neg\ all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi 1 \Longrightarrow \exists\psi\ \psi'.\ \psi \preceq \varphi 1 \wedge r\ \psi\ \psi'$
using $inc(1)$ *subformula-trans le* **by** *blast*
from $r\ IH\varphi 2\text{-}0$ **have** $IH\varphi 2$: $\neg\ all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi 2 \Longrightarrow \exists\psi.\ \psi \preceq \varphi 2 \wedge (\exists\psi'.\ r\ \psi\ \psi')$
using $inc(2)$ *subformula-trans le* **by** *blast*
have *cases*: $\neg test\text{-}symb\ \varphi \vee \neg all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi 1 \vee \neg all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi 2$
using *c nst* **by** *auto*
show $\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
using $IH\varphi 1\ IH\varphi 2$ *subformula-trans inc subformula-refl cases le* **by** *blast*
qed

7.2 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 $\forall\varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi' \longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \psi$ 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: $\forall c\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \longrightarrow wf\text{-}conn\ c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow test\text{-}symb\ \varphi' \longrightarrow test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$

7.2.1 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 $r :: {}'v\ propo \Rightarrow {}'v\ propo \Rightarrow bool$ **and** $test\text{-}symb :: {}'v\ propo \Rightarrow bool$ **and** $x :: {}'v$
and $\varphi\ \psi\ \Phi :: {}'v\ propo$
assumes H: $\forall\varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi'$
$\longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ \psi$
and H': $\forall(c :: {}'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi'$
$\longrightarrow wf\text{-}conn\ c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow test\text{-}symb\ \varphi'$
$\longrightarrow test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$ **and**
$propo\text{-}rew\text{-}step\ r\ \varphi\ \psi$ **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*
thus *?case* **using** *H* **by** *simp*
next
case (*propo-rew-one-step-lift* φ φ' *c* ξ ξ')
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*
hence $*$: $\forall\psi.\ \psi \in$ *set* $(\xi\ @\ \varphi'\ \#\ \xi') \longrightarrow$ *all-subformula-st test-symb* ψ **using** φ *sq* **by** *fastforce*
hence *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*)
thus *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 φ ψ :: $'v$ *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 φ ψ **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* φ ψ] *assms subformula-refl* **by** *metis*

The lemmas can be lifted to *full* (*propo-rew-step r*) instead of *propo-rew-step*

7.2.2 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 φ ψ :: $'v$ *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* φ φ'
$\longrightarrow$ *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*) φ ψ **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*)** φ ψ
using *full* **unfolding** *full-def* **by** *auto*
thus *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* ⇒ ′*v propo* ⇒ *bool* **and** *test-symb*:: ′*v propo* ⇒ *bool* **and** *x* :: ′*v*
and φ ψ :: ′*v propo*
assumes
H: ∀ φ ψ. *propo-rew-step r* φ ψ ⟶ *all-subformula-st test-symb* φ
⟶ *all-subformula-st test-symb* ψ **and**
H′: ∀ (*c*:: ′*v connective*) ξ φ ξ' φ'. *propo-rew-step r* φ φ' ⟶ *wf-conn c* (ξ @ φ # ξ')
⟶ *test-symb* (*conn c* (ξ @ φ # ξ')) ⟶ *test-symb* φ' ⟶ *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* ⇒ ′*v propo* ⇒ *bool* **and** *test-symb*:: ′*v propo* ⇒ *bool* **and** *x* :: ′*v*
and φ ψ :: ′*v propo*
assumes
H: ∀ φ ψ. *r* φ ψ ⟶ *all-subformula-st test-symb* φ ⟶ *all-subformula-st test-symb* ψ **and**
H′: ∀ (*c*:: ′*v connective*) ξ φ ξ' φ'. *wf-conn c* (ξ @ φ # ξ') ⟶ *test-symb* (*conn c* (ξ @ φ # ξ'))
⟶ *test-symb* φ' ⟶ *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*
thus *all-subformula-st test-symb* ψ
using *init*
proof (*induct rule*: *rtranclp-induct*)
case *base*
thus *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*)
hence *all-subformula-st test-symb b* **by** *metis*
thus *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: $\forall \varphi\ \psi.\ r\ \varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi \longrightarrow$ *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-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'))
    using H' **by** (*metis wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
  thus *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.

8 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.

8.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

inductive *elim-equiv* :: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *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* φ ψ) (*FImp* ψ φ)
  by *auto*

lemma *elim-equiv-explicit*: *elim-equiv* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
  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*) ⟷ *c* ≠ *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
¬*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) ⟷ *no-equiv* φ
no-equiv (*FAnd* φ ψ) ⟷ (*no-equiv* φ ∧ *no-equiv* ψ)
no-equiv (*FOr* φ ψ) ⟷ (*no-equiv* φ ∧ *no-equiv* ψ)
no-equiv (*FImp* φ ψ) ⟷ (*no-equiv* φ ∧ *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*: ¬ *no-equiv* φ
shows ∃ ψ ψ'. $\psi \preceq \varphi$ ∧ *elim-equiv* ψ ψ'
proof −
have *test-symb-false-nullary*:
∀ *x*::*'v*. *no-equiv-symb FF* ∧ *no-equiv-symb FT* ∧ *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 ⋀*p pa*. ¬ *elim-equiv* (*p*::*'v propo*) *pa* ∨ ¬ *no-equiv-symb p*
using *elim-equiv.cases no-equiv-symb.simps*(*1*) **by** *blast*
then have *elim-equiv* (*conn c l*) ψ ⟹ ¬*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*

8.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **where**
[*simp*]: *elim-imp* (*FImp* $\varphi\ \psi$) (*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* $\varphi\ \psi$ *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* $\equiv$ *all-subformula-st no-imp-symb*
declare *no-imp-def*[*simp*]

lemma *no-imp-Imp*[*simp*]:
$\neg$*no-imp* (*FImp* $\varphi\ \psi$)
no-imp FT
no-imp FF
unfolding *no-imp-def* **by** *auto*

lemma *all-subformula-st-decomp-explicit-imp*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*

shows
no-imp $(FNot\ \varphi) \longleftrightarrow$ *no-imp* φ
no-imp $(FAnd\ \varphi\ \psi) \longleftrightarrow$ (*no-imp* $\varphi \wedge$ *no-imp* ψ)
no-imp $(FOr\ \varphi\ \psi) \longleftrightarrow$ (*no-imp* $\varphi \wedge$ *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* $\varphi\ \psi$ *rule*: *elim-imp.induct*, *auto*)

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

8.3 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 φ ψ:: *'b propo*
    have *elimTB* φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$ **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 ψs $\Longrightarrow$
    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* $\varphi \Longrightarrow$ *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 $\varphi\ \psi :: 'v\ propo$
  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 $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
¬ *no-T-F-symb-except-toplevel* (*FAnd* φ ψ)
¬ *no-T-F-symb-except-toplevel* (*FOr* φ ψ)
¬ *no-T-F-symb-except-toplevel* (*FImp* φ ψ)
¬ *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 $\varphi = FT \lor \varphi = FF$
shows
¬ *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 ≡ *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 ≡ *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 $FT \in set\ l \lor FF \in set\ l$
shows ¬*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 $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
¬*no-T-F-except-top-level* (*FAnd* φ ψ)
¬*no-T-F-except-top-level* (*FOr* φ ψ)
¬*no-T-F-except-top-level* (*FEq* φ ψ)
¬*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 $\varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee$ *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* $[\varphi, \psi]$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
proof −
have *wf*: *wf-conn c* $[\varphi, \psi]$ **using** *c* **by** *auto*
then have *no-T-F* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ (*no-T-F-symb* (*conn c* $[\varphi, \psi]$) $\wedge$ *no-T-F* $\varphi \wedge$ *no-T-F* ψ)
by (*simp add*: *all-subformula-st-decomp no-T-F-def*)
then show *no-T-F* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *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* $[\varphi, \psi]$) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *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* $\varphi \wedge$ *no-T-F* ψ)
no-T-F (*FOr* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
no-T-F (*FEq* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
no-T-F (*FImp* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *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 ψ = *FF* $\vee$ ψ = *FT* **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*
then show *?case* **using** *ElimTB5 ElimTB6* **by** *blast*
next
case (*binary* φ' $\psi 1$ $\psi 2$)
note *IH1* = *this*(*1*) **and** *IH2* = *this*(*2*) **and** φ' = *this*(*3*) **and** $F\varphi$ = *this*(*4*) **and** *n* = *this*(*5*)
{
assume φ' = *FImp* $\psi 1$ $\psi 2$ $\vee$ φ' = *FEq* $\psi 1$ $\psi 2$
then have *False* **using** *n* $F\varphi$ *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* $\psi 1$ $\psi 2$ $\vee$ φ' = *FOr* $\psi 1$ $\psi 2$
then have $\psi 1$ = *FT* $\vee$ $\psi 2$ = *FT* $\vee$ $\psi 1$ = *FF* $\vee$ $\psi 2$ = *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*: $\neg$ *no-T-F-except-top-level* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists\psi$ ψ'. $\psi \preceq \varphi$ $\wedge$ *elimTB* 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*: *elimTB* (*conn c l*) $\psi \Longrightarrow \neg$*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 ⟹ no-imp φ ⟹ 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
```

8.4 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 ⊨ FNot (FAnd φ ψ) ⟷ A ⊨ (FOr (FNot φ) (FNot ψ))
A ⊨ FNot (FOr φ ψ)  ⟷ A ⊨ (FAnd (FNot φ) (FNot ψ))
A ⊨ FNot (FNot φ)   ⟷ A ⊨ φ
```

by *auto*

lemma *pushNeg-explicit*: *pushNeg* φ ψ $\Longrightarrow$ $\forall$ *A*. *A* $\models$ φ $\longleftrightarrow$ *A* $\models$ ψ
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 φ $\longleftrightarrow$ (φ = *FT* $\vee$ φ = *FF* $\vee$ ($\exists$ *x*. φ = *FVar x*))
by (*cases* φ) *auto*

lemma *subformula-conn-decomp-simple*:
fixes φ ψ :: *'v propo*
assumes *s*: *simple* ψ
shows $\varphi \preceq$ *FNot* ψ $\longleftrightarrow$ (φ = *FNot* ψ $\vee$ φ = ψ)
proof −
have $\varphi \preceq$ *conn CNot* [ψ] $\longleftrightarrow$ (φ = *conn CNot* [ψ] $\vee$ ($\exists$ $\psi\in$ *set* [ψ]. $\varphi \preceq \psi$))
using *subformula-conn-decomp wf-conn-helper-facts*(*1*) **by** *metis*
then show $\varphi \preceq$ *FNot* ψ $\longleftrightarrow$ (φ = *FNot* ψ $\vee$ φ = ψ) **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* $\longleftrightarrow$ (φ = *FNot FT* $\vee$ φ = *FT*)
$\varphi \preceq$ *FNot FF* $\longleftrightarrow$ (φ = *FNot FF* $\vee$ φ = *FF*)
$\varphi \preceq$ *FNot* (*FVar x*) $\longleftrightarrow$ (φ = *FNot* (*FVar x*) $\vee$ φ = *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* φ ψ))
$\neg$ *simple-not* (*FNot* (*FOr* φ ψ))
by *auto*

lemma *simple-not-step-exists*:
  fixes φ ψ :: *'v propo*
  assumes *no-equiv* φ **and** *no-imp* φ
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *simple-not-symb* $\psi \Longrightarrow \exists \psi'$. *pushNeg* ψ ψ'
  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* ψ ψ'
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*: $c \neq CF \wedge c \neq CT \wedge \varphi \neq FF \wedge \varphi \neq FT \wedge (\forall \psi \in set\ \xi \cup set\ \xi'.\ \psi \neq FF \wedge \psi \neq FT)$
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* (ξ @ φ' # ξ'). $\psi \neq FF \wedge \psi \neq FT$ **using** *x* **by** *auto*
}
ultimately show *no-T-F-symb* (*conn c* (ξ @ φ' # ξ')) **by** (*simp add*: *x*)
qed

lemma *propo-rew-step-pushNeg-no-T-F*:
propo-rew-step pushNeg φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *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* (ξ @ φ' # ξ')
using *wf-conn-no-arity-change wf-conn-no-arity-change-helper wf* **by** *metis*
ultimately show *no-T-F* (*conn c* (ξ @ φ' # ξ'))
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 $\varphi = FT \lor \varphi = FF$
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 $\varphi \neq FT \land \varphi \neq FF$
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*: $\zeta \preceq \varphi$
and *corr*: *wf-conn c* $(\xi$ @ ζ # $\xi')$
and *no-T-F*: *no-T-F-symb-except-toplevel* (*conn c* $(\xi$ @ ζ # $\xi')$)
and *n*: *no-T-F-symb-except-toplevel* ζ'
have *no-T-F-symb-except-toplevel* (*conn c* $(\xi$ @ ζ' # $\xi')$)
proof
have *p*: *no-T-F-symb* (*conn c* $(\xi$ @ ζ # $\xi')$)
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*: $\forall\varphi\in$*set* $(\xi$ @ ζ # $\xi')$. $\varphi \neq FT \land \varphi \neq FF$
using *corr wf-conn-no-T-F-symb-iff p* **by** *blast*
from *rel incl* **have** $\zeta'\neq FT \land \zeta'\neq FF$
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 $\forall\varphi \in$ *set* $(\xi$ @ ζ' # $\xi')$. $\varphi \neq FT \land \varphi \neq FF$ **using** *l* **by** *auto*
moreover have $c \neq CT \land c \neq CF$ **using** *corr* **by** *auto*
ultimately show *no-T-F-symb* (*conn c* $(\xi$ @ ζ' # $\xi')$)
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 $\Longrightarrow$ *no-equiv* φ $\Longrightarrow$ *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 $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *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*

8.5 Push inside

inductive *push-conn-inside* :: *'v connective* $\Rightarrow$ *'v connective* $\Rightarrow$ *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *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'* [$\varphi 1$, $\varphi 2$], ψ])
(*conn c'* [*conn c* [$\varphi 1$, ψ], *conn c* [$\varphi 2$, 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'* [$\varphi 1$, $\varphi 2$]])
(*conn c'* [*conn c* [ψ, $\varphi 1$], *conn c* [ψ, $\varphi 2$]])

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*]:
$\neg$*propo-rew-step* (*push-conn-inside c c'*) *FT* ψ $\neg$*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 $\Longrightarrow$ φ = *FT* $\vee$ φ = *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' ξ $\Longrightarrow$ ξ = *FF* $\vee$ ξ = *FT* $\vee$ ξ = *FVar x* $\vee$ ξ = *FNot FF* $\vee$ ξ = *FNot FT*
$\vee$ ξ = *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*]:
¬*not-c-in-c'-symb c c' FF*
¬*not-c-in-c'-symb c c' FT*
¬*not-c-in-c'-symb c c'* (*FVar x*)
¬*not-c-in-c'-symb c c'* (*FNot FF*)
¬*not-c-in-c'-symb c c'* (*FNot FT*)
¬*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' ≡ *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' ξ $\Longrightarrow$ *wf-conn c* [φ, ψ] $\Longrightarrow$ ξ = *conn c* [φ, ψ]
$\Longrightarrow$ *not-c-in-c'-symb c c'* (*conn c* [ψ, φ])
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'* [φ'', ψ'] **using** *conn-inj* **by** *auto*
have *wf-conn c* [*conn c'* [φ'', ψ'], φ]

using $H(1)$ *wf-conn-no-arity-change length-Cons* **by** *metis*
then show *not-c-in-c′-symb c c′* (*conn c* [ψ, φ])
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′* [φ', φ''] **using** *conn-inj* **by** *auto*
moreover have *wf-conn c* [ψ', *conn c′* [φ', φ'']]
using $H(1)$ *wf-conn-no-arity-change length-Cons* **by** *metis*
ultimately show *not-c-in-c′-symb c c′* (*conn c* [ψ, φ])
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 [φ, ψ] $\Longrightarrow$ *c-in-c′-symb c c′* (*conn c* [φ, ψ]) $\longleftrightarrow$ *c-in-c′-symb c c′* (*conn c* [ψ, φ])
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* [φ, ψ]
shows *c-in-c′-only c c′* (*conn c* [φ, ψ]) $\longleftrightarrow$ *c-in-c′-only c c′* (*conn c* [ψ, φ]) (**is** *?A* $\longleftrightarrow$ *?B*)
proof −
have *?A* $\longleftrightarrow$ (*c-in-c′-symb c c′* (*conn c* [φ, ψ])
$\wedge$ ($\forall \xi \in$ *set* [φ, ψ]. *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* [ψ, φ])
$\wedge$ ($\forall \xi \in$ *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 ∨ c = COr and c′: c′ = CAnd ∨ c′ = COr
  shows ψ ≼ φ ⟹ ¬ c-in-c′-symb c c′ ψ ⟹ ∃ψ′. push-conn-inside c c′ ψ ψ′
  apply (induct ψ rule: propo-induct-arity)
  apply auto[2]
proof −
  fix ψ1 ψ2 φ′:: ′v propo
  assume IHψ1: ψ1 ≼ φ ⟹ ¬ c-in-c′-symb c c′ ψ1 ⟹ Ex (push-conn-inside c c′ ψ1)
  and IHψ2: ψ1 ≼ φ ⟹ ¬ c-in-c′-symb c c′ ψ1 ⟹ Ex (push-conn-inside c c′ ψ1)
  and φ′: φ′ = FAnd ψ1 ψ2 ∨ φ′ = FOr ψ1 ψ2 ∨ φ′ = FImp ψ1 ψ2 ∨ φ′ = FEq ψ1 ψ2
  and inφ: φ′ ≼ φ and  n0: ¬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 [ψ1, ψ2]
    obtain a b where ψ1 = conn c′ [a, b] ∨ ψ2 = 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′: φ′ ≠ conn c [ψ1, ψ2]
     have ∀φ c ca. ∃φ1 ψ1 ψ2 ψ1′ ψ2′ φ2′. conn (c::′v connective) [φ1, conn ca [ψ1, ψ2]] = φ
           ∨ conn c [conn ca [ψ1′, ψ2′], φ2′] = φ ∨ 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: ¬c-in-c′-only c c′ φ
  and c: c = CAnd ∨ c = COr and c′: c′ = CAnd ∨ c′ = COr
  shows ∃ψ ψ′. ψ ≼ φ ∧ push-conn-inside c c′ ψ ψ′
proof −
  have test-symb-false-nullary:
    ∀x. c-in-c′-symb c c′ (FF:: ′v propo) ∧ c-in-c′-symb c c′ FT
      ∧ 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* $c\ c'\ \psi\ \psi'$
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 $\varphi\ \psi$ :: $'v$ *propo*
shows *propo-rew-step* (*push-conn-inside* $c\ c'$) $\varphi\ \psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case (*global-rel* $\varphi\ \psi$)
then show *no-T-F* ψ
by (*cases rule*: *push-conn-inside.cases*, *auto*)
next
case (*propo-rew-one-step-lift* $\varphi\ \varphi'\ c\ \xi\ \xi'$)
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 ξ = [] ξ' = [] **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 $\Longrightarrow$ $\psi \neq FT \wedge \psi \neq FF$
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 $\Longrightarrow$ *all-subformula-st simple-not-symb* φ $\Longrightarrow$ *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 $\Longrightarrow$ *all-subformula-st simple-not-symb* φ $\Longrightarrow$ *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 $\psi s'$, *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* $\vee$ *c* = *CF* $\vee$ *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 $\Longrightarrow$ *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 *n*φ: *all-subformula-st simple-not-symb* φ $\Longrightarrow$ *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* $\in$ *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*] = (ξ @ φ # ξ')
using *corr c list-length2-decomp wf-conn-bin-list-length* **by** *metis*
then have ξ @ φ' # ξ' = [*a*, φ'] $\vee$ (ξ @ φ' # ξ') = [φ', *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* χ $\vee$ χ $\notin$ *set* (ξ @ φ' # ξ')
using *wf'* **by** (*metis* (*no-types*) φ' *all-subformula-st-decomp calculation insert-iff*
list.set(*2*))
}
then have $\forall\varphi$. φ $\in$ *set* (ξ @ φ' # ξ') $\longrightarrow$ *all-subformula-st simple-not-symb* φ
by (*metis* (*no-types*))

moreover have *simple-not-symb* (*conn ca* (ξ @ φ' # ξ'))
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* $(\xi$ @ φ' # $\xi')$ **using** *c calculation(1)* **by** *auto*
ultimately show *all-subformula-st simple-not-symb* (*conn ca* $(\xi$ @ φ' # $\xi'))$
by (*metis all-subformula-st-decomp-imp*)
qed
}
moreover {
fix *ca* :: *'v connective* **and** ξ ξ' :: *'v propo list* **and** φ φ' :: *'v propo*
have *propo-rew-step* (*push-conn-inside c c'*) φ φ' $\Longrightarrow$ *wf-conn ca* $(\xi$ @ φ # $\xi')$
$\Longrightarrow$ *simple-not-symb* (*conn ca* $(\xi$ @ φ # $\xi'))$ $\Longrightarrow$ *simple-not-symb* φ'
$\Longrightarrow$ *simple-not-symb* (*conn ca* $(\xi$ @ φ' # $\xi'))$
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 φ ψ :: *'v propo*
have *H*: *propo-rew-step* (*push-conn-inside c c'*) φ ψ $\Longrightarrow$ *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'*) φ ψ
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* $\varphi \land \varphi \neq FF \land \varphi \neq FT$
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** ξ ξ' :: *'v propo list* **and** φ φ' :: *'v propo*
assume *rel*: *propo-rew-step* (*push-conn-inside c c'*) φ φ'
assume *corr*: *wf-conn ca* $(\xi$ @ φ # $\xi')$
then have *c*: $ca \neq CT \land ca \neq CF$ **by** *auto*
assume *no-T-F*: *no-T-F-symb-except-toplevel* (*conn ca* $(\xi$ @ φ # $\xi'))$
have *no-T-F-symb-except-toplevel* (*conn ca* $(\xi$ @ φ' # $\xi'))$
proof
have *c*: $ca \neq CT \land ca \neq CF$ **using** *corr* **by** *auto*
have ζ: $\forall \zeta \in$ *set* $(\xi$ @ φ # $\xi')$. $\zeta \neq FT \land \zeta \neq FF$
using *corr no-T-F no-T-F-symb-except-toplevel-if-is-a-true-false* **by** *blast*
then have $\varphi \neq FT \land \varphi \neq FF$ **by** *auto*
from *rel this* **have** $\varphi' \neq FT \land \varphi' \neq FF$
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 $\forall \zeta \in set\ (\xi\ @\ \varphi'\ \#\ \xi').\ \zeta \neq FT \wedge \zeta \neq FF$ **using** ζ **by** *auto*
moreover have *wf-conn ca* $(\xi\ @\ \varphi'\ \#\ \xi')$
using *corr wf-conn-no-arity-change* **by** (*metis wf-conn-no-arity-change-helper*)
ultimately show *no-T-F-symb* (*conn ca* $(\xi\ @\ \varphi'\ \#\ \xi')$) **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 $\varphi\ \psi$ :: ′*v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step* (*push-conn-inside c c*′)) $\varphi\ \psi$ **and**
no-T-F-except-top-level φ **and**
simple-not φ **and**
$c = CAnd \vee c = COr$ **and**
$c' = CAnd \vee c' = COr$
shows *c-in-c*′*-only c c*′ ψ
using *c-in-c*′*-symb-rew assms full-propo-rew-step-subformula* **by** *blast*

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

inductive *only-c-inside-symb* :: ′*v connective* ⇒ ′*v propo* ⇒ *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*:
$only\text{-}c\text{-}inside\text{-}symb\ c\ \psi \longleftrightarrow (simple\ \psi$
$\quad\vee (\exists\ \varphi'.\ \psi = FNot\ \varphi' \wedge simple\ \varphi')$
$\quad\vee (\exists\, l.\ \psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l))$
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*:
$only\text{-}c\text{-}inside\ c\ \varphi \longleftrightarrow$
$(\forall\,\psi.\ \psi \preceq \varphi \longrightarrow (simple\ \psi \vee (\exists\ \varphi'.\ \psi = FNot\ \varphi' \wedge simple\ \varphi')$
$\quad\vee (\exists\, l.\ \psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l)))$
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'*: $c \neq c'$ **and** *c*: $c = CAnd \vee c = COr$ **and** *c'*: $c' = CAnd \vee c' = COr$
and *only*: *only-c-inside c* φ **and** *incl*: $conn\ c'\ l \preceq \varphi$ **and** *wf*: *wf-conn c' l*
shows *False*
proof −
let $?\psi = conn\ c'\ l$
have $simple\ ?\psi \vee (\exists\ \varphi'.\ ?\psi = FNot\ \varphi' \wedge simple\ \varphi') \vee (\exists\, l.\ ?\psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l)$
using *only-c-inside-decomp only incl* **by** *blast*
moreover have $\neg\ simple\ ?\psi$
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 $?\psi \neq FNot\ \varphi'$ **using** *c' conn-inj-not*(*1*) *wf* **by** *blast*
}
ultimately obtain *l* :: *'v propo list* **where** $?\psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l$ **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: $c \neq c'$ **and** c: $c = CAnd \vee c = COr$ **and** c': $c' = CAnd \vee c' = COr$
  shows *only-c-inside c* $\varphi \Longrightarrow$ *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* $\Longrightarrow$ *ca* $\neq$ *c* $\Longrightarrow$ *c-in-c′-symb c c′* (*conn ca l*)
proof −
  have *not-c-in-c′-symb c c′* (*conn ca l*) $\Longrightarrow$ *wf-conn ca l* $\Longrightarrow$ *ca* = *c*
    by (*induct conn ca l rule*: *not-c-in-c′-symb.induct*, *auto simp*: *conn-inj*)
  then show *wf-conn ca l* $\Longrightarrow$ *ca* $\neq$ *c* $\Longrightarrow$ *c-in-c′-symb c c′* (*conn ca l*) **by** *blast*
qed

lemma *only-c-inside-implies-c-in-c′-only*:
  assumes δ: $c \neq c'$ **and** c: $c = CAnd \vee c = COr$ **and** c': $c' = CAnd \vee c' = COr$
  shows *only-c-inside c* $\varphi \Longrightarrow$ *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: $c = CAnd \vee c = COr$ $c' = CAnd \vee c' = COr$ $c \neq c'$ **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* $\Longrightarrow$ *c-in-c′-only c c′* (*conn c l*) $\Longrightarrow$ ($\forall \psi \in$ *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** φ = *this(3)* **and** *only* = *this(5)* **and** *wf* = *this(4)*
    and *no-equiv* = *this(6)* **and** *no-imp* = *this(7)* **and** *simple-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* $\varphi 1$
    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-inside*$\varphi 1$: *only-c-inside c* $\varphi 1$ **by** *metis*

have *c-in-only*$\varphi 2$: *c-in-c′-only c c′* (*conn c2 l2*)
  using *only l* $\varphi 2$ *wf*$\varphi 2$ *assms* **unfolding** *c-in-c′-only-def* **by** *auto*
have *c2c*: $c2 \neq c'$
  proof
    assume *c2c*: $c2 = c'$
    then obtain $\xi 1$ $\xi 2$ **where** *l2*: $l2 = [\xi 1, \xi 2]$
     by (*metis assms*(2) *wf*$\varphi 2$ *wf-conn.simps connective.distinct*(7,9,19,21,29,31,37,39))
    then have *c-in-c′-symb c c′* (*conn c* [$\varphi 1$, *conn c′ l2*])
      using *c2c l only* $\varphi 2$ *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* [$\varphi 1$, *conn c′ l2*])
      using *assms*(1) *c2c l2 not-c-in-c′-symb-r wf*$\varphi 2$ *wf-conn-helper-facts*(5,6) **by** *metis*
    ultimately show *False* **by** *auto*
  qed
then have ($\varphi 2 = conn\ c\ l2 \wedge wf\text{-}conn\ c\ l2$) $\vee$ ($\exists \psi 2.\ \varphi 2 = FNot\ \psi 2$) $\vee$ *simple* $\varphi 2$
  using *c2-eq* **by** (*metis* $\varphi 2$ *assms*(1−3) *c2-eq c2-imp simple.elims*(3) *wf*$\varphi 2$ *wf-conn-list*(4−7))

moreover {
  assume $\varphi 2 = conn\ c\ l2 \wedge wf\text{-}conn\ c\ l2$
  then have *only-c-inside* $c\ \varphi 2$
    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 $\exists \psi 2.\ \varphi 2 = FNot\ \psi 2$
  then obtain $\psi 2$ **where** $\varphi 2 = FNot\ \psi 2$ **by** *metis*
  then have *only-c-inside* $c\ \varphi 2$
    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* $\varphi 2$
  then have *only-c-inside* $c\ \varphi 2$
    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* $c\ \varphi 2$ **by** *metis*
show *?case* **using** *l only-c-insideφ1 only-c-insideφ2* **by** *auto*
qed

8.5.2 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 $\varphi\ \psi$ :: *′v propo*
  assumes *full* (*propo-rew-step pushConj*) $\varphi\ \psi$
  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 $\varphi\ \psi$ :: *′v propo*
  assumes
    no-equiv φ **and**
    no-imp φ **and**
    full (*propo-rew-step pushConj*) $\varphi\ \psi$ **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*))

8.5.3 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*))

9 The full transformations

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

9.1.1 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* φ ⟹ *grouped-by c* φ |
simple-not-is-grouped[*simp*]: *simple* φ ⟹ *grouped-by c* (*FNot* φ) |
connected-is-group[*simp*]: *grouped-by c* φ ⟹ *grouped-by c* ψ ⟹ *wf-conn c* [φ, ψ]
  ⟹ *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'* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \vee c' = COr$ $\Longrightarrow$ *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'* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \vee c' = COr$ $\Longrightarrow$ *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'*: $c' = CAnd \vee c' = COr$
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* [$\varphi 1$, $\varphi 2$] **and** *wf-conn c* [$\varphi 1$, $\varphi 2$] **using** φ*-c''* *wf* l'' **by** *auto*
qed
have *grouped-by c* $\varphi 1$ **using** *wf IH*$\varphi 1$ *IH*$\varphi 2$ φ*-conn only* φ **unfolding** *only-c-inside-def* **by** *auto*
moreover have *grouped-by c* $\varphi 2$
using *wf* φ *IH*$\varphi 1$ *IH*$\varphi 2$ 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* $\Rightarrow$ $'a$ *connective* $\Rightarrow$ $'a$ *propo* $\Rightarrow$ *bool* **for** $c\ c'$ **where**
grouped-is-super-grouped[*simp*]: *grouped-by* $c\ \varphi \Longrightarrow$ *super-grouped-by* $c\ c'\ \varphi$ |
connected-is-super-group: *super-grouped-by* $c\ c'\ \varphi \Longrightarrow$ *super-grouped-by* $c\ c'\ \psi \Longrightarrow$ *wf-conn* c $[\varphi, \psi]$
$\Longrightarrow$ *super-grouped-by* $c\ c'$ (*conn* c' $[\varphi, \psi]$)

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* $\vee$ c = *COr* **and** c': c' = *CAnd* $\vee$ c' = *COr* **and** cc': $c \neq c'$
shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *c-in-c'-only* $c\ c'\ \varphi$
$\Longrightarrow$ *super-grouped-by* $c\ c'\ \varphi$
(**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* $\vee$ φ = *FF* $\vee$ ($\exists$ x. φ = *FVar* x) **by** (*cases* φ, *auto*)
then show *?S* (*FNot* φ) **by** *auto*
next
case (*binary* φ $\varphi 1$ $\varphi 2$)
note *IH*$\varphi 1$ = *this*(*1*) **and** *IH*$\varphi 2$ = *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* $\varphi 1$ $\varphi 2$ $\vee$ φ = *FEq* $\varphi 1$ $\varphi 2$
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* $c\ c'\ \varphi$
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

9.2 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* $\varphi \wedge$ *no-T-F-except-top-level* φ

9.2.1 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 $\varphi' \Longrightarrow$ *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* φ*Neg no-imp* φ*Neg no-T-F-except-top-level* φ*Neg*
using *pushNeg-inv Neg noTB noTB-inv* **by** *blast+*

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

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

9.3 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* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *and-in-or-only* $\varphi \Longrightarrow$ *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 $\varphi \longleftrightarrow$ *is-disj-with-TF* $\varphi \wedge$ *no-T-F-except-top-level* φ

9.3.1 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 $\varphi' \Longrightarrow$ *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*))

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

10.1 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-equiv* $?\varphi$; *no-imp* $?\varphi$; $?\psi \preceq ?\varphi$; $\neg$ *no-T-F-symb-except-toplevel* $?\psi]\!] \Longrightarrow \exists \psi'.$ *elimTB* $?\psi$ ψ', 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 $\psi = FF \lor \psi = FT$ **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 $\psi 1 = FT \lor \psi 2 = FT \lor \psi 1 = FF \lor \psi 2 = FF$
    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 ¬ *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*: ¬ *no-T-F-except-top-level* φ
  shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land$ *elimTBFull* $\psi\ \psi'$
proof −
  have *test-symb-false-nullary*:
    $\forall x.$ *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*: *elimTBFull* (*conn c l*) $\psi \Longrightarrow$ ¬*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 $\varphi\ \psi$ :: *'v propo*
  assumes *full* (*propo-rew-step elimTBFull*) $\varphi\ \psi$
  shows *no-T-F-except-top-level* ψ
  using *full-propo-rew-step-subformula no-T-F-except-top-level-rew' assms* **by** *fastforce*

10.2 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* $\varphi\ \psi \Longrightarrow$ *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* $\varphi'\ \psi'$
  have $\forall x0\ x1.$ (¬ *elim-equiv* (*x1* :: *'v propo*) *x0* ∨ (∃ *v2 v3 v4 v5 v6 v7*. *x1* = *FEq v2 v3*
    ∧ *x0* = *FAnd* (*FImp v4 v5*) (*FImp v6 v7*) ∧ *v2* = *v4* ∧ *v4* = *v7* ∧ *v3* = *v5* ∧ *v3* = *v6*))
      = (¬ *elim-equiv x1 x0* ∨ (∃ *v2 v3 v4 v5 v6 v7*. *x1* = *FEq v2 v3*

∧ *x0* = *FAnd* (*FImp v4 v5*) (*FImp v6 v7*) ∧ *v2* = *v4* ∧ *v4* = *v7* ∧ *v3* = *v5* ∧ *v3* = *v6*))
by *meson*
then have ∀ *p pa*. ¬ *elim-equiv* (*p* :: ′*v propo*) *pa* ∨ (∃ *pb pc pd pe pf pg*. *p* = *FEq pb pc*
∧ *pa* = *FAnd* (*FImp pd pe*) (*FImp pf pg*) ∧ *pb* = *pd* ∧ *pd* = *pg* ∧ *pc* = *pe* ∧ *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 φ* ⟹ *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* ∈ *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* ∨ *φ* = *b*
by (*metis append.simps*(*1*) *append-is-Nil-conv list.distinct*(*1*) *list.sel*(*3*) *nth-Cons-0*
tl-append2)
have *ζ*: ∀ *ζ*∈ *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*] ∨ *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 ∀ *ζ* ∈ *set* (*ξ* @ *φ* # *ξ*′). *ζ* ≠ *FT* ∧ *ζ* ≠ *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 $\Longrightarrow$ *no-T-F-except-top-level* φ
$\Longrightarrow$ *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 $\varphi = FT \vee \varphi = FF$
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 $\varphi \neq FT \wedge \varphi \neq FF$
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-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*: $\zeta \preceq \varphi$
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*: $\forall\varphi\in$*set* (ξ @ ζ # ξ'). $\varphi \neq FT \wedge \varphi \neq FF$
using *corr wf-conn-no-T-F-symb-iff p* **by** *blast*
from *rel incl* **have** $\zeta'\neq FT \wedge\zeta'\neq FF$
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 $\forall\varphi \in$ *set* (ξ @ ζ' # ξ'). $\varphi \neq FT \wedge \varphi \neq FF$ **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'))
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 $\Longrightarrow$ *no-T-F* φ $\Longrightarrow$ *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* $\in$ *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* $\vee$ 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: $\forall$ ζ $\in$ *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*] $\vee$ 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 *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* $\vee$ *c* = *CF* $\vee$ *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
  shows*no-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 ∀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'))
    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

10.3 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' $\Longrightarrow$ *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' $\Longrightarrow$ *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

11 Partial Clausal Logic

theory *Partial-Clausal-Logic*

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

11.1 Clauses

Clauses are (finite) multisets of literals.

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

11.2 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*]

11.2.1 Consistency

definition *consistent-interp* :: $'a$ *literal set* $\Rightarrow$ *bool* **where**
consistent-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*

11.2.2 Atoms

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* ⇒ *'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* ⊆ *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* ψs ∪ *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* ψs ∪ *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* ⊆ *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) ⊆ *atms-of-ms* ψ
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-remove-subset*:
atms-of-ms ($\varphi - \psi$) ⊆ *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 = {} ⟷ *A* = {{#}} ∨ *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* ∈# *C* **and** *C* ∈ *N*
  shows *atm-of L* ∈ *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*#} ∈ *N*
  shows *atm-of L* ∈ *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* ∈ *ψs*
  shows *atm-of A* ∈ *atms-of-ms ψs*
  using *assms* **by** (*auto dest*: *atms-of-atms-of-ms-mono*)

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

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

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

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

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

11.2.3 Totality

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

definition *total-over-m* :: ′*a literal set* ⇒ ′*a clause set* ⇒ *bool* **where**

total-over-m I ψs = total-over-set I (atms-of-ms ψs)

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

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

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

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

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

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

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

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

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

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

lemma *total-over-m-consistent-extension*:
fixes *I* :: *'v literal set* **and** *A* :: *'v 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 $\{\#A\#\} + D \in \psi s$
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: $\neg\ L \in\#\ \psi\ -L \notin\#\ \psi$
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 $\neg$ *?thesis*
then have *atm-of* $L \in$ *atms-of* ψ **by** *auto*
then have *Pos* (*atm-of* L) $\in\#\ \psi \vee$ *Neg* (*atm-of* L) $\in\#\ \psi$
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*

11.2.4 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*)

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

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

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

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

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

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

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

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

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

lemma *atms-of-ms-empty*:
  fixes ψ :: *'v clauses*
  assumes *atms-of-ms* ψ = {}
  shows ψ = {} ∨ ψ = {{#}}
  using *assms* **by** (*auto simp add*: *atms-of-ms-def*)

lemma *consistent-interp-disjoint*:
 assumes *consI*: *consistent-interp I*
 and *disj*: *atms-of-s A* ∩ *atms-of-s I* = {}
 and *consA*: *consistent-interp A*
 shows *consistent-interp* (*A* ∪ *I*)
proof (*rule ccontr*)
  assume ¬ *?thesis*
  moreover have ⋀*L*. ¬ (*L* ∈ *A* ∧ −*L* ∈ *I*)
    using *disj* **unfolding** *atms-of-s-def* **by** (*auto simp add*: *rev-image-eqI*)
  ultimately show *False*
    using *consA consI* **unfolding** *consistent-interp-def* **by** (*metis* (*full-types*) *Un-iff*
      literal.exhaust-sel uminus-Neg uminus-Pos)
qed

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

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

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

11.2.7 Tautologies

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

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

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

lemma *tautology-exists-Pos-Neg*:
  assumes *tautology* ψ
  shows ∃ *p*. *Pos p* ∈# ψ ∧ *Neg p* ∈# ψ
proof (*rule ccontr*)

assume p: $\neg$ ($\exists$ *p*. *Pos p* $\in$# ψ $\wedge$ *Neg p* $\in$# ψ)
let *?I* = {$-$*L* | *L*. *L* $\in$# ψ}
have *total-over-set ?I* (*atms-of* ψ)
unfolding *total-over-set-def* **using** *atm-imp-pos-or-neg-lit* **by** *force*
moreover have $\neg$ *?I* $\models$ ψ
unfolding *true-cls-def true-lit-def Bex-def* **apply** *clarify*
using *p* **by** (*rename-tac x L*, *case-tac L*) *fastforce+*
ultimately show *False* **using** *assms* **unfolding** *tautology-def* **by** *auto*
qed

lemma *tautology-decomp*:
tautology ψ $\longleftrightarrow$ ($\exists$ *p*. *Pos p* $\in$# ψ $\wedge$ *Neg p* $\in$# ψ)
using *tautology-exists-Pos-Neg* **by** *auto*

lemma *tautology-false*[*simp*]: $\neg$*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* | *L*. *L*$\in$# χ} $\models$ χ
shows *tautology* χ
proof $-$
obtain *L* **where** *L* $\in$# χ $\wedge$ $-$*L* $\in$# χ
using *assms* **unfolding** *true-cls-def* **by** *auto*
then show *?thesis* **using** *tautology-decomp literal.exhaust uminus-Neg uminus-Pos* **by** *metis*
qed

lemma *remove-literal-in-model-tautology*:
assumes *I* $\cup$ {*Pos P*} $\models$ φ
and *I* $\cup$ {*Neg P*} $\models$ φ
shows *I* $\models$ φ $\vee$ *tautology* φ
using *assms* **unfolding** *true-cls-def* **by** *auto*

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

11.2.8 Entailment for clauses and propositions

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

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

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

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

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

lemma *true-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*]:
$A \models ps\ C \cup D \longleftrightarrow (A \models ps\ C \wedge A \models ps\ D)$
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: $\forall x \in I'.\ atm\text{-}of\ x \in atms\text{-}of\text{-}ms\ D \wedge atm\text{-}of\ x \notin atms\text{-}of\text{-}ms\ (A \cup C)$
using *total-over-m-consistent-extension*[*OF - cons, of* $A \cup C$] *tot tot'* **by** *blast*
moreover have $I \cup I' \models s\ A$ **using** I **by** *simp*
ultimately have $I \cup I' \models s\ C \cup D$ **using** A **unfolding** *true-clss-clss-def* **by** *auto*
then have $I \cup I' \models s\ C \cup D$ **by** *auto*
then show $I \models s\ C$ **using** *notin-vars-union-true-clss-true-clss*[*of* I'] H **by** *auto*
qed
} **note** $H = this$
assume $A \models ps\ C \cup D$
then show $A \models ps\ C \wedge A \models ps\ D$ **using** H[*of* A] *Un-commute*[*of* $C\ D$] **by** *metis*
next
assume $A \models ps\ C \wedge A \models ps\ D$
then show $A \models ps\ C \cup D$

unfolding *true-clss-clss-def* **by** *auto*
qed

lemma *true-clss-clss-insert*[*iff*]:
$A \models ps\ insert\ L\ Ls \longleftrightarrow (A \models p\ L \land A \models ps\ Ls)$
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: $L \in I \lor -L \in I$
  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: $L \notin I \land -L \notin I$
  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 $I \models D + C$ **unfolding** *true-cls-def* **by** *auto*
 }
 ultimately show $I \models D + C$ **by** *blast*
qed

lemma *true-cls-union-mset*[*iff*]: $I \models C\ \#\cup\ D \longleftrightarrow I \models C \lor I \models D$
 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: $L \notin I \wedge -L \notin I$
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 $I \models D\ \ \#\cup\ C$ **unfolding** *true-cls-def* **by** *auto*
}
ultimately show $I \models D\ \#\cup\ C$ **by** *blast*
qed

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

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

11.3 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 $\{C\} \longrightarrow I \models C \longrightarrow I \models \varphi$) $\vee$ *tautology* φ
  using *assms*
proof (*induct card* {*Pos* v | v. $v \in$ *atms-of* $D \wedge v \notin$ *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 |v. $v \in$ *atms-of* $D \wedge v \notin$ *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 $I \models \varphi \vee$ *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* $\{Pos\ v\ |v.\ v \in atms\text{-}of\ ?D' \wedge v \notin atms\text{-}of\ C\} <$
card $\{Pos\ v\ |v.\ v \in atms\text{-}of\ D \wedge v \notin atms\text{-}of\ C\}$
using L **by** (*auto intro*!: *psubset-card-mono*)
then show *?case*
using *IH C-in-D' H'* **unfolding** *card*[*symmetric*] **by** *blast*
qed

11.4 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*

11.5 Set of all Simple Clauses

definition *simple-clss* :: *'v set* $\Rightarrow$ *'v clause set* **where**
simple-clss atms = $\{C.\ atms\text{-}of\ C \subseteq atms \wedge \neg tautology\ C \wedge distinct\text{-}mset\ C\}$

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

lemma *simple-clss-insert*:
assumes $l \notin atms$
shows *simple-clss* (*insert l atms*) =
(*op* + $\{\#Pos\ l\#\}$) ‘ (*simple-clss atms*)
$\cup$ (*op* + $\{\#Neg\ l\#\}$) ‘ (*simple-clss atms*)
$\cup$ *simple-clss atms*(**is** *?I* = *?U*)
proof (*standard*; *standard*)
fix C
assume $C \in ?I$
then have
atms: *atms-of* $C \subseteq$ *insert l atms* **and**
taut: $\neg$*tautology C* **and**
dist: *distinct-mset C*
unfolding *simple-clss-def* **by** *auto*
have H: $\bigwedge x.\ x \in\# C \Longrightarrow atm\text{-}of\ x \in insert\ l\ atms$
using *atm-of-lit-in-atms-of atms* **by** *blast*
consider
(*Add*) L **where** $L \in\# C$ **and** $L = Neg\ l \vee L = Pos\ l$
| (*No*) $Pos\ l \notin\# C$ $Neg\ l \notin\# C$
by *auto*
then show $C \in ?U$
proof *cases*
case *Add*

```
      then have L ∉# C − {#L#}
        using dist unfolding distinct-mset-def by (auto simp: not-in-iff)
      moreover have −L ∉# C
        using taut Add by auto
      ultimately have atms-of (C − {#L#}) ⊆ atms
        using atms Add by (smt H atms-of-def imageE in-diffD insertE literal.exhaust-sel
          subset-iff 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
    $x \in$ *simple-clss atms*
  shows *atms-of* $x \subseteq$ *atms* $\wedge$ $\neg$*tautology* x $\wedge$ *distinct-mset* x
  using *assms* **unfolding** *simple-clss-def* **by** *auto*

lemma *cls-in-simple-clss*:
  shows $\{\#\} \in$ *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*) $\leq$ (*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* ⊆ *atms'*
  shows *simple-clss atms* ⊆ *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** ¬*tautology* χ
  shows χ ∈ *simple-clss* (*atms-of* χ)
  using *assms* **unfolding** *simple-clss-def* **by** *auto*

lemma *simplified-in-simple-clss*:
  assumes *distinct-mset-set* ψ **and** ∀ χ ∈ ψ. ¬*tautology* χ
  shows ψ ⊆ *simple-clss* (*atms-of-ms* ψ)
  using *assms* **unfolding** *simple-clss-def*
  by (*auto simp*: *distinct-mset-set-def atms-of-ms-def*)

11.6 Experiment: Expressing the Entailments as Locales

locale *entail* =
  fixes *entail* :: *'a set* ⇒ *'b* ⇒ *bool* (**infix** ⊨*e 50*)
  assumes *entail-insert*[*simp*]: *I* ≠ {} ⟹ *insert L I* ⊨*e x* ⟷ {*L*} ⊨*e x* ∨ *I* ⊨*e x*
  assumes *entail-union*[*simp*]: *I* ⊨*e A* ⟹ *I* ∪ *I'* ⊨*e A*
begin

definition *entails* :: *'a set* ⇒ *'b set* ⇒ *bool* (**infix** ⊨*es 50*) **where**
  I ⊨*es A* ⟷ (∀ *a* ∈ *A*. *I* ⊨*e a*)

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

lemma *entails-single*[*iff*]:
  I ⊨*es* {*a*} ⟷ *I* ⊨*e a*
  unfolding *entails-def* **by** *auto*

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

lemma *entails-union*[*iff*]: *I* ⊨*es CC* ∪ *DD* ⟷ *I* ⊨*es CC* ∧ *I* ⊨*es DD*
  unfolding *entails-def* **by** *blast*

lemma *entails-insert*[*iff*]: *I* ⊨*es insert C DD* ⟷ *I* ⊨*e C* ∧ *I* ⊨*es DD*
  unfolding *entails-def* **by** *blast*

lemma *entails-insert-mono*: *DD* ⊆ *CC* ⟹ *I* ⊨*es CC* ⟹ *I* ⊨*es DD*
  unfolding *entails-def* **by** *blast*

lemma *entails-union-increase*[*simp*]:
 assumes *I* ⊨*es* ψ

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*)

11.7 Entailment to be extended

definition *true-clss-ext* :: *'a literal set* ⇒ *'a literal multiset set* ⇒ *bool* (**infix** $\models$*sext 49*)
where
$I \models sext\ N \longleftrightarrow (\forall J.\ I \subseteq J \longrightarrow consistent\text{-}interp\ J \longrightarrow total\text{-}over\text{-}m\ J\ N \longrightarrow J \models s\ N)$

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

lemma *true-clss-ext-decrease-right-remove-r*:
assumes $I \models sext\ N$
shows $I \models sext\ N - \{C\}$
unfolding *true-clss-ext-def*
proof (*intro allI impI*)
fix J
assume
$I \subseteq J$ **and**
cons: *consistent-interp* J **and**
tot: *total-over-m* $J\ (N - \{C\})$
let $?J = J \cup \{Pos\ (atm\text{-}of\ P)|P.\ P \in\#\ C \wedge atm\text{-}of\ P \notin atm\text{-}of\ `\ J\}$
have $I \subseteq ?J$ **using** ‹$I \subseteq J$› **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* $a \in N - \{C\}$)
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 $?J \models s\ N$
using *assms* **unfolding** *true-clss-ext-def* **by** *blast*
then have $?J \models s\ N - \{C\}$ **by** *auto*
have $\{v \in ?J.\ atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ (N - \{C\})\} \subseteq J$
using *tot* **unfolding** *total-over-m-def total-over-set-def*
by (*auto intro*!: *rev-image-eqI*)

then show $J \models s\ N - \{C\}$
using *true-clss-remove-unused*$[OF\ \langle ?J \models s\ N - \{C\}\rangle]$ **unfolding** *true-clss-def*
by (*meson true-cls-mono-set-mset-l*)
qed

lemma *consistent-true-clss-ext-satisfiable*:
assumes *consistent-interp* I **and** $I \models sext\ A$
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 $\neg$*consistent-interp* I
shows $I \models sext\ A$
by (*meson assms consistent-interp-subset true-clss-ext-def*)
end
theory *Prop-Logic-Multiset*
imports *../lib/Multiset-More Prop-Normalisation Partial-Clausal-Logic*
begin

12 Link with Multiset Version

12.1 Transformation to Multiset

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

12.2 Equisatisfiability of the two Version

lemma *is-conj-with-TF-FNot*:
is-conj-with-TF (*FNot* φ) $\longleftrightarrow$ ($\exists v.\ \varphi = FVar\ v \lor \varphi = FF \lor \varphi = FT$)
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* φ) $\longleftrightarrow$ ($\exists v.\ \varphi = FVar\ v \lor \varphi = FF \lor \varphi = FT$)
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* φ*1* φ*2*) $\longleftrightarrow$ *grouped-by CAnd* φ*1* $\wedge$ *grouped-by CAnd* φ*2*
  apply (*rule iffI*)
  apply (*induction FAnd* φ*1* φ*2 rule*: *grouped-by.induct*)
  using *connected-is-group*[*of CAnd* φ*1* φ*2*] **by** *auto*

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

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

lemma *grouped-by-COr-FEq*[*simp*]: $\neg$ *grouped-by COr* (*FEq* φ*1* φ*2*)
  apply *clarify*
   apply (*induction FEq* φ*1* φ*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* φ*1* φ*2*) $\Longrightarrow$ *is-conj-with-TF* φ*1* $\wedge$ *is-conj-with-TF* φ*2*
  unfolding *is-conj-with-TF-def*
  apply (*induction FAnd* φ*1* φ*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. 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

13 Resolution

13.1 Simplification Rules

inductive *simplify* :: $'v$ *clauses* $\Rightarrow$ $'v$ *clauses* $\Rightarrow$ *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** $\langle A \in N \rangle$ **by** (*simp add*: *true-clss-def*)
moreover have $I \models A \Longrightarrow I \models B$ **using** $\langle A <\# B \rangle$ **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* $\psi\ \psi'$
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* $\psi\ \psi'$
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* $\psi\ \psi'$
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* $\psi\ \psi'$
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

13.2 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*)

13.2.1 Subsumption

definition *subsumes* :: $'a$ *literal multiset* $\Rightarrow$ $'a$ *literal multiset* $\Rightarrow$ *bool* **where**
subsumes $\chi\ \chi' \longleftrightarrow$
$(\forall I.$ *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 $\chi\ \chi$
unfolding *subsumes-def* **by** *auto*

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

13.3 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 $(\exists x \in \mathit{Set.remove}\ a\ A.\ P\ x) \longleftrightarrow (\exists x \in A.\ x \neq a \wedge P\ x)$ **by** *auto*
  } **note** *ex-member-remove* = *this*
  {
   fix a $a0$ :: $'v$ *clause* **and** A :: $'v$ *clauses* **and** y
   assume $a \in A$ **and** $a0 \subset\# a$
   then have $(\exists x \in A.\ \mathit{subsumes}\ x\ y) \longleftrightarrow (\mathit{subsumes}\ a\ y \ \vee (\exists x \in A.\ x \neq a \wedge \mathit{subsumes}\ x\ y))$
     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: $x \in \mathit{snd}\ (N - \{B\},\ \mathit{already\text{-}used})$ **and** [*simp*]: $x = (a,\ b)$
    obtain p **where** p: $\mathit{Pos}\ p \in\#\ a \wedge \mathit{Neg}\ p \in\#\ b$ **and**
     q: $(\exists \chi{\in}N.\ \mathit{subsumes}\ \chi\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\})))$
      $\vee\ \mathit{tautology}\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\}))$
     using *inv x* **by** *fastforce*
    consider (*taut*) $\mathit{tautology}\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\}))$ |
     (χ) χ **where** $\chi \in N$ $\mathit{subsumes}\ \chi\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\}))$
      $\neg\mathit{tautology}\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\}))$
     using q **by** *auto*
    then show
     $\exists p.\ \mathit{Pos}\ p \in\#\ a \wedge \mathit{Neg}\ p \in\#\ b$
       $\wedge\ ((\exists \chi{\in}\mathit{fst}\ (N - \{B\},\ \mathit{already\text{-}used}).\ \mathit{subsumes}\ \chi\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\})))$
        $\vee\ \mathit{tautology}\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\})))$
     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 $C + \{\#\mathit{Pos}\ P\#\} + \{\#\mathit{Neg}\ P\#\} \in N$
   assume *already-used-inv* (N, *already-used*)
   and $(a,\ b) \in \mathit{snd}\ (N - \{C + \{\#\mathit{Pos}\ P\#\} + \{\#\mathit{Neg}\ P\#\}\},\ \mathit{already\text{-}used})$
   then obtain p **where**
    $\mathit{Pos}\ p \in\#\ a \wedge \mathit{Neg}\ p \in\#\ b\ \wedge$
     $((\exists \chi{\in}\mathit{fst}\ (N \cup \{C + \{\#\mathit{Pos}\ P\#\} + \{\#\mathit{Neg}\ P\#\}\},\ \mathit{already\text{-}used}).$
        $\mathit{subsumes}\ \chi\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\})))$
      $\vee\ \mathit{tautology}\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\})))$
    by *fastforce*
   moreover have $\mathit{tautology}\ (C + \{\#\mathit{Pos}\ P\#\} + \{\#\mathit{Neg}\ P\#\})$ **by** *auto*
   ultimately show
    $\exists p.\ \mathit{Pos}\ p \in\#\ a \wedge \mathit{Neg}\ p \in\#\ b$
    $\wedge\ ((\exists \chi{\in}\mathit{fst}\ (N - \{C + \{\#\mathit{Pos}\ P\#\} + \{\#\mathit{Neg}\ P\#\}\},\ \mathit{already\text{-}used}).$
       $\mathit{subsumes}\ \chi\ (a - \{\#\mathit{Pos}\ p\#\} + (b - \{\#\mathit{Neg}\ p\#\})))$

$\vee$ *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*#}} ∪ {*A* + {#*L*#}}) ⊨*s* ψ − {*A* + {#*L*#} + {#*L*#}} ∪ {*A* + {#*L*#}}
∧ *consistent-interp* (*LL* (ψ − {*A* + {#*L*#} + {#*L*#}} ∪ {*A* + {#*L*#}}))
∧ *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* ⊨*s ?*ψ' **and** *cons*: *consistent-interp I* **and** *tot*: *total-over-m I ?*ψ'
using *sat* **unfolding** *satisfiable-def* **by** *auto*
have *I* ⊨ *A* **using** *A I* **by** (*metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def*)
then have *I* ⊨ *B* **using** *AB subset-mset.less-imp-le true-cls-mono-leD* **by** *blast*
then have *I* ⊨*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') ⟶ *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′*) ⟶ *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*]:
(⋀*xs*:: *′v sem-tree*. (⋀*ys*:: *′v sem-tree*. *sem-tree-size ys* < *sem-tree-size xs* ⟹ *P ys*) ⟹ *P xs*)
⟹ *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 = (∃ χ. ¬ *I* ⊨ χ ∧ χ ∈ ψ ∧ *total-over-m I* {χ}) |
partial-interps (*Node v ag ad*) *I* ψ ⟷
(*partial-interps ag* (*I* ∪ {*Pos v*}) ψ ∧ *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* $\Rightarrow$ *'v clauses* $\Rightarrow$ *'v sem-tree* **where**
build-sem-tree atms ψ =
(*if atms* = {} $\vee$ $\neg$ *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*]:
$\neg$*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: $\exists\chi \in \psi.\ \neg\ Ia \models \chi$
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*: $\bigwedge\chi.\ \chi \in \psi \Longrightarrow$ *total-over-m Ia* $\{\chi\}$
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 ∪ *atms-of-s* {} **unfolding** *atms-of-s-def* **by** *auto*
moreover have *atms-of-ms* ψ ∩ *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* × (*'v clause* × *'v clause* × *'v*) *set*
assumes *count* χ *L* = *n*
and *L* ∈# χ **and** χ ∈ *fst* ψ
shows $\exists\psi'\ \chi'$. *inference*** $\psi\ \psi'$ ∧ χ' ∈ *fst* ψ' ∧ (∀ *L*. *L* ∈# χ ⟷ *L* ∈# χ')
∧ *count* χ' *L* = *1*
∧ (∀ φ. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ')
∧ (*I* ⊨ χ ⟷ *I* ⊨ χ')
∧ (∀ *I'*. *total-over-m I'* {χ} ⟶ *total-over-m I'* {χ'})
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 χ ∈ *fst* ψ
and ∀ *L*. (*L* ∈# χ) ⟷ (*L* ∈# χ)
and *count* χ *L* = (*1*::*nat*)
and ∀ φ. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ
by (*auto simp add*: *count L* χ)
then have *?case* **by** *metis*
}
moreover {
assume *n* > *0*
then have ∃ *C*. χ = *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*: χ = *C* + {#*L*, *L*#} **by** *metis*
let *?*χ' = *C* +{#*L*#}

let $?\psi' = (fst\ \psi \cup \{?\chi'\},\ snd\ \psi)$
have φ: $\forall \varphi \in fst\ \psi.\ (\varphi \in fst\ \psi \vee \varphi \neq ?\chi') \longleftrightarrow \varphi \in fst\ ?\psi'$ **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 $\chi'\psi'$: $?\chi' \in fst\ ?\psi'$ **by** *auto*
ultimately obtain ψ'' **and** χ''
where
  $inference^{**}$ $?\psi'$ ψ'' **and**
  α: $\chi'' \in fst\ \psi''$ **and**
  $\forall La.\ (La \in\# ?\chi') \longleftrightarrow (La \in\# \chi'')$ **and**
  β: *count* χ'' $L = (1::nat)$ **and**
  φ': $\forall \varphi.\ \varphi \in fst\ ?\psi' \longrightarrow \varphi \in fst\ \psi''$ **and**
  $I\chi$: $I \models ?\chi' \longleftrightarrow I \models \chi''$ **and**
  tot: $\forall I'.$ *total-over-m* I' $\{?\chi'\} \longrightarrow$ *total-over-m* I' $\{\chi''\}$
  using *IH*[*of* $?\chi'$ $?\psi'$] *count'* $L\chi'$ $\chi'\psi'$ **by** *blast*

then have $inference^{**}$ ψ ψ''
and $\forall La.\ (La \in\# \chi) \longleftrightarrow (La \in\# \chi'')$
using *inf* **unfolding** *C* **by** *auto*
moreover have $\forall \varphi.\ \varphi \in fst\ \psi \longrightarrow \varphi \in fst\ \psi''$ **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' $\{\chi\} \longrightarrow$ *total-over-m* I' $\{\chi''\}$
  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* $(fst\ \psi)$ **and** *already-used-inv* ψ
and *partial-interps tree I* $(fst\ \psi)$
shows $\exists (tree' :: 'v\ sem\text{-}tree)\ \psi'.\ inference^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ I\ (fst\ \psi')$
  $\wedge\ (sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ tree \vee sem\text{-}tree\text{-}size\ tree = 0)$
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**
      $tot\chi$: *total-over-m* $(I \cup \{Pos\ v\})$ $\{\chi\}$ **and**

$\chi\psi$: $\chi \in fst\ \psi$ **and**
χ': $\neg\ I \cup \{Neg\ v\} \models \chi'$ **and**
$tot\chi'$: $total\text{-}over\text{-}m\ (I \cup \{Neg\ v\})\ \{\chi'\}$ **and**
$\chi'\psi$: $\chi' \in fst\ \psi$
using *part* **unfolding** *xs* **by** *auto*
have *Posv*: $\neg Pos\ v \in\#\ \chi$ **using** χ **unfolding** *true-cls-def true-lit-def* **by** *auto*
have *Negv*: $\neg Neg\ v \in\#\ \chi'$ **using** χ' **unfolding** *true-cls-def true-lit-def* **by** *auto*
{
assume $Neg\chi$: $\neg Neg\ v \in\#\ \chi$
have $\neg\ I \models \chi$ **using** χ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have $total\text{-}over\text{-}m\ I\ \{\chi\}$
using *Posv* $Neg\chi$ *atm-imp-pos-or-neg-lit* $tot\chi$ **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*
ultimately have $partial\text{-}interps\ Leaf\ I\ (fst\ \psi)$
and $sem\text{-}tree\text{-}size\ Leaf < sem\text{-}tree\text{-}size\ xs$
and $inference^{**}\ \psi\ \psi$
unfolding *xs* **by** (*auto simp add*: $\chi\psi$)
}
moreover {
assume $Pos\chi$: $\neg Pos\ v \in\#\ \chi'$
then have $I\chi$: $\neg\ I \models \chi'$ **using** χ' *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have $total\text{-}over\text{-}m\ I\ \{\chi'\}$
using *Negv* $Pos\chi$ *atm-imp-pos-or-neg-lit* $tot\chi'$
unfolding *total-over-m-def total-over-set-def* **by** *fastforce*
ultimately have $partial\text{-}interps\ Leaf\ I\ (fst\ \psi)$ **and**
$sem\text{-}tree\text{-}size\ Leaf < sem\text{-}tree\text{-}size\ xs$ **and**
$inference^{**}\ \psi\ \psi$
using $\chi'\psi$ $I\chi$ **unfolding** *xs* **by** *auto*
}
moreover {
assume *neg*: $Neg\ v \in\#\ \chi$ **and** *pos*: $Pos\ v \in\#\ \chi'$
then obtain ψ' $\chi 2$ **where** *inf*: $rtranclp\ inference\ \psi\ \psi'$ **and** $\chi 2incl$: $\chi 2 \in fst\ \psi'$
and $\chi\chi 2\text{-}incl$: $\forall L.\ L \in\#\ \chi \longleftrightarrow L \in\#\ \chi 2$
and $count\chi 2$: $count\ \chi 2\ (Neg\ v) = 1$
and φ: $\forall \varphi{::}'v\ literal\ multiset.\ \varphi \in fst\ \psi \longrightarrow \varphi \in fst\ \psi'$
and $I\chi$: $I \models \chi \longleftrightarrow I \models \chi 2$
and $tot\text{-}imp\chi$: $\forall I'.\ total\text{-}over\text{-}m\ I'\ \{\chi\} \longrightarrow total\text{-}over\text{-}m\ I'\ \{\chi 2\}$
using *can-decrease-count*[*of* χ *Neg v count* χ (*Neg v*) ψ *I*] $\chi\psi$ $\chi'\psi$ **by** *auto*

have $\chi' \in fst\ \psi'$ **by** (*simp add*: $\chi'\psi$ φ)
with *pos*
obtain ψ'' $\chi 2'$ **where**
inf': $inference^{**}\ \psi'\ \psi''$
and $\chi 2'\text{-}incl$: $\chi 2' \in fst\ \psi''$
and $\chi'\chi 2\text{-}incl$: $\forall L{::}'v\ literal.\ (L \in\#\ \chi') = (L \in\#\ \chi 2')$
and $count\chi 2'$: $count\ \chi 2'\ (Pos\ v) = (1{::}nat)$
and φ': $\forall \varphi{::}'v\ literal\ multiset.\ \varphi \in fst\ \psi' \longrightarrow \varphi \in fst\ \psi''$
and $I\chi'$: $I \models \chi' \longleftrightarrow I \models \chi 2'$
and $tot\text{-}imp\chi'$: $\forall I'.\ total\text{-}over\text{-}m\ I'\ \{\chi'\} \longrightarrow total\text{-}over\text{-}m\ I'\ \{\chi 2'\}$
using *can-decrease-count*[*of* χ' *Pos v count* χ' (*Pos v*) ψ' *I*] **by** *auto*

obtain *C* **where** $\chi 2$: $\chi 2 = C + \{\#Neg\ v\#\}$ **and** *negC*: $Neg\ v \notin\#\ C$ **and** *posC*: $Pos\ v \notin\#\ C$
proof –
have $\bigwedge m.\ Suc\ 0 - count\ m\ (Neg\ v) = count\ (\chi 2 - m)\ (Neg\ v)$
by (*simp add*: $count\chi 2$)

then show *?thesis*
using *that* **by** (*metis* (*no-types*) *One-nat-def Posv Suc-inject Suc-pred* $\chi\chi$*2-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**
$\chi 2'$: $\chi 2' = C' + \{\#Pos\ v\#\}$ **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* $\psi\ \psi'$] *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 $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \notin snd\ \psi''$
then have *inf''*: *inference* ψ'' $(fst\ \psi'' \cup \{C + C'\}, snd\ \psi'' \cup \{(\chi 2', \chi 2)\})$
using *add.commute* φ' χ*2incl* ‹$\chi 2' \in fst\ \psi''$› **unfolding** $\chi 2$ $\chi 2'$
by (*metis prod.collapse inference-step resolution*)
have $inference^{**}\ \psi\ (fst\ \psi'' \cup \{C + C'\}, snd\ \psi'' \cup \{(\chi 2', \chi 2)\})$
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*: $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \in snd\ \psi''$
then have $(\exists \chi \in fst\ \psi''. (\forall I. total\text{-}over\text{-}m\ I\ \{C{+}C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\})$
$\wedge (\forall I. total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C))$
$\vee\ tautology\ (C' + C)$

```
      proof -
        obtain p where p: Pos p ∈# ({#Pos v#} + C') and
        n: Neg p ∈# ({#Neg v#} + C) and
        decomp: ((∃χ∈fst ψ''.
                  (∀I. total-over-m I {({#Pos v#} + C') - {#Pos p#}
                          + (({#Neg v#} + C) - {#Neg p#})}
                     ⟶ total-over-m I {χ})
                  ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ
                  ⟶ I ⊨ ({#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
```

$\longrightarrow$ (*partial-interps ag* ($I \cup \{Pos\ v\}$) (*fst* ψ) $\longrightarrow$
($\exists$ *tree'* ψ'. *inference*** ψ ψ' $\wedge$ *partial-interps tree'* ($I \cup \{Pos\ v\}$) (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* $<$ *sem-tree-size ag* $\vee$ *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'* ($I \cup \{Pos\ v\}$) (*fst* ψ')
and *size*: *sem-tree-size tree'* $<$ *sem-tree-size ag* $\vee$ *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* ψ')
using *rtranclp-inference-preserve-partial-tree inf partad* **by** *metis*
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* ($I \cup \{Pos\ v\}$) (*fst* ψ) **and**
partial-interps ad ($I \cup \{Neg\ v\}$) (*fst* ψ)
using *part partial-interps.simps*(*2*) **unfolding** *xs* **by** *metis+*
moreover have *sem-tree-size ad* $<$ *sem-tree-size xs* $\longrightarrow$ *finite* (*fst* ψ) $\longrightarrow$ *already-used-inv* ψ
$\longrightarrow$ (*partial-interps ad* ($I \cup \{Neg\ v\}$) (*fst* ψ)
$\longrightarrow$ ($\exists$ *tree'* ψ'. *inference*** ψ ψ' $\wedge$ *partial-interps tree'* ($I \cup \{Neg\ v\}$) (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* $<$ *sem-tree-size ad* $\vee$ *sem-tree-size ad* $=$ *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'* ($I \cup \{Neg\ v\}$) (*fst* ψ')
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* ψ')
using *rtranclp-inference-preserve-partial-tree inf partag* **by** *metis*
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 *inference-completeness-inv*:
fixes ψ :: *'v ::linorder state*
assumes
unsat: $\neg$*satisfiable* (*fst* ψ) **and**
finite: *finite* (*fst* ψ) **and**
a-u-v: *already-used-inv* ψ
shows $\exists\psi'$. (*inference*** ψ ψ' $\wedge$ $\{\#\} \in$ *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: $\neg$ {} $\models$ χ **and** *tot*χ: *total-over-m* {} {χ} **and** $\chi\psi$: $\chi \in$ *fst* ψ
using H **unfolding** *tree* **by** *auto*
moreover have {#} = χ
using *tot*χ **unfolding** *total-over-m-def total-over-set-def* **by** *fastforce*
moreover have *inference*** ψ ψ **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* $\vee$ *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*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows $\exists\psi'$. (*rtranclp inference* ψ ψ' $\wedge$ {#} $\in$ *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** {#} $\in$ *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 ($\exists\psi'$. (*inference*** ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
using *assms inference-completeness inference-soundness* **by** *metis*

13.4 Lemma about the simplified state

abbreviation *simplified* $\psi \equiv$ (*no-step simplify* ψ)

lemma *simplified-count*:
  assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
  shows *count* χ $L \leq 1$
proof –
  {
    let $?\chi' = \chi - \{\#L, L\#\}$
    assume *count* χ $L \geq 2$
    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 $\neg$ $l \in\# \psi$

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 $\{\psi\}$] *add.commute*)
next
case (*condensation A L*)
have $A + \{\#L\#\} + \{\#L\#\} + \{\#l\#\} \in \{\psi\}$
using *A condensation.hyps* **by** *blast*
then have $\{\#L, L\#\} + (A + \{\#l\#\}) \in \{\psi\}$
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*: $\psi' \subseteq \psi$
shows *simplified* ψ'
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then obtain ψ'' **where** *simplify* ψ' ψ'' **by** *metis*
then have $\exists\, l'.$ *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 $N \in \psi$

shows *simplified* $\{N\}$
using *assms* **by** (*metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono*)

lemma *subsumes-imp-formula*:
assumes $\psi \leq\# \varphi$
shows $\{\psi\} \models p\ \varphi$
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** $\forall \chi \in \psi'$. $\neg$*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*)

13.5 Resolution and Invariants

inductive *resolution* :: *'v state* ⇒ *'v state* ⇒ *bool* **where**
full1-simp: *full1 simplify* N N' ⟹ *resolution* (N, *already-used*) (N', *already-used*) |
inferring: *inference* (N, *already-used*) (N', *already-used'*) ⟹ *simplified* N
⟹ *full simplify* N' N'' ⟹ *resolution* (N, *already-used*) (N'', *already-used'*)

13.5.1 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* × *'a literal multiset*) *set* ⇒ *'a set* ⇒ *bool* **where**
already-used-all-simple already-used vars ≡
(∀ (*A*, *B*) ∈ *already-used*. *simplified* {*A*} ∧ *simplified* {*B*} ∧ *atms-of A* ⊆ *vars* ∧ *atms-of B* ⊆ *vars*)

lemma *already-used-all-simple-vars-incl*:
  assumes *vars* ⊆ *vars'*
  shows *already-used-all-simple a vars* ⟹ *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*) ⊆ *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′*) ⊆ *atms-of-ms* (*fst S*)
by (*simp add*: *infstar local.finite rtranclp-resolution-atms-of*)
then have *atms-of-ms* (*fst S′*) ⊆ *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* ⊂ *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* $S \subset$ *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* $\equiv$ *simple-clss vars* $\times$ *simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
assumes *already-used-all-simple s vars* **and** *finite vars*
shows $s \subseteq$ *already-used-top vars*
proof
fix x
assume *x-s*: $x \in s$
obtain A B **where** x: $x = (A, B)$ **by** (*cases* x, *auto*)
then have *simplified* $\{A\}$ **and** *atms-of* $A \subseteq$ *vars* **using** *assms*(*1*) *x-s* **by** *fastforce+*
then have A: $A \in$ *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* $B \subseteq$ *vars* **using** *assms*(*1*) *x-s* x **by** *fast+*
then have B: $B \in$ *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 $x \in$ *simple-clss vars* $\times$ *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* $\subseteq$ *var'* **and** *finite var'*
shows *already-used-top var* $\subseteq$ *already-used-top var'*
using *assms simple-clss-mono* **by** *auto*

lemma *already-used-all-simple-finite*:
fixes $s :: ('a\ literal\ multiset \times 'a\ literal\ multiset)\ set$ **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* $\psi \equiv$ *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* $\times$ *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*) $\geq$ *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) $\subseteq$ *?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

13.5.2 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*) $\subseteq$ *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*}
proof −
{
fix *a b* :: *'v*::*linorder state*
assume (*b*, *a*) $\in$ {(*y*, *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*}
then have
atms-of-ms (*fst a*) $\subseteq$ *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* $\subseteq$ *already-used-top vars* **by** *auto*
moreover have *snd b* $\subseteq$ *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*) $\subseteq$ *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*) ⊆ *vars* ∧ *simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
∪ {(*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*}) (**is** *wf* (*?R* ∪ *?S*))
proof −
have *Domain ?R Int Range ?S* = {} **using** *resolution-always-simplified* **by** *auto blast*
then show *wf* (*?R* ∪ *?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*** ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
using *assms* **apply** *induction*
using *simplify-clause-preserves-sat* **by** *blast+*

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
using *assms rtanclp-simplify-preserves-unsat*[*of* ψ ψ'] *tranclp-into-rtranclp*
unfolding *full1-def* **by** *metis*

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
using *assms rtanclp-simplify-preserves-unsat*[*of* ψ ψ'] **unfolding** *full-def* **by** *metis*

lemma *resolution-preserves-unsat*:
assumes *resolution* ψ ψ'
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*** ψ ψ'
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* $\{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \land L \in\#\ \varphi \land P\ \varphi\ L\}$
  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 $\{f\ \varphi\ L\ |\varphi\ L.\ (\varphi = x \lor \varphi \in N) \land L \in\#\ \varphi \land P\ \varphi\ L\} \subseteq \{f\ x\ L \mid L.\ L \in\#\ x \land P\ x\ L\}$
    $\cup\ \{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \land L \in\#\ \varphi \land P\ \varphi\ L\}$ **by** *auto*
  moreover have *finite* $\{f\ x\ L \mid L.\ L \in\#\ x\}$ **by** *auto*
  ultimately show *?case* **using** *IH finite-subset* **by** *fastforce*

qed

value *card*
value *filter-mset*
value $\{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}$
value $(\lambda\varphi.\ msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\})$

syntax
  -comprehension1'-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
    (({#-/. - : *setof* -#}))
translations
  {#*e*. *x*: *setof M*#} == *CONST set-mset* (*CONST image-mset* (%*x*. *e*) *M*)
value {# *a*. *a* : *setof* {#*1*,*1*,*2*::*int*#}#} = {*1*,*2*}

definition *sum-count-ge-2* :: *'a multiset set* ⇒ *nat* (Ξ) **where**
sum-count-ge-2 ≡ *folding.F* $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0$

interpretation *sum-count-ge-2*:
  folding $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0$
rewrites
  folding.F $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0 = sum\text{-}count\text{-}ge\text{-}2$
proof −
  show *folding* $(\lambda\varphi.\ op\ +\ (msetsum\ (image\text{-}mset\ (count\ \varphi)\ \{\#\ L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\})))$
    by *standard auto*
  then interpret *sum-count-ge-2*:
    folding $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0$ **.**
  show *folding.F* $(\lambda\varphi.\ op\ +\ (msetsum\ (image\text{-}mset\ (count\ \varphi)\ \{\#\ L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\})))\ 0$
    = *sum-count-ge-2* **by** (*auto simp add*: *sum-count-ge-2-def*)
qed

lemma *finite-incl-le-setsum*:
 finite (*B*::*'a multiset set*) ⟹ *A* ⊆ *B* ⟹ Ξ *A* ≤ Ξ *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* ∈ *A*)
      assume *a* ∉ *A*
      then have *A* ⊆ *F* **using** *AF* **by** *auto*
      then show *?case* **using** *IH*[*of A*] **by** (*simp add*: *aF local.finite*)
    next
      assume *aA*: *a* ∈ *A*
      then have *A* − {*a*} ⊆ *F* **using** *AF* **by** *auto*
      then have Ξ (*A* − {*a*}) ≤ Ξ *F* **using** *IH* **by** *blast*
      then show *?case*
        proof −
          obtain *nn* :: *nat* ⇒ *nat* ⇒ *nat* **where**
            ∀ *x0 x1*. (∃ *v2*. *x0* = *x1* + *v2*) = (*x0* = *x1* + *nn x0 x1*)
            by *moura*
          then have Ξ *F* = Ξ (*A* − {*a*}) + *nn* (Ξ *F*) (Ξ (*A* − {*a*}))
            by (*meson* ‹Ξ (*A* − {*a*}) ≤ Ξ *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*: $\{\#\ La \in\#\ A.\ L \neq La \longrightarrow 2 \leq count\ A\ La\#\} =$
$\{\#\ La \in\#\ A.\ L \neq La \wedge 2 \leq count\ A\ La\#\} + replicate\text{-}mset\ (count\ A\ L)\ L$
by (*auto simp*: *multiset-eq-iff*)
show *?thesis*
by (*auto simp*: *mset-decomp mset-decomp2 filter-mset-eq ac-simps*)
qed
have $\Xi\ ?N' < \Xi\ N$
proof *cases*
assume *a1*: $?C' \in N$
then show *?thesis*
proof −
have *f2*: $\bigwedge m\ M.\ insert\ (m::'a\ literal\ multiset)\ (M - \{m\}) = M \cup \{\} \vee m \notin M$
using *Un-empty-right insert-Diff* **by** *blast*
have *f3*: $\bigwedge m\ M\ Ma.\ insert\ (m::'a\ literal\ multiset)\ M - insert\ m\ Ma = M - insert\ m\ Ma$
by *simp*
then have *f4*: $\bigwedge M\ m.\ M - \{m::'a\ literal\ multiset\} = M \cup \{\} \vee m \in M$
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 $\bigwedge m\ M.\ insert\ (m::'a\ literal\ multiset)\ M = M \cup \{\} \vee m \notin M$
using *f3 f2 Un-empty-right add.right-neutral insert-iff* **by** *fastforce*
then have $\Xi\ (N - \{A + \{\#L\#\} + \{\#L\#\}\}) < \Xi\ N$
using *f5 f4* **by** (*metis Un-empty-right* $\langle \Xi\ \{A + \{\#L\#\}\} < \Xi\ \{A + \{\#L\#\} + \{\#L\#\}\} \rangle$

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 $?C' \notin N$
have *mset-decomp*:
$\{\# La \in\# A. (L = La \longrightarrow Suc\ 0 \leq count\ A\ La) \wedge (L \neq La \longrightarrow 2 \leq count\ A\ La)\#\}$
$= \{\# La \in\# A. L \neq La \wedge 2 \leq count\ A\ La\#\} +$
$\{\# La \in\# A. L = La \wedge Suc\ 0 \leq count\ A\ L\#\}$
by (*auto simp*: *multiset-eq-iff ac-simps*)
have *mset-decomp2*: $\{\# La \in\# A. L \neq La \longrightarrow 2 \leq count\ A\ La\#\} =$
$\{\# La \in\# A. L \neq La \wedge 2 \leq count\ A\ La\#\}$ + *replicate-mset* (*count A L*) *L*
by (*auto simp*: *multiset-eq-iff*)

show *?thesis*
using ‹$\Xi \{A + \{\#L\#\}\} < \Xi \{A + \{\#L\#\} + \{\#L\#\}\}$› *condensation.hyps fin sum-count-ge-2.remove*[*of - A* + {#*L*#} + {#*L*#}] ‹$?C' \notin N$›
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 $\Xi (N - \{B\}) \leq \Xi N$
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* $\wedge$ *simplify N N′*$\}$
using *assms* **apply** (*rule wfP-if-measure*[*of finite simplify* $\lambda N.$ *card N* $+ \Xi N$])
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 $\exists (tree' :: 'v\ sem\text{-}tree)\ \psi'.\ resolution^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ I\ (fst\ \psi')$
$\wedge$ (*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* $= 0 \land$ *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* $(I \cup \{Pos\ v\})$ $\{\chi\}$ **and**
$\chi\psi$: $\chi \in$ *fst* ψ **and**
χ': $\neg$ $I \cup \{Neg\ v\} \models \chi'$ **and**
totχ': *total-over-m* $(I \cup \{Neg\ v\})$ $\{\chi'\}$ **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: $\neg$*Neg v* $\in\#$ χ
then have $\neg$ $I \models \chi$ **using** χ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m I* $\{\chi\}$
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: $\neg$*Pos v* $\in\#$ χ'
then have $I\chi$: $\neg$ $I \models \chi'$ **using** χ' *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m I* $\{\chi'\}$
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** $\chi'\psi$ $I\chi$ **unfolding** *xs* **by** *auto*
}
moreover {
assume *neg*: *Neg v* $\in\#$ χ **and** *pos*: *Pos v* $\in\#$ χ'
have *count* χ (*Neg v*) = *1*
using *simplified-count*[*OF simp* $\chi\psi$] *neg*
by (*simp add*: *dual-order.antisym*)
have *count* χ' (*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* χ (*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* χ' (*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* 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* $\{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})$)) $\vee$ *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* ψ. ($\forall I$. *total-over-m I* $\{C+C'\} \longrightarrow$ *total-over-m I* $\{\chi\}$)
$\wedge$ ($\forall I$. *total-over-m I* $\{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C$)

then obtain ϑ **where**
ϑ: $\vartheta \in$ *fst* ψ **and**
tot-ϑ*-CC'*: $\forall I.$ *total-over-m* I $\{C+C'\} \longrightarrow$ *total-over-m* I $\{\vartheta\}$ **and**
ϑ*-inv*: $\forall I.$ *total-over-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'* ψ'. *resolution*** ψ ψ' $\wedge$ *partial-interps tree'* ($I \cup \{Pos\ v\}$) (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* < *sem-tree-size ag* $\vee$ *sem-tree-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*** ψ ψ'
and *part*: *partial-interps tree'* ($I \cup \{Pos\ v\}$) (*fst* ψ')
and *size*: *sem-tree-size tree'* < *sem-tree-size ag* $\vee$ *sem-tree-size ag* = *0*
using *finite part rtranclp.rtrancl-refl a-u-i simp* **by** *blast*

have *partial-interps ad* ($I \cup \{Neg\ v\}$) (*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* ($I \cup \{Pos\ v\}$) (*fst* ψ) **and**
partial-interps ad ($I \cup \{Neg\ v\}$) (*fst* ψ)
using *part partial-interps.simps*(*2*) **unfolding** *xs* **by** *metis+*
moreover have *sem-tree-size ad* < *sem-tree-size xs* $\longrightarrow$ *finite* (*fst* ψ) $\longrightarrow$ *already-used-inv* ψ

$\longrightarrow$ (*partial-interps ad* $(I \cup \{Neg\ v\})$ (*fst* ψ) $\longrightarrow$ *simplified* (*fst* ψ)
$\longrightarrow$ ($\exists$ *tree'* ψ'. *resolution*** ψ ψ' $\wedge$ *partial-interps tree'* $(I \cup \{Neg\ v\})$ (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* $<$ *sem-tree-size ad* $\vee$ *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'* $(I \cup \{Neg\ v\})$ (*fst* ψ')
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 simp* **by** *blast*

have *partial-interps ag* $(I \cup \{Pos\ v\})$ (*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: $\neg$*satisfiable* (*fst* ψ) **and**
finite: *finite* (*fst* ψ) **and**
a-u-v: *already-used-inv* ψ
shows $\exists\psi'$. (*resolution*** ψ ψ' $\wedge$ $\{\#\}$ $\in$ *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: $\neg$ $\{\}$ $\models$ χ **and** *tot*χ: *total-over-m* $\{\}$ $\{\chi\}$ **and** $\chi\psi$: $\chi \in$ *fst* ψ
using H **unfolding** *tree* **by** *auto*
moreover have $\{\#\} = \chi$
using H *atms-empty-iff-empty tot*χ
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*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* $\psi = \{\}$
shows $\exists \psi'$. (*resolution*** ψ ψ' $\wedge$ {#} $\in$ *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* ψ = {}
shows $(\exists\,\psi'.\ (resolution^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow unsatisfiable\ (fst\ \psi)$
  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*: ¬ *?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 $(\exists\,\psi'.\ (resolution^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow (\exists\, a\text{-}u\text{-}v.\ resolution^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))$
    (**is** *?A* ⟷ *?B*)
proof
  assume *?B*
  then show *?A* **by** *auto*
next
  assume *?A*
  then obtain χs *a-u-v* **where** χs: $resolution^{**}$ ψ $(\chi s,\ a\text{-}u\text{-}v)$ **and** *F*: $\{\#\} \in \chi s$ **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 $\{\#\} \in \chi s'$
      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* $\psi = \{\}$
shows ($\exists$ *a-u-v*. (*resolution*** ψ ({{#}}, *a-u-v*))) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
using *assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone*
by *metis*

end

theory *Partial-Annotated-Clausal-Logic*
imports *Partial-Clausal-Logic*

begin

14 Partial Clausal Logic

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

14.1 Marked Literals

14.1.1 Definition

datatype (*'v*, *'lvl*, *'mark*) *marked-lit* =
is-marked: *Marked* (*lit-of*: *'v literal*) (*level-of*: *'lvl*) |
is-proped: *Propagated* (*lit-of*: *'v literal*) (*mark-of*: *'mark*)

lemma *marked-lit-list-induct*[*case-names nil marked proped*]:
assumes *P* [] **and**
$\bigwedge$*L l xs*. *P xs* $\Longrightarrow$ *P* (*Marked L 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-marked-ex-Marked*:
is-marked L $\Longrightarrow$ $\exists$ *K lvl*. *L* = *Marked K lvl*
by (*cases L*) *auto*

type-synonym (*'v*, *'l*, *'m*) *marked-lits* = (*'v*, *'l*, *'m*) *marked-lit list*

definition *lits-of* :: (*'a*, *'b*, *'c*) *marked-lit set* $\Rightarrow$ *'a literal set* **where**
lits-of Ls = *lit-of* ‘ *Ls*

abbreviation *lits-of-l* :: (*'a*, *'b*, *'c*) *marked-lit list* $\Rightarrow$ *'a literal set* **where**
lits-of-l Ls $\equiv$ *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-append*[*simp*]:
lits-of $(l \cup l')$ = *lits-of* $l \cup$ *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 $\equiv$ (λa. {#*lit-of a*#})

abbreviation *unmark-s* **where**
unmark-s M $\equiv$ *unmark* ‘ *M*

abbreviation *unmark-l* **where**
unmark-l M $\equiv$ *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 = {} $\longleftrightarrow$ *M* = []
by (*induct M*) (*auto simp*: *lits-of-def*)

14.1.2 Entailment

definition *true-annot* :: ('*a*, '*l*, '*m*) *marked-lits* $\Rightarrow$ '*a clause* $\Rightarrow$ *bool* (**infix** $\models$*a* *49*) **where**
I $\models$*a C* $\longleftrightarrow$ (*lits-of-l I*) $\models$ *C*

definition *true-annots* :: ('*a*, '*l*, '*m*) *marked-lits* $\Rightarrow$ '*a clauses* $\Rightarrow$ *bool* (**infix** $\models$*as* *49*) **where**
I $\models$*as CC* $\longleftrightarrow$ ($\forall$ *C* $\in$ *CC*. *I* $\models$*a C*)

lemma *true-annot-empty-model*[*simp*]:
$\neg$[] $\models$*a* ψ
unfolding *true-annot-def true-cls-def* **by** *simp*

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

lemma *empty-true-annots-def*[*iff*]:
[] $\models$*as* $\psi \longleftrightarrow \psi$ = {}
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-marked-true-cls*[*iff*]:
$map\ (\lambda M.\ Marked\ M\ a)\ M \models as\ N \longleftrightarrow set\ M \models s\ N$
proof −
have $*$: *lit-of* ‘ ($\lambda M.$ *Marked M a*) ‘ *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 add: *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 add*: *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*

14.1.3 Defined and undefined literals

definition *defined-lit* :: ($'a$, $'l$, $'m$) *marked-lit list* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool*
where
defined-lit $I\ L \longleftrightarrow (\exists\, l.$ *Marked* $L\ l \in$ *set* $I) \vee (\exists\, P.$ *Propagated* $L\ P \in$ *set* $I)$
$\vee\ (\exists\, l.$ *Marked* $(-L)\ l \in$ *set* $I) \vee (\exists\, P.$ *Propagated* $(-L)\ P \in$ *set* $I)$

abbreviation *undefined-lit* :: ($'a$, $'l$, $'m$) *marked-lit list* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool*
where *undefined-lit* $I\ L \equiv \neg$*defined-lit* $I\ L$

lemma *defined-lit-rev*[*simp*]:
defined-lit (*rev* M) $L \longleftrightarrow$ *defined-lit* $M\ L$
unfolding *defined-lit-def* **by** *auto*

lemma *atm-imp-marked-or-proped*:
assumes $x \in$ *set* I
shows
$(\exists\, l.$ *Marked* $(-$ *lit-of* $x)\ l \in$ *set* $I)$
$\vee\ (\exists\, l.$ *Marked* (*lit-of* x) $l \in$ *set* $I)$
$\vee\ (\exists\, l.$ *Propagated* $(-$ *lit-of* $x)\ l \in$ *set* $I)$
$\vee\ (\exists\, l.$ *Propagated* (*lit-of* x) $l \in$ *set* $I)$
using *assms marked-lit.exhaust-sel* **by** *metis*

lemma *literal-is-lit-of-marked*:
assumes $L =$ *lit-of* x
shows $(\exists\, l.\ x =$ *Marked* $L\ l) \vee (\exists\, l'.\ x =$ *Propagated* $L\ l')$
using *assms* **by** (*cases* x) *auto*

lemma *true-annot-iff-marked-or-true-lit*:
defined-lit I L $\longleftrightarrow$ ((*lits-of-l I*) $\models$*l L* $\vee$ (*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-marked*)

lemma *consistent-interp* (*lits-of-l I*) $\Longrightarrow$ *I* $\models$*as N* $\Longrightarrow$ *satisfiable N*
by (*simp add*: *true-annots-true-cls*)

lemma *defined-lit-map*:
defined-lit Ls L $\longleftrightarrow$ *atm-of L* $\in$ ($\lambda l.$ *atm-of* (*lit-of l*)) ' *set Ls*
unfolding *defined-lit-def* **apply** (*rule iffI*)
using *image-iff* **apply** *fastforce*
by (*fastforce simp add*: *atm-of-eq-atm-of dest*: *atm-imp-marked-or-proped*)

lemma *defined-lit-uminus*[*iff*]:
defined-lit I ($-L$) $\longleftrightarrow$ *defined-lit I L*
unfolding *defined-lit-def* **by** *auto*

lemma *Marked-Propagated-in-iff-in-lits-of-l*:
defined-lit I L $\longleftrightarrow$ ($L \in$ *lits-of-l I* $\vee$ $-L \in$ *lits-of-l I*)
unfolding *lits-of-def defined-lit-def*
by (*auto simp*: *rev-image-eqI*) (*rename-tac x*, *case-tac x*, *auto*)+

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*: *Marked-Propagated-in-iff-in-lits-of-l*)

lemma *decided-empty*[*simp*]:
$\neg$*defined-lit* [] *L*
unfolding *defined-lit-def* **by** *simp*

14.2 Backtracking

fun *backtrack-split* :: ('*v*, '*l*, '*m*) *marked-lits*
$\Rightarrow$ ('*v*, '*l*, '*m*) *marked-lits* $\times$ ('*v*, '*l*, '*m*) *marked-lits* **where**
backtrack-split [] = ([], []) |
backtrack-split (*Propagated L P* # *mlits*) = *apfst* ((*op* #) (*Propagated L P*)) (*backtrack-split mlits*) |
backtrack-split (*Marked L l* # *mlits*) = ([], *Marked L l* # *mlits*)

lemma *backtrack-split-fst-not-marked*: $a \in$ *set* (*fst* (*backtrack-split l*)) $\Longrightarrow$ $\neg$*is-marked a*
by (*induct l rule*: *marked-lit-list-induct*) *auto*

lemma *backtrack-split-snd-hd-marked*:
snd (*backtrack-split l*) $\neq$ [] $\Longrightarrow$ *is-marked* (*hd* (*snd* (*backtrack-split l*)))
by (*induct l rule*: *marked-lit-list-induct*) *auto*

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

lemma *backtrack-snd-empty-not-marked*:
backtrack-split M = (M'', []) $\Longrightarrow$ $\forall$ *l*$\in$*set M*. $\neg$ *is-marked l*
by (*metis append-Nil2 backtrack-split-fst-not-marked backtrack-split-list-eq snd-conv*)

lemma *backtrack-split-some-is-marked-then-snd-has-hd*:
$\exists l \in set\ M.\ is\text{-}marked\ l \Longrightarrow \exists M'\ L'\ M''.\ backtrack\text{-}split\ M = (M'', L' \# M')$
by (*metis backtrack-snd-empty-not-marked 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-marked*) *M*, *dropWhile* (*Not o is-marked*) *M*)
proof (*induct M*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons L M*) **then show** *?case* **by** (*cases L*) *auto*
qed

14.3 Decomposition with respect to the marked literals

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

fun *get-all-marked-decomposition* :: ('*a*, '*l*, '*m*) *marked-lits*
⇒ (('*a*, '*l*, '*m*) *marked-lits* × ('*a*, '*l*, '*m*) *marked-lits*) *list* **where**
get-all-marked-decomposition (*Marked L l* # *Ls*) =
(*Marked L l* # *Ls*, []) # *get-all-marked-decomposition Ls* |
get-all-marked-decomposition (*Propagated L P*# *Ls*) =
(*apsnd* ((*op* #) (*Propagated L P*)) (*hd* (*get-all-marked-decomposition Ls*)))
tl (*get-all-marked-decomposition Ls*) |
get-all-marked-decomposition [] = [([], [])]

value *get-all-marked-decomposition* [*Propagated A5 B5*, *Marked C4 D4*, *Propagated A3 B3*,
Propagated A2 B2, *Marked C1 D1*, *Propagated A0 B0*]

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

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

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

lemma *get-all-marked-decomposition-backtrack-split*:
backtrack-split S = (*M*, *M'*) ⟷ *hd* (*get-all-marked-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-marked-decomposition-nil-backtrack-split-snd-nil*:
  get-all-marked-decomposition S = [([], *A*)] ⟹ *snd* (*backtrack-split S*) = []
  by (*simp add*: *get-all-marked-decomposition-backtrack-split sndI*)

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

lemma *get-all-marked-decomposition-fst-empty-or-hd-in-M*:
  assumes *get-all-marked-decomposition M* = (*a*, *b*) # *l*
  shows *a* = [] ∨ (*is-marked* (*hd a*) ∧ *hd a* ∈ *set M*)
  using *assms* **apply** (*induct M arbitrary*: *a b rule*: *marked-lit-list-induct*)
    apply *auto*[*2*]
  by (*metis UnCI backtrack-split-snd-hd-marked get-all-marked-decomposition-backtrack-split*
    get-all-marked-decomposition-decomp hd-in-set list.sel(*1*) *set-append snd-conv*)

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

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

lemma *hd-get-all-marked-decomposition-skip-some*:
  assumes (*x*, *y*) = *hd* (*get-all-marked-decomposition M1*)
  shows (*x*, *y*) ∈ *set* (*get-all-marked-decomposition* (*M0* @ *Marked K i* # *M1*))
  using *assms*
proof (*induct M0*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons L M0*)
  then have *xy*: (*x*, *y*) ∈ *set* (*get-all-marked-decomposition* (*M0* @ *Marked K i* # *M1*)) **by** *blast*

show *?case*
  proof (*cases L*)
    case (*Marked l m*)
    then show *?thesis* **using** *xy* **by** *auto*
  next
    case (*Propagated l m*)
    then show *?thesis*
      using *xy Cons.prems*
      by (*cases get-all-marked-decomposition* (*M0* @ *Marked K i* # *M1*))
        (*auto dest*!: *get-all-marked-decomposition-decomp*
          arg-cong[*of get-all-marked-decomposition - - hd*])
  qed
qed

lemma *get-all-marked-decomposition-snd-union*:
  set M = $\bigcup$(*set* ' *snd* ' *set* (*get-all-marked-decomposition M*)) ∪ {*L* |*L*. *is-marked L* ∧ *L* ∈ *set M*}
  (**is** *?M M* = *?U M* ∪ *?Ls M*)
proof (*induct M arbitrary*:)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons L M*)
  show *?case*
    proof (*cases L*)
      case (*Marked a l*) **note** *L* = *this*
      then have *L* ∈ *?Ls* (*L*#*M*) **by** *auto*
      moreover have *?U* (*L*#*M*) = *?U M* **unfolding** *L* **by** *auto*
      moreover have *?M M* = *?U M* ∪ *?Ls M* **using** *Cons.hyps* **by** *auto*
      ultimately show *?thesis* **by** *auto*
    next
      case (*Propagated a P*)
      then show *?thesis* **using** *Cons.hyps* **by** (*cases* (*get-all-marked-decomposition M*)) *auto*
    qed
qed

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

lemma *get-all-marked-decomposition-remove-unmark-ssed-length*:
  assumes ∀ *l* ∈ *set M'*. ¬*is-marked l*
  shows *length* (*get-all-marked-decomposition* (*M'* @ *M''*))
    = *length* (*get-all-marked-decomposition M''*)
  using *assms* **by** (*induct M' arbitrary*: *M'' rule*: *marked-lit-list-induct*) *auto*

lemma *get-all-marked-decomposition-not-is-marked-length*:
  assumes ∀ *l* ∈ *set M'*. ¬*is-marked l*
  shows *1* + *length* (*get-all-marked-decomposition* (*Propagated* (−*L*) *P* # *M*))
    = *length* (*get-all-marked-decomposition* (*M'* @ *Marked L l* # *M*))
 using *assms get-all-marked-decomposition-remove-unmark-ssed-length* **by** *fastforce*

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

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

lemma *get-all-marked-decomposition-hd-hd*:
  assumes *get-all-marked-decomposition Ls* = (*M*, *C*) # (*M0*, $M0'$) # *l*
  shows *tl M* = $M0'$ @ *M0* ∧ *is-marked* (*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* = *Marked L level*
    have *Ls* = $M0'$ @ *M0*
      using *g a* **by** (*force intro*: *get-all-marked-decomposition-decomp*)
    then have *tl M* = $M0'$ @ *M0* ∧ *is-marked* (*hd M*) **using** *g a* **by** *auto*
  }
  moreover {
    fix *L P*
    assume *a*: *a* = *Propagated L P*
    have *tl M* = $M0'$ @ *M0* ∧ *is-marked* (*hd M*)
      using *IH Cons.prems* **unfolding** *a* **by** (*cases get-all-marked-decomposition Ls*) *auto*
  }
  ultimately show *?case* **by** (*cases a*) *auto*
qed

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

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

lemma *get-all-marked-decomposition-exists-prepend'*:
  assumes (*a*, *b*) ∈ *set* (*get-all-marked-decomposition M*)
  obtains *c* **where** *M* = *c* @ *b* @ *a*

using *assms* **apply** (*induct M rule*: *marked-lit-list-induct*)
  apply *auto*[*1*]
by (*rename-tac L′ m xs*, *case-tac hd* (*get-all-marked-decomposition xs*),
  auto dest!: *get-all-marked-decomposition-decomp simp add*: *list.set-sel*(*2*))+

lemma *union-in-get-all-marked-decomposition-is-subset*:
assumes $(a, b) \in set$ (*get-all-marked-decomposition M*)
shows $set\ a \cup set\ b \subseteq set\ M$
using *assms* **by** *force*

lemma *Marked-cons-in-get-all-marked-decomposition-append-Marked-cons*:
$\exists M1\ M2.$ (*Marked K i* # *M1*, *M2*) $\in$ *set* (*get-all-marked-decomposition* (*c* @ *Marked K i* # *c′*))
apply (*induction c rule*: *marked-lit-list-induct*)
  apply *auto*[*2*]
apply (*rename-tac L m xs*,
    case-tac hd (*get-all-marked-decomposition* (*xs* @ *Marked K i* # *c′*)))
apply (*case-tac get-all-marked-decomposition* (*xs* @ *Marked K i* # *c′*))
by *auto*

definition *all-decomposition-implies* :: *′a literal multiset set*
$\Rightarrow$ ((*′a*, *′l*, *′m*) *marked-lit list* $\times$ (*′a*, *′l*, *′m*) *marked-lit list*) *list* $\Rightarrow$ *bool* **where**
all-decomposition-implies N S
$\longleftrightarrow$ ($\forall$ (*Ls*, *seen*) $\in$ *set S*. *unmark-l Ls* $\cup$ *N* $\models$*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*)]
  $\longleftrightarrow$ *unmark-l Ls* $\cup$ *N* $\models$*ps unmark-l seen*
unfolding *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-append*[*iff*]:
all-decomposition-implies N (*S* @ *S′*)
  $\longleftrightarrow$ (*all-decomposition-implies N S* $\wedge$ *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′*)
  $\longleftrightarrow$ (*all-decomposition-implies N* [(*Ls*, *seen*)] $\wedge$ *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′*) $\longleftrightarrow$
  (*unmark-l* (*fst l*) $\cup$ *N* $\models$*ps unmark-l* (*snd l*) $\wedge$
    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-marked-decomposition M*)
shows *N* $\cup$ {*unmark L* |*L*. *is-marked L* $\wedge$ *L* $\in$ *set M*}
  $\models$*ps unmark* ‘ $\bigcup$(*set* ‘ *snd* ‘ *set* (*get-all-marked-decomposition M*))
using *assms*
proof (*induct length* (*get-all-marked-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-marked-decomposition M) ≤ 1
      | (gt1) length (get-all-marked-decomposition M) > 1
      by arith
    then show ?case
      proof cases
        case le1
        then obtain a b where g: get-all-marked-decomposition M = (a, b) # []
          by (cases get-all-marked-decomposition M) auto
        moreover {
          assume a = []
          then have ?thesis using Suc.prems g by auto
        }
        moreover {
          assume l: length a = 1 and m: is-marked (hd a) and hd: hd a ∈ set M
          then have unmark (hd a) ∈ {unmark L |L. is-marked L ∧ L ∈ set M} by auto
          then have H: unmark-l a ∪ N ⊆ N ∪ {unmark L |L. is-marked 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-marked-decomposition-length-1-fst-empty-or-length-1 by blast
      next
        case gt1
        then obtain Ls0 seen0 M' where
          Ls0: get-all-marked-decomposition M = (Ls0, seen0) # get-all-marked-decomposition M' and
          length': length (get-all-marked-decomposition M') = n and
          M'-in-M: set M' ⊆ set M
          using length by (induct M rule: marked-lit-list-induct) (auto simp: subset-insertI2)
        let ?d = ⋃(set ` snd ` set (get-all-marked-decomposition M'))
        let ?unM = {unmark L |L. is-marked L ∧ L ∈ set M}
        let ?unM' = {unmark L |L. is-marked L ∧ L ∈ set M'}
        {
          assume n = 0
          then have get-all-marked-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-marked-decomposition M' = (Ls1, seen1) # l
            using length' by (induct M' rule: marked-lit-list-induct) auto

          have all-decomposition-implies N (get-all-marked-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: $N \cup$ *?unM* $\models$*ps unmark-s ?d* **by** *auto*
have *is-marked* (*hd Ls0*) **and** *LS*: *tl Ls0* = *seen1* @ *Ls1*
using *get-all-marked-decomposition-hd-hd*[*of M*] **unfolding** *Ls0 Ls1* **by** *auto*

have *LSM*: *seen1* @ *Ls1* = M' **using** *get-all-marked-decomposition-decomp*[*of* M'] *Ls1* **by** *auto*
have M': *set* M' = *?d* $\cup$ $\{L \mid L.$ *is-marked* $L \wedge L \in$ *set* $M'\}$
using *get-all-marked-decomposition-snd-union* **by** *auto*

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

have *l*: *unmark* ‘ (*?d* $\cup$ $\{L \mid L.$ *is-marked* $L \wedge L \in$ *set* $M'\}$) = *unmark-s ?d* $\cup$ *?unM'*
by *auto*
have $N \cup$ *?unM'* $\models$*ps unmark* ‘ (*?d* $\cup$ $\{L \mid L.$ *is-marked* $L \wedge L \in$ *set* $M'\}$)
unfolding *l* **using** *N* **by** (*auto simp*: *all-in-true-clss-clss*)
then have *t*: $N \cup$ *?unM'* $\models$*ps unmark-l* (*tl Ls0*)
using M' **unfolding** *LS LSM* **by** *auto*
then have $N \cup$ *?unM* $\models$*ps unmark-l* (*tl Ls0*)
using *M'-in-M true-clss-clss-subset*[*OF - t*, *of N* $\cup$ *?unM*] **by** *auto*
then have $N \cup$ *?unM* $\models$*ps unmark-l Ls0*
using *hd-Ls0* **by** (*cases Ls0*) *auto*

moreover have *unmark-l Ls0* $\cup$ *N* $\models$*ps unmark-l seen0*
using *decomp* **unfolding** *Ls0* **by** *simp*
moreover have $\bigwedge M$ *Ma*. (*M*::'*a literal multiset set*) $\cup$ *Ma* $\models$*ps M*
by (*simp add*: *all-in-true-clss-clss*)
ultimately have Ψ: $N \cup$ *?unM* $\models$*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* $\cup$ *?d*) = *unmark-l seen0* $\cup$ *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-marked-decomposition M*)
shows $N \cup \{$*unmark L* $\mid L.$ *is-marked* $L \wedge L \in$ *set* $M\}$ $\models$*ps unmark-l M*
(**is** *?I* $\models$*ps ?A*)
proof –
have *?I* $\models$*ps unmark-s* $\{L \mid L.$ *is-marked* $L \wedge L \in$ *set* $M\}$
by (*auto intro*: *all-in-true-clss-clss*)
moreover have *?I* $\models$*ps unmark* ‘ $\bigcup$(*set* ‘ *snd* ‘ *set* (*get-all-marked-decomposition M*))
using *all-decomposition-implies-trail-is-implied assms* **by** *blast*
ultimately have $N \cup \{$*unmark m* $\mid m.$ *is-marked* $m \wedge m \in$ *set* $M\}$
$\models$*ps unmark* ‘ $\bigcup$(*set* ‘ *snd* ‘ *set* (*get-all-marked-decomposition M*))

$\cup$ *unmark* ‘ $\{m \mid m.$ *is-marked* $m \wedge m \in$ *set* $M\}$
by *blast*
then show *?thesis*
by (*metis* (*no-types*) *get-all-marked-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*

14.4 Negation of Clauses

definition *CNot* :: *'v clause* $\Rightarrow$ *'v clauses* **where**
CNot $\psi = \{ \{\#-L\#\} \mid L.\ L \in\# \psi \}$

lemma *in-CNot-uminus*[*iff*]:
shows $\{\#L\#\} \in$ *CNot* $\psi \longleftrightarrow -L \in\# \psi$
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* φ
using *assms* **unfolding** *total-over-m-def total-over-set-def true-clss-def true-cls-def CNot-def*
apply *clarify*
by (*rename-tac x L*, *case-tac L*) (*force intro*: *pos-lit-in-atms-of neg-lit-in-atms-of*)+

lemma *total-not-CNot*:

assumes *total-over-m I* $\{\varphi\}$ **and** $\neg I \models s$ *CNot* φ
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-of-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 *consistent-CNot-not-tautology*:
consistent-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-interp I* **and**
I: $I \models s\ A$
let $?I = I \cup \{Pos\ P|P.\ P \in atms\text{-}of\ CC \wedge P \notin atm\text{-}of\ `\ I\}$
have *cons'*: *consistent-interp ?I*
using *cons* **unfolding** *consistent-interp-def*
by (*auto simp*: *uminus-lit-swap atms-of-def rev-image-eqI*)
have *I'*: $?I \models s\ A$
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 $?I \models CC + \{\#L\#\}$ **using** *CC-L cons' I'* **unfolding** *true-clss-cls-def* **by** *blast*
moreover
have $?I \models s\ CNot\ CC$ **using** *CNot-CC cons' I' tot-CNot* **unfolding** *true-clss-clss-def* **by** *auto*
then have $\neg A \models p\ CC$
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 $\neg ?I \models CC$ **using** ‹$?I \models s\ CNot\ CC$› *cons' consistent-CNot-not* **by** *blast*
ultimately have $?I \models \{\#L\#\}$ **by** *blast*
then show $I \models \{\#L\#\}$
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*: $L\ \#\ M \models as\ CNot\ A$ **and** *LA*: $lit\text{-}of\ L \notin\#\ A\ -lit\text{-}of\ L \notin\#\ A$
shows $M \models as\ CNot\ A$
using *LM* **unfolding** *true-annots-def Ball-def*
proof (*intro allI impI*)
fix *l*
assume *H*: $\forall x.\ x \in CNot\ A \longrightarrow L\ \#\ M \models a\ x$ **and** *l*: $l \in CNot\ A$
then have $L\ \#\ M \models a\ l$ **by** *auto*

then show $M \models a\ l$ **using** $LA\ l$ **by** (*cases* L) (*auto simp*: *CNot-def*)
qed

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:
$A \cup \{B\} \models ps\ \{\{\#\}\} \longleftrightarrow A \models ps\ CNot\ B$
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 $a \# M' \models a\ D$ **and** *atm-of* (*lit-of* a) $\notin$ *atms-of* D
shows $M' \models a\ D$
using *assms true-cls-remove-hd-if-notin-vars* **unfolding** *true-annot-def* **by** *auto*

lemma *true-annot-remove-if-notin-vars*:
assumes $M\ @\ M' \models a\ D$ **and** $\forall x \in$ *atms-of* D. $x \notin$ *atm-of* ` *lits-of-l* M
shows $M' \models a\ D$
using *assms* **apply** (*induct* M, *simp*)
using *true-annot-remove-hd-if-notin-vars* **by** *force+*

lemma *true-annots-remove-if-notin-vars*:
assumes $M\ @\ M' \models as\ D$ **and** $\forall x \in$ *atms-of-ms* D. $x \notin$ *atm-of* ` *lits-of-l* M
shows $M' \models as\ D$ **unfolding** *true-annots-def*
using *assms true-annot-remove-if-notin-vars*[*of* $M\ M'$]
unfolding *true-annots-def atms-of-ms-def* **by** *force*

lemma *all-variables-defined-not-imply-cnot*:
assumes
$\forall s \in$ *atms-of-ms* $\{B\}$. $s \in$ *atm-of* ` *lits-of-l* A **and**
$\neg A \models a\ B$
shows $A \models as\ CNot\ B$
unfolding *true-annot-def true-annots-def Ball-def CNot-def true-lit-def*
proof (*clarify*, *rule ccontr*)
fix L
assume LB: $L \in\# B$ **and** $\neg$ *lits-of-l* $A \models l - L$
then have *atm-of* $L \in$ *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* $A \lor -L \in$ *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** ‹$\neg$ *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 ($A\ \#\cup\ B$) = *CNot* $A \cup$ *CNot* B
unfolding *CNot-def* **by** *auto*

14.5 Other

abbreviation *no-dup* $L \equiv$ *distinct* (*map* (λl. *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 image-iff* **unfolding** *defined-lit-def* **by** *fastforce*
  then show *?case*
    using *a1* **by** *simp*
qed

lemma *distinct-get-all-marked-decomposition-no-dup*:
  assumes (*a*, *b*) $\in$ *set* (*get-all-marked-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* $\notin$# *A*
  and *no-dup* (*L # M*)
  shows *M* $\models$*as CNot A*
proof $-$
  have $\forall$ *l* $\in$# *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

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 $[\![$*?N* $\models$*ps ?B*; *?A* $\subseteq$ *?B*$]\!]$ $\Longrightarrow$ *?N* $\models$*ps ?A*

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* ⇒ *'a clause* ⇒ *bool* (**infix** ⊨*pm 50*) **where**
I ⊨*pm C* ≡ *set-mset I* ⊨*p C*

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

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

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

Other definition using *Union-mset*

lemma *atms-of-mm U* ≡ *set-mset* (⋃# *image-mset* (*image-mset atm-of*) *U*)
  unfolding *atms-of-ms-def* **by** (*auto simp*: *atms-of-def*)

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

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

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*

14.6 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* ⇒ *'v clause* **and**
    insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
    remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls*
  assumes
    insert-cls[*simp*]: *mset-cls* (*insert-cls L C*) = *mset-cls C* + {#*L*#} **and**
    remove-lit[*simp*]: *mset-cls* (*remove-lit L C*) = *remove1-mset L* (*mset-cls C*)
begin
end

locale *raw-ccls-union* =
  fixes
    mset-cls:: *'cls* ⇒ *'v clause* **and**
    union-cls :: *'cls* ⇒ *'cls* ⇒ *'cls* **and**

insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls*
assumes
insert-ccls[*simp*]: *mset-cls* (*insert-cls L C*) = *mset-cls C* + {#*L*#} **and**
mset-ccls-union-cls[*simp*]: *mset-cls* (*union-cls C D*) = *mset-cls C* #∪ *mset-cls D* **and**
remove-clit[*simp*]: *mset-cls* (*remove-lit L C*) = *remove1-mset L* (*mset-cls C*)
begin
end

Instantiation of the previous locale, in an unnamed context to avoid polluating with simp rules

context
begin
interpretation *list-cls*: *raw-cls mset*
op # *remove1*
by *unfold-locales* (*auto simp*: *union-mset-list ex-mset*)

interpretation *cls-cls*: *raw-cls id*
$\lambda L\ C.\ C$ + {#*L*#} *remove1-mset*
by *unfold-locales* (*auto simp*: *union-mset-list*)

interpretation *list-cls*: *raw-ccls-union mset*
union-mset-list
op # *remove1*
by *unfold-locales* (*auto simp*: *union-mset-list ex-mset*)

interpretation *cls-cls*: *raw-ccls-union id*
op #∪ $\lambda L\ C.\ C$ + {#*L*#} *remove1-mset*
by *unfold-locales* (*auto simp*: *union-mset-list*)
end

Over the abstract clauses, we have the following properties:

- We can insert a clause
- We can take the union (used only in proofs for the definition of *clauses*)
- there is an operator indicating whether the abstract clause is contained or not
- if a concrete clause is contained the abstract clauses, then there is an abstract clause

locale *raw-clss* =
raw-cls mset-cls insert-cls remove-lit
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* +
fixes
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss*
assumes
insert-clss[*simp*]: *mset-clss* (*insert-clss L C*) = *mset-clss C* + {#*mset-cls L*#} **and**
union-clss[*simp*]: *mset-clss* (*union-clss C D*) = *mset-clss C* + *mset-clss D* **and**

mset-clss-union-clss[*simp*]: *mset-clss* (*insert-clss* C' D) = {#*mset-cls* C'#} + *mset-clss* D **and**
in-clss-mset-clss[*dest*]: *in-clss a C* $\Longrightarrow$ *mset-cls a* $\in$# *mset-clss C* **and**
in-mset-clss-exists-preimage: b $\in$# *mset-clss C* $\Longrightarrow$ $\exists$ b'. *in-clss* b' C $\wedge$ *mset-cls* b' = b **and**
remove-from-clss-mset-clss[*simp*]:
mset-clss (*remove-from-clss a C*) = *mset-clss C* − {#*mset-cls a*#} **and**
in-clss-union-clss[*simp*]:
in-clss a (*union-clss C D*) $\longleftrightarrow$ *in-clss a C* $\vee$ *in-clss a D*
begin

end

experiment
begin
fun *remove-first* **where**
remove-first - [] = [] |
remove-first C (C' # L) = (*if mset C* = *mset* C' *then L else* C' # *remove-first C L*)

lemma *mset-map-mset-remove-first*:
mset (*map mset* (*remove-first a C*)) = *remove1-mset* (*mset a*) (*mset* (*map mset C*))
by (*induction C*) (*auto simp*: *ac-simps remove1-mset-single-add*)

interpretation *clss-clss*: *raw-clss id* λL C. C + {#L#} *remove1-mset*
id op + *op* $\in$# λL C. C + {#L#} *remove1-mset*
by *unfold-locales* (*auto simp*: *ac-simps*)

interpretation *list-clss*: *raw-clss mset*
op # *remove1* λL. *mset* (*map mset L*) *op* @ λL C. $L \in$ *set C op* #
remove-first
by *unfold-locales* (*auto simp*: *ac-simps union-mset-list mset-map-mset-remove-first ex-mset*)
end

end
theory *CDCL-WNOT-Measure*
imports *Main*
begin

15 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**
μ_C *s b M* $\equiv$ ($\sum$ *i=0..<length M*. *M!i* $*$ *b*^ (*s* +*i* − *length M*))

lemma μ_C-*nil*[*simp*]:
μ_C *s b* [] = *0*
unfolding μ_C-*def* **by** *auto*

lemma μ_C-*single*[*simp*]:
μ_C *s b* [*L*] = *L* $*$ *b* ^ (*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 ... $= (\sum i{=}0..{<}1.\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M)))$
$+ (\sum i{=}1..{<}length\ (L\#M).\ (L\#M)!i * b\hat{}\ (s + i - length\ (L\#M)))$
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 ... $= (\sum i{=}0..{<}length\ (M).\ M!i * b\hat{}\ (s + i - length\ M))$
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 ... $= (\sum i{=}0..{<}\ length\ M.\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
$+ (\sum i{=}length\ M..{<}length\ (M@M').\ (M@M')!i * b\hat{}\ (s + i - length\ (M@M')))$
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 $\mu_C\ s\ b\ M \geq b\ \hat{}\ (s - length\ M)$
  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 $\mu_C\ s\ b\ M < b\hat{}s$
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 $\mu_C\ s\ b\ M = 0$ **unfolding** μ_C*-def* **by** *auto*
      then show *?thesis* **using** ‹$b > 0$› **by** *auto*
    next
      case *b*
      have $\forall\ i \in \{0..<length\ M\}.\ M!i * b\hat{}\ (s +i - length\ M) \leq (b-1) * b\hat{}\ (s +i - length\ M)$
        using *M-le* ‹$b > 1$› **by** *auto*
      then have $\mu_C\ s\ b\ M \leq\ (\sum i=0..<length\ M.\ (b-1) * b\hat{}\ (s +i - length\ M))$
         using ‹$M\neq[]$› ‹$b>0$› **unfolding** μ_C*-def* **by** (*auto intro*: *setsum-mono*)
      also
       have $\forall\ i \in \{0..<length\ M\}.\ (b-1) * b\hat{}\ (s +i - length\ M) = (b-1) * b\hat{}\ i * b\hat{}(s - length\ M)$
         by (*metis Nat.add-diff-assoc2 add.commute assms(4) mult.assoc power-add*)
       then have $(\sum i=0..<length\ M.\ (b-1) * b\hat{}\ (s +i - length\ M))$
         $= (\sum i=0..<length\ M.\ (b-1)* b\hat{}\ i * b\hat{}(s - length\ M))$
         by (*auto simp add*: *ac-simps*)
      also have ... $= (\sum i=0..<length\ M.\ b\hat{}\ i) * b\hat{}(s - length\ M) * (b-1)$
         by (*simp add*: *setsum-left-distrib setsum-right-distrib ac-simps*)
      finally have $\mu_C\ s\ b\ M \leq (\sum i=0..<length\ M.\ b\hat{}\ i) * (b-1) * b\hat{}(s - length\ M)$
        by (*simp add*: *ac-simps*)

      also
        have $(\sum i=0..<length\ M.\ b\hat{}\ i)* (b-1) = b\ \hat{}\ (length\ M) - 1$
          using *sum-of-powers*[*of b length M*] ‹$b>1$›

by (*auto simp add*: *ac-simps*)
finally have μ_C *s b M* $\leq$ (*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* ^ *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* $<$ *length M*. *M*!*i* $<$ *b* **and**
s $\geq$ *length M*
b $>$ *0*
shows μ_C *s b M* $<$ *b* ^ *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 *length M* $\leq$ *s*
shows μ_C *s 0 M* $\leq$ *M*!*0*
proof $-$
{
assume *s* = *length M*
moreover {
fix *n*
have ($\sum$ *i*=*0*..<*n*. *M* ! *i* $*$ (*0*::*nat*) ^ *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 *length M* $<$ *s*
then have μ_C *s 0 M* = *0* **unfolding** μ_C-*def* **by** *auto***}**
ultimately show *?thesis* **using** *assms* **unfolding** μ_C-*def* **by** *linarith*
qed

end
theory *CDCL-NOT*
imports *CDCL-Abstract-Clause-Representation List-More Wellfounded-More CDCL-WNOT-Measure Partial-Annotated-Clausal-Logic*
begin

16 NOT's CDCL

16.1 Auxiliary Lemmas and Measure

lemma *no-dup-cannot-not-lit-and-uminus*:
$no\text{-}dup\ M \Longrightarrow -\ lit\text{-}of\ xa = lit\text{-}of\ x \Longrightarrow x \in set\ M \Longrightarrow xa \notin set\ M$
by (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id'*)

lemma *atms-of-ms-single-atm-of*[*simp*]:
$atms\text{-}of\text{-}ms\ \{unmark\ L\ |L.\ P\ L\} = atm\text{-}of\ `\ \{lit\text{-}of\ L\ |L.\ P\ L\}$
unfolding *atms-of-ms-def* **by** *force*

lemma *atms-of-uminus-lit-atm-of-lit-of*:
$atms\text{-}of\ \{\#\ -lit\text{-}of\ x.\ x \in\#\ A\#\} = atm\text{-}of\ `\ (lit\text{-}of\ `\ (set\text{-}mset\ A))$
unfolding *atms-of-def* **by** (*auto simp add*: *Fun.image-comp*)

lemma *atms-of-ms-single-image-atm-of-lit-of*:
$atms\text{-}of\text{-}ms\ (unmark\text{-}s\ A) = atm\text{-}of\ `\ (lit\text{-}of\ `\ A)$
unfolding *atms-of-ms-def* **by** *auto*

16.2 Initial definitions

16.2.1 The state

We define here an abstraction over operation on the state we are manipulating.

locale *dpll-state-ops* =
raw-clss mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
for
$mset\text{-}cls :: 'cls \Rightarrow 'v\ clause$ **and**
$insert\text{-}cls :: 'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
$remove\text{-}lit :: 'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
$mset\text{-}clss :: 'clss \Rightarrow 'v\ clauses$ **and**
$union\text{-}clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
$in\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**
$insert\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
$remove\text{-}from\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ +
fixes
$trail :: 'st \Rightarrow ('v, unit, unit)\ marked\text{-}lits$ **and**
$raw\text{-}clauses :: 'st \Rightarrow 'clss$ **and**
$prepend\text{-}trail :: ('v, unit, unit)\ marked\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$
begin

notation *insert-cls* (**infix** !++ *50*)

notation *in-clss* (**infix** !∈! *50*)

notation *union-clss* (**infix** ⊕ *50*)
notation *insert-clss* (**infix** !++! *50*)

abbreviation $clauses_{NOT}$ **where**
$clauses_{NOT}$ *S* ≡ *mset-clss* (*raw-clauses S*)

end

locale *dpll-state* =
dpll-state-ops mset-cls insert-cls remove-lit — related to each clause
mset-clss union-clss in-clss insert-clss remove-from-clss — related to the clauses

trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ — related to the state
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* +
assumes
trail-prepend-trail[*simp*]:
⋀*st L*. *undefined-lit* (*trail st*) (*lit-of L*) ⟹ *trail* (*prepend-trail L st*) = *L* # *trail st*
and
tl-trail[*simp*]: *trail* (*tl-trail S*) = *tl* (*trail S*) **and**
$trail\text{-}add\text{-}cls_{NOT}$[*simp*]: ⋀*st C*. *no-dup* (*trail st*) ⟹ *trail* ($add\text{-}cls_{NOT}$ *C st*) = *trail st* **and**
$trail\text{-}remove\text{-}cls_{NOT}$[*simp*]: ⋀*st C*. *trail* ($remove\text{-}cls_{NOT}$ *C st*) = *trail st* **and**

clauses-prepend-trail[*simp*]:
⋀*st L*. *undefined-lit* (*trail st*) (*lit-of L*) ⟹
$clauses_{NOT}$ (*prepend-trail L st*) = $clauses_{NOT}$ *st*
and
clauses-tl-trail[*simp*]: ⋀*st*. $clauses_{NOT}$ (*tl-trail st*) = $clauses_{NOT}$ *st* **and**
$clauses\text{-}add\text{-}cls_{NOT}$[*simp*]:
⋀*st C*. *no-dup* (*trail st*) ⟹ $clauses_{NOT}$ ($add\text{-}cls_{NOT}$ *C st*) = {#*mset-cls C*#} + $clauses_{NOT}$ *st*
and
$clauses\text{-}remove\text{-}cls_{NOT}$[*simp*]:
⋀*st C*. $clauses_{NOT}$ ($remove\text{-}cls_{NOT}$ *C st*) = *removeAll-mset* (*mset-cls C*) ($clauses_{NOT}$ *st*)
begin

function $reduce\text{-}trail\text{-}to_{NOT}$ :: *'a list* ⇒ *'st* ⇒ *'st* **where**
$reduce\text{-}trail\text{-}to_{NOT}$ *F S* =
(*if length* (*trail S*) = *length F* ∨ *trail S* = [] *then S else* $reduce\text{-}trail\text{-}to_{NOT}$ *F* (*tl-trail S*))
by *fast+*
termination by (*relation measure* (λ(-, *S*). *length* (*trail S*))) *auto*
declare $reduce\text{-}trail\text{-}to_{NOT}$.*simps*[*simp del*]

lemma
shows
reduce-trail-to$_{NOT}$*-nil*[*simp*]: *trail S* = [] $\Longrightarrow$ *reduce-trail-to*$_{NOT}$ *F S* = *S* **and**
reduce-trail-to$_{NOT}$*-eq-length*[*simp*]: *length* (*trail S*) = *length F* $\Longrightarrow$ *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*) $\neq$ *length F* $\Longrightarrow$ *trail S* $\neq$ [] $\Longrightarrow$
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*) $\geq$ *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*)

abbreviation *trail-weight* **where**
trail-weight S $\equiv$ *map* ((λ*l*. *1* + *length l*) *o snd*) (*get-all-marked-decomposition* (*trail S*))

definition *state-eq*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* (**infix** ∼ *50*) **where**
$S \sim T \longleftrightarrow$ *trail* S = *trail* T ∧ *clauses*$_{NOT}$ S = *clauses*$_{NOT}$ T

lemma *state-eq*$_{NOT}$*-ref*[*simp*]:
  $S \sim S$
  unfolding *state-eq*$_{NOT}$*-def* **by** *auto*

lemma *state-eq*$_{NOT}$*-sym*:
  $S \sim T \longleftrightarrow T \sim S$
  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 *trail-eq-reduce-trail-to*$_{NOT}$*-eq*:
  trail S = *trail* $T \Longrightarrow$ *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 tl-trail reduce-trail-to*$_{NOT}$*-eq-length reduce-trail-to*$_{NOT}$*-length-ne reduce-trail-to*$_{NOT}$*-nil*)

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

lemma *trail-reduce-trail-to*$_{NOT}$*-add-cls*$_{NOT}$[*simp*]:
  no-dup (*trail* S) $\Longrightarrow$
    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' @ *Marked* K () # $F \Longrightarrow$
     trail (*reduce-trail-to*$_{NOT}$ F (*tl-trail* S)) = F
  apply (*rule reduce-trail-to*$_{NOT}$*-skip-beginning*[*of - tl* (F' @ *Marked* K () # [])])
  by (*cases* F') (*auto simp add*:*tl-append reduce-trail-to*$_{NOT}$*-skip-beginning*)

lemma *reduce-trail-to*$_{NOT}$*-length*:
  length M = *length* $M' \Longrightarrow$ *reduce-trail-to*$_{NOT}$ M S = *reduce-trail-to*$_{NOT}$ M' S
  apply (*induction M S arbitrary*: *rule*: *reduce-trail-to*$_{NOT}$*.induct*)
  by (*simp add*: *reduce-trail-to*$_{NOT}$*.simps*)

end

16.2.2 Definition of the operation

locale *propagate-ops* =
  dpll-state mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  for
    mset-cls:: *'cls* ⇒ *'v clause* **and**
    insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
    remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
    mset-clss:: *'clss* ⇒ *'v clauses* **and**
    union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
    in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
    insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
    remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
    trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
    raw-clauses :: *'st* ⇒ *'clss* **and**
    prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
    tl-trail :: *'st* ⇒*'st* **and**
    $add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
    $remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* +
  fixes
    propagate-cond :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *bool*
begin
inductive $propagate_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **where**
$propagate_{NOT}$[*intro*]: C + {#L#} ∈# $clauses_{NOT}$ S ⟹ *trail* S ⊨*as CNot* C
    ⟹ *undefined-lit* (*trail* S) L
    ⟹ *propagate-cond* (*Propagated* L ()) S
    ⟹ T ∼ *prepend-trail* (*Propagated* L ()) S
    ⟹ $propagate_{NOT}$ S T
inductive-cases $propagate_{NOT}E$[*elim*]: $propagate_{NOT}$ S T

end

locale *decide-ops* =
  dpll-state mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  for
    mset-cls:: *'cls* ⇒ *'v clause* **and**
    insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
    remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
    mset-clss:: *'clss* ⇒ *'v clauses* **and**
    union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
    in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
    insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
    remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
    trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
    raw-clauses :: *'st* ⇒ *'clss* **and**
    prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
    tl-trail :: *'st* ⇒*'st* **and**
    $add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
    $remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st*
begin

inductive $decide_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$decide_{NOT}[intro]$: $undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$
$\Longrightarrow T \sim prepend\text{-}trail\ (Marked\ L\ ())\ S$
$\Longrightarrow decide_{NOT}\ S\ T$

inductive-cases $decide_{NOT}E[elim]$: $decide_{NOT}\ S\ S'$
end

locale *backjumping-ops* =
dpll-state mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
mset-cls:: $'cls \Rightarrow 'v\ clause$ **and**
insert-cls :: $'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
remove-lit :: $'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
mset-clss:: $'clss \Rightarrow 'v\ clauses$ **and**
union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
in-clss :: $'cls \Rightarrow 'clss \Rightarrow bool$ **and**
insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
remove-from-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
trail :: $'st \Rightarrow ('v,\ unit,\ unit)\ marked\text{-}lits$ **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v,\ unit,\ unit)\ marked\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ +
fixes
backjump-conds :: $'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
begin

inductive *backjump* **where**
$trail\ S = F' @ Marked\ K\ ()\#\ F$
$\Longrightarrow T \sim prepend\text{-}trail\ (Propagated\ L\ ())\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ S)$
$\Longrightarrow C \in\#\ clauses_{NOT}\ S$
$\Longrightarrow trail\ S \models as\ CNot\ C$
$\Longrightarrow undefined\text{-}lit\ F\ L$
$\Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S))$
$\Longrightarrow clauses_{NOT}\ S \models pm\ C' + \{\#L\#\}$
$\Longrightarrow F \models as\ CNot\ C'$
$\Longrightarrow backjump\text{-}conds\ C\ C'\ L\ S\ T$
$\Longrightarrow backjump\ S\ T$
inductive-cases *backjumpE*: *backjump S T*

The condition $atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S)$ is not implied by the the condition $clauses_{NOT}\ S \models pm\ C' + \{\#L\#\}$ (no negation).

end

16.3 DPLL with backjumping

locale *dpll-with-backjumping-ops* =
propagate-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *propagate-conds* +
decide-ops mset-cls insert-cls remove-lit

mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ +
backjumping-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ *backjump-conds*
for
mset-cls:: $'cls \Rightarrow$ $'v$ *clause* **and**
insert-cls :: $'v$ *literal* $\Rightarrow$ $'cls \Rightarrow$ $'cls$ **and**
remove-lit :: $'v$ *literal* $\Rightarrow$ $'cls \Rightarrow$ $'cls$ **and**
mset-clss:: $'clss \Rightarrow$ $'v$ *clauses* **and**
union-clss :: $'clss \Rightarrow$ $'clss \Rightarrow$ $'clss$ **and**
in-clss :: $'cls \Rightarrow$ $'clss \Rightarrow$ *bool* **and**
insert-clss :: $'cls \Rightarrow$ $'clss \Rightarrow$ $'clss$ **and**
remove-from-clss :: $'cls \Rightarrow$ $'clss \Rightarrow$ $'clss$ **and**
trail :: $'st \Rightarrow$ ($'v$, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: $'st \Rightarrow$ $'clss$ **and**
prepend-trail :: ($'v$, *unit*, *unit*) *marked-lit* $\Rightarrow$ $'st \Rightarrow$ $'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-cls$_{NOT}$ :: $'cls \Rightarrow$ $'st \Rightarrow$ $'st$ **and**
remove-cls$_{NOT}$ :: $'cls \Rightarrow$ $'st \Rightarrow$ $'st$ **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*, *unit*) *marked-lit* $\Rightarrow$ $'st \Rightarrow$ *bool* +
assumes
bj-can-jump:
$\bigwedge S\ C\ F'\ K\ F\ L.$
inv S $\Longrightarrow$
no-dup (*trail S*) $\Longrightarrow$
trail S = F' @ *Marked 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' @ *Marked K* () # *F*)) $\Longrightarrow$
clauses$_{NOT}$ *S* $\models$*pm* C' + {#*L*#} $\Longrightarrow$
F $\models$*as CNot* C' $\Longrightarrow$
$\neg$*no-step backjump S*
begin

We cannot add a like condition *atms-of* C' $\subseteq$ *atms-of-ms N* because to ensure that we can backjump even if the last decision variable has disappeared.

The part of the condition *atm-of L* $\in$ *atm-of* ‘ *lits-of-l* (F' @ *Marked K* () # *F*) is important, otherwise you are not sure that you can backtrack.

16.3.1 Definition

We define dpll with backjumping:

inductive *dpll-bj* :: $'st \Rightarrow$ $'st \Rightarrow$ *bool* **for** *S* :: $'st$ **where**
bj-decide$_{NOT}$: *decide*$_{NOT}$ *S* S' $\Longrightarrow$ *dpll-bj S* S' |
bj-propagate$_{NOT}$: *propagate*$_{NOT}$ *S* S' $\Longrightarrow$ *dpll-bj S* S' |
bj-backjump: *backjump S* S' $\Longrightarrow$ *dpll-bj S* S'

lemmas *dpll-bj-induct* = *dpll-bj.induct*[*split-format*(*complete*)]
thm *dpll-bj-induct*[*OF dpll-with-backjumping-ops-axioms*]
lemma *dpll-bj-all-induct*[*consumes 2*, *case-names decide*$_{NOT}$ *propagate*$_{NOT}$ *backjump*]:

fixes S T :: $'st$
assumes
dpll-bj S T **and**
inv S
$\bigwedge L\ T.$ *undefined-lit* (*trail* S) $L \Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* ($clauses_{NOT}$ S)
$\Longrightarrow T \sim$ *prepend-trail* (*Marked* 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-lit* (*trail* S) L
$\Longrightarrow T \sim$ *prepend-trail* (*Propagated* L ()) S
$\Longrightarrow P\ S\ T$ **and**
$\bigwedge C\ F'\ K\ F\ L\ C'\ T.\ C \in\#$ $clauses_{NOT}$ $S \Longrightarrow F'$ @ *Marked* K () # $F \models as$ *CNot* C
$\Longrightarrow$ *trail* $S = F'$ @ *Marked* K () # F
$\Longrightarrow$ *undefined-lit* F L
$\Longrightarrow$ *atm-of* $L \in$ *atms-of-mm* ($clauses_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (F' @ *Marked* K () # F))
$\Longrightarrow$ $clauses_{NOT}$ $S \models pm$ $C' + \{\#L\#\}$
$\Longrightarrow F \models as$ *CNot* C'
$\Longrightarrow T \sim$ *prepend-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*

16.3.2 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* $reduce\text{-}trail\text{-}to_{NOT}\text{-}skip\text{-}beginning$)

Valuations **lemma** *dpll-bj-sat-iff*:
assumes *dpll-bj* S T **and** *inv* S
shows $I \models sm$ $clauses_{NOT}$ $S \longleftrightarrow I \models sm$ $clauses_{NOT}$ 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-marked-decomposition* (*trail S*))
shows *all-decomposition-implies-m* ($clauses_{NOT}$ *T*) (*get-all-marked-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-marked-decomposition ?M′* = (*a*, *y*) # *l*
by (*cases get-all-marked-decomposition ?M′*) *fastforce+*
then have *M′*: *?M′* = *y* @ *a* **using** *get-all-marked-decomposition-decomp*[*of ?M′*] **by** *auto*
have *M*: *get-all-marked-decomposition* (*trail S*) = (*a*, *tl y*) # *l*
using *ay undef* **by** (*cases get-all-marked-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-marked-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-marked-decomposition F*)
using *decomp* **unfolding** *tr all-decomposition-implies-def*
by (*metis* (*no-types*, *lifting*) *get-all-marked-decomposition.simps*(*1*)
get-all-marked-decomposition-never-empty hd-Cons-tl insert-iff list.sel(*3*) *list.set*(*2*)
tl-get-all-marked-decomposition-skip-some)

obtain *a b li* **where** *F*: *get-all-marked-decomposition F* = (*a*, *b*) # *li*
by (*cases get-all-marked-decomposition F*) *auto*
have *F* = *b* @ *a*
using *get-all-marked-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

16.3.3 Termination

Using a proper measure **lemma** *length-get-all-marked-decomposition-append-Marked*:
length (*get-all-marked-decomposition* (*F'* @ *Marked K* () # *F*)) =
length (*get-all-marked-decomposition F'*)
\+ *length* (*get-all-marked-decomposition* (*Marked K* () # *F*))
− *1*
by (*induction F' rule*: *marked-lit-list-induct*) *auto*

lemma *take-length-get-all-marked-decomposition-marked-sandwich*:
take (*length* (*get-all-marked-decomposition F*))
(*map* (*f o snd*) (*rev* (*get-all-marked-decomposition* (*F'* @ *Marked K* () # *F*))))
=
map (*f o snd*) (*rev* (*get-all-marked-decomposition F*))

proof (*induction F' rule*: *marked-lit-list-induct*)
case *nil*

then show *?case* **by** *auto*
next
case (*marked K*)
then show *?case* **by** (*simp add*: *length-get-all-marked-decomposition-append-Marked*)
next
case (*proped L m F'*) **note** *IH* = *this*(*1*)
obtain *a b l* **where** *F'*: *get-all-marked-decomposition* (*F'* @ *Marked K* () # *F*) = (*a*, *b*) # *l*
by (*cases get-all-marked-decomposition* (*F'* @ *Marked K* () # *F*)) *auto*
have *length* (*get-all-marked-decomposition F*) − *length l* = *0*
using *length-get-all-marked-decomposition-append-Marked*[*of F' K F*]
unfolding *F'* **by** (*cases get-all-marked-decomposition F'*) *auto*
then show *?case*
using *IH* **by** (*simp add*: *F'*)
qed

lemma *length-get-all-marked-decomposition-length*:
length (*get-all-marked-decomposition M*) ≤ *1* + *length M*
by (*induction M rule*: *marked-lit-list-induct*) *auto*

lemma *length-in-get-all-marked-decomposition-bounded*:
assumes *i*:*i* ∈ *set* (*trail-weight S*)
shows *i* ≤ *Suc* (*length* (*trail S*))
proof −
obtain *a b* **where**
(*a*, *b*) ∈ *set* (*get-all-marked-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-marked-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-marked-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* ⇒ *'a list* ⇒ *nat* **where**
unassigned-lit N M ≡ *card* (*atms-of-ms N*) − *length M*
lemma *dpll-bj-trail-mes-increasing-prop*:
fixes *M* :: (*'v*, *unit*, *unit*) *marked-lits* **and** *N* :: *'v clauses*
assumes
dpll-bj S T **and**
inv S **and**
NA: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A* **and**
MA: *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *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*)

$> \mu_C$ (*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*) ⊆ *atms-of-ms A*
using $propagate_{NOT}$ *dpll-bj-atms-in-trail-in-set* *bj-*$propagate_{NOT}$ *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-marked-decomposition* (*trail S*) = (*a*, *b*) # *l*
by (*cases get-all-marked-decomposition* (*trail S*)) *auto*
have *b-le-M*: *length b* ≤ *length* (*trail S*)
using *get-all-marked-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*) ≤ *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* (*Marked 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* (*Marked L* () # (*trail S*))
using *defined-lit-map n-d undef-L* **by** *auto*
obtain *a b l* **where** *M*: *get-all-marked-decomposition* (*trail S*) = (*a*, *b*) # *l*
by (*cases get-all-marked-decomposition* (*trail S*)) *auto*

then have *length* (*Marked 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-marked-decomposition* (*trail S*) = (*a*, *b*) # *l*
by (*cases get-all-marked-decomposition* (*trail S*)) *auto*
have *b-le-M*: *length b* ≤ *length* (*trail S*)
using *get-all-marked-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*) $\leq$ (*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-marked-decomposition F* = (*a*, *b*) # *l*
by (*cases get-all-marked-decomposition F*) *auto*
then have *F* = *b* @ *a*
using *get-all-marked-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-marked-decomposition* (*F'* @ *Marked K* () # *F*)))
= *map* (λ*a*. *Suc* (*length* (*snd a*))) (*rev* (*get-all-marked-decomposition F*)) @ *rem*
using *take-length-get-all-marked-decomposition-marked-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-marked-decomposition* (*F'* @ *Marked K* () # *F*))
= *rev rem* @ *map* (λ*a*. *Suc* (*length* (*snd a*))) ((*get-all-marked-decomposition F*))
by (*simp add*: *rev-map*[*symmetric*] *rev-swap*)
have *length* (*rev rem* @ *map* (λ*a*. *Suc* (*length* (*snd a*))) (*get-all-marked-decomposition F*))
$\leq$ *Suc* (*card* (*atms-of-ms A*))
using *arg-cong*[*OF rem*, *of length*] *tr-S-le-A*
length-get-all-marked-decomposition-length[*of F'* @ *Marked K* () # *F*] *tr-S* **by** *auto*
moreover
{ **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* $\Longrightarrow$ *rev 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 $\forall$ *i*<*length rem*. *rev rem* ! *i* < *card* (*atms-of-ms A*) + *2*
using *tr-S-le-A length-in-get-all-marked-decomposition-bounded*[*of - S*] **unfolding** *tr-S*
by (*force simp add*: *o-def rem dest*!: *H intro*: *length-get-all-marked-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*) $\subseteq$ *atms-of-ms A* **and**
M-A: *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *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*) $\subseteq$ *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* $\Longrightarrow$ *length xs* − *Suc i* < *length xs*
    by *auto*
  then have *H*: *i*<*length xs* $\Longrightarrow$ *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-marked-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-marked-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*) ≤ *?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*
  ∧ *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** *wf ?A*)
proof (*rule wf-bounded-measure*[*of -*
    λ-. (*2* + *card* (*atms-of-ms A*))^(*1* + *card* (*atms-of-ms A*))
    λ*S*. 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*) ∈ *?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*) ⊆ *atms-of-ms A* **and**
  M-A: *atm-of* ‘ *lits-of-l* (*trail a*) ⊆ *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*) ⊆ *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-marked-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-marked-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

16.3.4 Normal Forms

We prove that given a normal form of DPLL, with some invariants, the 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* rules tells us that every variable in N has a value.
2. $\neg$ M $\models$*as* N tells us that there is conflict.
3. There is at least one decision in the trail (otherwise, M is a model of 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 literal multiset 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-marked-decomposition (trail S))
  shows unsatisfiable (set-mset (clauses_NOT S))
    ∨ (trail S ⊨asm clauses_NOT S ∧ satisfiable (set-mset (clauses_NOT S)))
proof −
  let ?N = set-mset (clauses_NOT S)
  let ?M = trail S
  consider
      (sat) satisfiable ?N and ?M ⊨as ?N
    | (sat') satisfiable ?N and ¬ ?M ⊨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 ∈ ?N and ¬?M ⊨a C using M unfolding true-annots-def by auto
      obtain I :: 'v literal set where
        I ⊨s ?N and
        cons: consistent-interp I and
        tot: total-over-m I ?N and
        atm-I-N: atm-of `I ⊆ 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-marked 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 dest!: is-marked-ex-Marked)

      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 Marked-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-marked l
  proof (rule ccontr)
    let ?O = {unmark L |L. is-marked 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-marked L ∧ L ∈ set ?M}
      = {unmark L |L. is-marked 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, unit) marked-lit list where
  M-K: ?M = F' @ Marked K () # F and
  nm: ∀f∈set F'. ¬is-marked f
  unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let ?K = Marked K () :: ('v, unit, unit) marked-lit
have ?K ∈ set ?M
  unfolding M-K by auto
let ?C = image-mset lit-of {#L∈#mset ?M. is-marked L ∧ L≠?K#} :: 'v literal multiset
let ?C' = set-mset (image-mset (λL::'v literal. {#L#}) (?C + unmark ?K))
have ?N ∪ {unmark L |L. is-marked 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-marked 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 ∪ (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-marked a ∧ a ≠ Marked K ()} =
          set (trail S) ∩ {L. is-marked L ∧ L ≠ Marked K ()}
         by auto
        then have tot': total-over-set I
           (atm-of ‘ lit-of ‘ (set ?M ∩ {L. is-marked L ∧ L ≠ Marked K ()}))
          using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
        { fix x :: ('v, unit, unit) marked-lit
          assume
            a3: lit-of x ∉ I and
            a1: x ∈ set ?M and
            a4: is-marked x and
            a5: x ≠ Marked 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 using ‹(K ∈ I ∧ −K ∉ I) ∨ (−K ∈ I ∧ K ∉ I)›
          by (auto dest!: H)
      qed
    moreover have F ⊨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-can-jump[of S F' K F C −K
       image-mset uminus (image-mset lit-of {# L :# mset ?M. is-marked L ∧ L ≠ Marked K ()#})]
       ‹C∈?N› n-s ‹?M ⊨as CNot C› bj-backjump inv ‹no-dup (trail S)› unfolding M-K by auto
      then show ?thesis by fast
  qed auto
qed

end

locale dpll-with-backjumping =
  dpll-with-backjumping-ops mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-cls_NOT remove-cls_NOT inv backjump-conds
    propagate-conds
```

for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *bool*
+
assumes *dpll-bj-inv*: $\bigwedge S\ T.$ *dpll-bj S T* ⟹ *inv S* ⟹ *inv T*
begin

lemma *rtranclp-dpll-bj-inv*:
assumes *dpll-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-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) $\subseteq$ A **and**
atm-of ' (*lits-of-l* (*trail* S)) $\subseteq$ A
shows *atm-of* ' (*lits-of-l* (*trail* T)) $\subseteq$ 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-marked-decomposition* (*trail* S))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ T) (*get-all-marked-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
$\wedge$ *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* $S\}$
$\subseteq$ $\{$(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** *?A* $\subseteq$ *?B*$^+$)
proof *standard*
fix x
assume *x-A*: $x \in$ *?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) $\subseteq$ *atms-of-ms* A **and**
atm-of ' *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
no-dup (*trail* S) **and**
inv S
using *x-A* **by** *auto*
then show $x \in$ *?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)) $\subseteq$ *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 $(U, T) \in$ *?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*
$\wedge$ *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*$\}$
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 $\Longrightarrow$ *inv S* $\Longrightarrow$ *I*$\models$*sextm clauses*$_{NOT}$ *S* $\longleftrightarrow$ *I*$\models$*sextm clauses*$_{NOT}$ *T*
by (*simp add*: *dpll-bj-clauses*)

lemma *rtranclp-dpll-bj-sat-ext-iff*:
dpll-bj** *S T* $\Longrightarrow$ *inv S* $\Longrightarrow$ *I*$\models$*sextm clauses*$_{NOT}$ *S* $\longleftrightarrow$ *I*$\models$*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 literal multiset set* **and** *S T* :: *'st*
assumes
full: *full dpll-bj S T* **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-marked-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
$\vee$ (*trail T* $\models$*asm clauses*$_{NOT}$ *S* $\wedge$ *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*) $\subseteq$ *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*) $\subseteq$ *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-marked-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*))
$\vee$ (*trail T* $\models$*asm clauses*$_{NOT}$ *T* $\wedge$ *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 literal multiset 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*: $dpll\text{-}bj^{++}$ 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

16.4 CDCL

16.4.1 Learn and Forget

locale *learn-ops* =
dpll-state mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for

mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* +
fixes
learn-cond :: *'cls* ⇒ *'st* ⇒ *bool*
begin
inductive *learn* :: *'st* ⇒ *'st* ⇒ *bool* **where**
$learn_{NOT}$*-rule*: $clauses_{NOT}$ *S* ⊨*pm mset-cls C* ⟹
atms-of (*mset-cls C*) ⊆ *atms-of-mm* ($clauses_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*)) ⟹
learn-cond C S ⟹
T ∼ $add\text{-}cls_{NOT}$ *C S* ⟹
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

locale *forget-ops* =
dpll-state mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* +
fixes
forget-cond :: *'cls* ⇒ *'st* ⇒ *bool*
begin
inductive $forget_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **where**
$forget_{NOT}$:

removeAll-mset (*mset-cls* C)(*clauses*$_{NOT}$ S) $\models$*pm* *mset-cls* $C \Longrightarrow$
forget-cond C $S \Longrightarrow$
C !$\in$! *raw-clauses* $S \Longrightarrow$
$T \sim$ *remove-cls*$_{NOT}$ C $S \Longrightarrow$
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 mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ *learn-cond* +
forget-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ *forget-cond*
for
mset-cls:: $'$*cls* $\Rightarrow$ $'$*v clause* **and**
insert-cls :: $'$*v literal* $\Rightarrow$ $'$*cls* $\Rightarrow$ $'$*cls* **and**
remove-lit :: $'$*v literal* $\Rightarrow$ $'$*cls* $\Rightarrow$ $'$*cls* **and**
mset-clss:: $'$*clss* $\Rightarrow$ $'$*v clauses* **and**
union-clss :: $'$*clss* $\Rightarrow$ $'$*clss* $\Rightarrow$ $'$*clss* **and**
in-clss :: $'$*cls* $\Rightarrow$ $'$*clss* $\Rightarrow$ *bool* **and**
insert-clss :: $'$*cls* $\Rightarrow$ $'$*clss* $\Rightarrow$ $'$*clss* **and**
remove-from-clss :: $'$*cls* $\Rightarrow$ $'$*clss* $\Rightarrow$ $'$*clss* **and**
trail :: $'$*st* $\Rightarrow$ ($'$*v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: $'$*st* $\Rightarrow$ $'$*clss* **and**
prepend-trail :: ($'$*v*, *unit*, *unit*) *marked-lit* $\Rightarrow$ $'$*st* $\Rightarrow$ $'$*st* **and**
tl-trail :: $'$*st* $\Rightarrow$$'$*st* **and**
add-cls$_{NOT}$ :: $'$*cls* $\Rightarrow$ $'$*st* $\Rightarrow$ $'$*st* **and**
remove-cls$_{NOT}$ :: $'$*cls* $\Rightarrow$ $'$*st* $\Rightarrow$ $'$*st* **and**
learn-cond forget-cond :: $'$*cls* $\Rightarrow$ $'$*st* $\Rightarrow$ *bool*
begin
inductive *learn-and-forget*$_{NOT}$ :: $'$*st* $\Rightarrow$ $'$*st* $\Rightarrow$ *bool*
where
lf-learn: *learn* S T $\Longrightarrow$ *learn-and-forget*$_{NOT}$ S T |
lf-forget: *forget*$_{NOT}$ S T $\Longrightarrow$ *learn-and-forget*$_{NOT}$ S T
end

16.4.2 Definition of CDCL

locale *conflict-driven-clause-learning-ops* =
dpll-with-backjumping-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds +
learn-and-forget$_{NOT}$ *mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ *learn-cond*
forget-cond
for
mset-cls:: $'$*cls* $\Rightarrow$ $'$*v clause* **and**

insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *bool* **and**
learn-cond forget-cond :: *'cls* ⇒ *'st* ⇒ *bool*
begin

inductive $cdcl_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
c-dpll-bj: *dpll-bj S S'* ⟹ $cdcl_{NOT}$ *S S'* |
c-learn: *learn S S'* ⟹ $cdcl_{NOT}$ *S S'* |
$c\text{-}forget_{NOT}$: $forget_{NOT}$ *S S'* ⟹ $cdcl_{NOT}$ *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: ⋀*T*. *dpll-bj S T* ⟹ *P S T* **and**
learning:
⋀*C T*. $clauses_{NOT}$ *S* ⊨*pm mset-cls C* ⟹
atms-of (*mset-cls C*) ⊆ *atms-of-mm* ($clauses_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*)) ⟹
T ∼ $add\text{-}cls_{NOT}$ *C S* ⟹
P S T **and**
forgetting: ⋀*C T*. *removeAll-mset* (*mset-cls C*) ($clauses_{NOT}$ *S*) ⊨*pm mset-cls C* ⟹
C !∈! *raw-clauses S* ⟹
T ∼ $remove\text{-}cls_{NOT}$ *C S* ⟹
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 possible that some variable of the trail are not 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*) ⊆ *atms-of-mm* ($clauses_{NOT}$ *S*) ∪ *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*)) ⊆ *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*: $cdcl_{NOT}$*-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-marked-decomposition* (*trail S*))
shows
all-decomposition-implies-m ($clauses_{NOT}$ *T*) (*get-all-marked-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-marked-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*:*mset-cls C* ∈ *set-mset* ($clauses_{NOT}$ *S*)
using *C* **by** *blast*
have $clauses_{NOT}$ *T* = $clauses_{NOT}$ (*remove-cls*$_{NOT}$ *C S*)

using *T state-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-remove-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 ⊨*sextm clauses*$_{NOT}$ *S* **and**
I ⊆ *J* **and**
tot: *total-over-m J* (*set-mset* ({#*mset-cls C*#} + *clauses*$_{NOT}$ *S*)) **and**
cons: *consistent-interp J*
then have *J* ⊨*sm clauses*$_{NOT}$ *S* **unfolding** *true-clss-ext-def* **by** *auto*

moreover
with ‹*clauses*$_{NOT}$ *S*⊨*pm mset-cls C*› **have** *J* ⊨ *mset-cls C*
using *tot cons* **unfolding** *true-clss-cls-def* **by** *auto*
ultimately have *J* ⊨*sm* {#*mset-cls C*#} + *clauses*$_{NOT}$ *S* **by** *auto*
}
then have *H*: *I* ⊨*sextm* (*clauses*$_{NOT}$ *S*) ⟹ *I* ⊨*sext insert* (*mset-cls 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 ⊨*sext set-mset* (*clauses*$_{NOT}$ *S*) − {*mset-cls C*} **and**
I ⊆ *J* **and**
tot: *total-over-m J* (*set-mset* (*clauses*$_{NOT}$ *S*)) **and**
cons: *consistent-interp J*
then have *J* ⊨*s set-mset* (*clauses*$_{NOT}$ *S*) − {*mset-cls C*}
unfolding *true-clss-ext-def* **by** (*meson Diff-subset total-over-m-subset*)

moreover
with *cls-C* **have** *J* ⊨ *mset-cls C*
using *tot cons* **unfolding** *true-clss-cls-def*
by (*metis Un-commute forget*$_{NOT}$*.hyps*(*2*) *in-clss-mset-clss 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*) $-$ {*mset-cls 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*

16.5 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-marked-decomposition* (*trail S*))
  shows
    all-decomposition-implies-m (*clauses$_{NOT}$ T*) (*get-all-marked-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 $\exists j. \forall i \geq j.$ *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*) $\exists j. \forall i \geq j.$ *learn-or-forget* (*f i*) (*f* (*Suc i*))
| (*dpll-more*) $\neg(\exists j. \forall i \geq j.$ *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*: $\exists i.$ ¬ *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**
$\forall k<i.$ *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 $\{i.\ i \leq i_0 \land$ ¬ *learn* (*f i*) (*f* (*Suc i*)) ∧ ¬$forget_{NOT}$ (*f i*) (*f* (*Suc i*))} ≠ {}
by *auto*
let *?I* = $\{i.\ i \leq i_0 \land$ ¬ *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 $\forall k<?i.$ *learn-or-forget* (*f k*) (*f* (*Suc k*))
using *Min.coboundedI*[*of* $\{i.\ i \leq i_0 \land$ ¬ *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 $g \equiv \lambda n.$ *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: ⋀i. i≥j ⟹ learn-or-forget (g i) (g (Suc i))
        using IH unfolding μ[symmetric] by presburger
      show ?thesis
        proof
          {
            fix k
            assume k ≥ 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 i₀ where ¬ learn (f i₀) (f (Suc i₀)) ∧ ¬forget_NOT (f i₀) (f (Suc i₀))
            using j by auto
          then have {i. i≤i₀ ∧ ¬ learn (f i) (f (Suc i)) ∧ ¬forget_NOT (f i) (f (Suc i))} ≠ {}
            by auto
          let ?I = {i. i≤i₀ ∧ ¬ 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* ≤ *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: ⋀*f j*. ¬ (∀ *i*≥*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: $\bigwedge f\ j.\ \neg\ (\forall\ i \geq j.$ *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-marked-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-marked-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-marked-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*

16.6 Termination

16.6.1 Restricting learn and forget

locale *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =
dpll-state mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
conflict-driven-clause-learning mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds
$\lambda C\ S.$ *distinct-mset* (*mset-cls C*) $\wedge$ $\neg$*tautology* (*mset-cls C*) $\wedge$ *learn-restrictions C S* $\wedge$
($\exists F\ K\ d\ F'\ C'\ L.$ *trail S* = F' @ *Marked K* () # *F* $\wedge$ *mset-cls C* = C' + {#*L*#} $\wedge$ *F* $\models$*as CNot* C'
$\wedge$ C' + {#*L*#} $\notin$# $clauses_{NOT}$ *S*)
$\lambda C\ S.$ $\neg$($\exists F'\ F\ K\ d\ L.$ *trail S* = F' @ *Marked K* () # *F* $\wedge$ *F* $\models$*as CNot* (*remove1-mset L* (*mset-cls C*)))
$\wedge$ *forget-restrictions C S*
for
mset-cls:: '*cls* $\Rightarrow$ '*v clause* **and**
insert-cls :: '*v literal* $\Rightarrow$ '*cls* $\Rightarrow$ '*cls* **and**
remove-lit :: '*v literal* $\Rightarrow$ '*cls* $\Rightarrow$ '*cls* **and**
mset-clss:: '*clss* $\Rightarrow$ '*v clauses* **and**
union-clss :: '*clss* $\Rightarrow$ '*clss* $\Rightarrow$ '*clss* **and**
in-clss :: '*cls* $\Rightarrow$ '*clss* $\Rightarrow$ *bool* **and**
insert-clss :: '*cls* $\Rightarrow$ '*clss* $\Rightarrow$ '*clss* **and**
remove-from-clss :: '*cls* $\Rightarrow$ '*clss* $\Rightarrow$ '*clss* **and**
trail :: '*st* $\Rightarrow$ ('*v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: '*st* $\Rightarrow$ '*clss* **and**
prepend-trail :: ('*v*, *unit*, *unit*) *marked-lit* $\Rightarrow$ '*st* $\Rightarrow$ '*st* **and**
tl-trail :: '*st* $\Rightarrow$'*st* **and**
$add\text{-}cls_{NOT}$ :: '*cls* $\Rightarrow$ '*st* $\Rightarrow$ '*st* **and**
$remove\text{-}cls_{NOT}$ :: '*cls* $\Rightarrow$ '*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*, *unit*) *marked-lit* $\Rightarrow$ '*st* $\Rightarrow$ *bool* **and**
learn-restrictions forget-restrictions :: '*cls* $\Rightarrow$ '*st* $\Rightarrow$ *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: $\bigwedge T.$ *dpll-bj S T* $\Longrightarrow$ *P S T* **and**
learning:
$\bigwedge C\ F\ K\ F'\ C'\ L\ T.$ $clauses_{NOT}$ *S* $\models$*pm mset-cls C* $\Longrightarrow$
atms-of (*mset-cls C*) $\subseteq$ *atms-of-mm* ($clauses_{NOT}$ *S*) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail S*)) $\Longrightarrow$
distinct-mset (*mset-cls C*) $\Longrightarrow$
$\neg$ *tautology* (*mset-cls C*) $\Longrightarrow$
learn-restrictions C S $\Longrightarrow$
trail S = F' @ *Marked K* () # *F* $\Longrightarrow$
mset-cls C = C' + {#*L*#} $\Longrightarrow$
F $\models$*as CNot* C' $\Longrightarrow$
C' + {#*L*#} $\notin$# $clauses_{NOT}$ *S* $\Longrightarrow$
T $\sim$ $add\text{-}cls_{NOT}$ *C S* $\Longrightarrow$
P S T **and**

forgetting: $\bigwedge C\ T$. *removeAll-mset* (*mset-cls* C) (*clauses*$_{NOT}$ S) $\models$*pm mset-cls* $C \Longrightarrow$
C !∈! *raw-clauses* $S \Longrightarrow$
$\neg(\exists F'\ F\ K\ L.$ *trail* $S = F'$ @ *Marked* K () # $F \wedge F \models$*as CNot* (*mset-cls* $C - \{\#L\#\}$)) $\Longrightarrow$
$T \sim$ *remove-cls*$_{NOT}$ $C\ S \Longrightarrow$
forget-restrictions $C\ S \Longrightarrow$
$P\ S\ T$
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$_{NOT}$** $S\ T \Longrightarrow$ *inv* $S \Longrightarrow$ *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)
$\subseteq$ *simple-clss* (*atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ‘ *lits-of-l* (*trail* S))
proof
fix C **assume** C: $C \in$ *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 $C \in$ *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* $S \equiv$
$\{C+\{\#L\#\}$ |$C\ L.\ C+\{\#L\#\}$ ∈# *clauses*$_{NOT}$ S $\wedge$ *distinct-mset* ($C+\{\#L\#\}$)
$\wedge$ ¬*tautology* ($C+\{\#L\#\}$)
$\wedge$ ($\exists F'\ K\ F.$ *trail* $S = F'$ @ *Marked* 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 $-$ {*mset-cls* 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 $-$ {*mset-cls* 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. *mset-cls* C' = C +{#L#} $\wedge$ *distinct-mset* (C+{#L#}) $\wedge$ $\neg$*tautology* (C+{#L#})
  $\wedge$ ($\exists$ F' K d F. *trail* S = F' @ *Marked* K () # F $\wedge$ F $\models$*as* *CNot* C)
  then {*mset-cls* 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' @ *Marked* 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#} $\in$# {#*mset-cls* 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#} $\in$# {#*mset-cls* C'#} $\wedge$ P C L T} =
   (*if* $\exists$ C L. *mset-cls* C' = C +{#L#} $\wedge$ P C L S *then* {*mset-cls* 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. *mset-cls* C' = C +{#L#}$\wedge$ *distinct-mset* (C+{#L#}) $\wedge$ $\neg$*tautology* (C+{#L#})
   $\wedge$ ($\exists$ F' K d F. *trail* S = F' @ *Marked* K () # F $\wedge$ F $\models$*as* *CNot* C)
   then {*mset-cls* 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 *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 *auto*
  unfolding *conflicting-bj-clss-def*
  apply *clarify*
  using *diff-union-cancelR* **by** (*metis diff-union-cancelR*)

lemma *forget-*μ_L*-decrease*:
  assumes $forget_{NOT}$: $forget_{NOT}$ *S T*
  shows (μ_L *b T*, μ_L *b S*) ∈ *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* ∨ *Q a*) ∧ *S a*} = (*if S b then* {*b*} *else* {}) ∪ {*a*. *Q a* ∧ *S a*}
  by *auto*

lemma *set-insert-neq*:
  A ≠ *insert a A* ⟷ *a* ∉ *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′* @ *Marked K* () # *F* **and**

C: *mset-cls* $C = C' + \{\#L\#\}$ **and**
F: $F \models as$ *CNot* C' **and**
C-S:$C' + \{\#L\#\} \notin\#$ $clauses_{NOT}$ S
by *blast*
moreover have *distinct-mset* (*mset-cls* C) $\neg$ *tautology* (*mset-cls* C) **using** *inv* **by** *blast+*
ultimately have $C' + \{\#L\#\} \in$ *conflicting-bj-clss* T
using T *n-d* **unfolding** *conflicting-bj-clss-def* **by** *fastforce*
moreover have $C' + \{\#L\#\} \notin$ *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*: $conflicting\text{-}bj\text{-}clss\text{-}add\text{-}cls_{NOT}$)
ultimately have *card* (*conflicting-bj-clss* T) $\geq$ *card* (*conflicting-bj-clss* S)
using *card-mono* **by** *blast*

moreover
have *fin'*: *finite* (*atms-of-mm* ($clauses_{NOT}$ T) $\cup$ *atm-of* ' *lits-of-l* (*trail* T))
by *auto*
have *1*:*atms-of-ms* (*conflicting-bj-clss* T) $\subseteq$ *atms-of-mm* ($clauses_{NOT}$ T)
unfolding *conflicting-bj-clss-def atms-of-ms-def* **by** *auto*
have *2*: $\bigwedge x.$ $x\in$ *conflicting-bj-clss* $T \Longrightarrow \neg$ *tautology* $x \wedge$ *distinct-mset* x
unfolding *conflicting-bj-clss-def* **by** *auto*
have T: *conflicting-bj-clss* T
$\subseteq$ *simple-clss* (*atms-of-mm* ($clauses_{NOT}$ T) $\cup$ *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) $\cup$ *atm-of* ' *lits-of-l* (*trail* T))
$\geq$ *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) $\cup$ *atm-of* ' *lits-of-l* (*trail* T) $\subseteq A$
using $learn_{NOT}E$[*OF learnST*] A **by** *simp*
then have *3* ^ (*card* A) $\geq$ *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* $S \neq$ *conflicting-bj-clss* T⟩
⟨*conflicting-bj-clss* $S \subseteq$ *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) $\subseteq$ *atms-of-ms* A and in the clauses *atms-of-mm* ($clauses_{NOT}$ S) $\subseteq$ *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*) ⊆ *atms-of-ms A* ∧ *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*)
∧ *inv S*)
∧ *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 literal multiset 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 $\mu_{CDCL}{}'$ :: *'v literal multiset set* ⇒ *'st* ⇒ *nat* **where**
$\mu_{CDCL}{}'$ *A T* ≡
((*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 μ_{CDCL}' A T < μ_{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)) − μ_C' A T
< (*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A)) − μ_C' A S
using *dpll-bj-trail-mes-decreasing-prop fin-A inv n-d atms-clss atms-trail*
unfolding μ_C'*-def* **by** *blast*
then have *XX*: ((*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A)) − μ_C' A T) + *1*
≤ (*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A)) − μ_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)) − μ_C' A T) ∗
(*1* + *3* ^ *card* (*atms-of-ms* A)) + (*1* + *3* ^ *card* (*atms-of-ms* A))
≤ ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_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)) − μ_C' A T)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) + *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) T
< ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A S) ∗(*1* + *3* ^ *card* (*atms-of-ms* A))
by *linarith*
then have ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A T)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A))
+ *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) T
< ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A S)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A))
+ *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) S
by *linarith*
then have ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A T)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) ∗ *2*
+ *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) T ∗ *2*
< ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A S)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) ∗ *2*
+ *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) S ∗ *2*
by *linarith*
then show *?case* **unfolding** μ_{CDCL}'*-def cl-T-S* **by** *presburger*
next
case (*learn* C F' K F C' L T) **note** *clss-S-C* = *this*(*1*) **and** *atms-C* = *this*(*2*) **and** *dist* = *this*(*3*)
and *tauto* = *this*(*4*) **and** *learn-restr* = *this*(*5*) **and** *tr-S* = *this*(*6*) **and** C' = *this*(*7*) **and**
F-C = *this*(*8*) **and** *C-new* = *this*(*9*) **and** T = *this*(*10*)
have *insert* (*mset-cls* C) (*conflicting-bj-clss* S) $\subseteq$ *simple-clss* (*atms-of-ms* A)

proof −
have *mset-cls* $C \in$ *simple-clss* (*atms-of-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-bj-clss* $S \subseteq$ *simple-clss* (*atms-of-ms* A)
proof
fix x :: *'v literal multiset*
assume $x \in$ *conflicting-bj-clss* S
then have $x \in\#$ $clauses_{NOT}$ $S \wedge$ *distinct-mset* $x \wedge \neg$ *tautology* x
unfolding *conflicting-bj-clss-def* **by** *blast*
then show $x \in$ *simple-clss* (*atms-of-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* (*mset-cls* C) (*conflicting-bj-clss* S)) ≤ 3 ^ (*card* (*atms-of-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* (*mset-cls* C) (*conflicting-bj-clss* S))
= *Suc* (*card* ((*conflicting-bj-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-bj-clss* ($add\text{-}cls_{NOT}$ C S) = *conflicting-bj-clss* $S \cup \{$*mset-cls* $C\}$
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-bj-clss-yet* (*card* (*atms-of-ms* A)) S
= *Suc* (*conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) ($add\text{-}cls_{NOT}$ C S))
by *simp*
have *1*: $clauses_{NOT}$ T = $clauses_{NOT}$ ($add\text{-}cls_{NOT}$ 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)) ($add\text{-}cls_{NOT}$ C S)
using T **unfolding** *conflicting-bj-clss-def* **by** *auto*
have *3*: μ_C' A T = μ_C' A ($add\text{-}cls_{NOT}$ C S)
using T **unfolding** μ_C'*-def* **by** *auto*
have ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A ($add\text{-}cls_{NOT}$ C S))
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) ∗ *2*
= ((*2* + *card* (*atms-of-ms* A)) ^ (*1* + *card* (*atms-of-ms* A)) − μ_C' A S)
∗ (*1* + *3* ^ *card* (*atms-of-ms* A)) ∗ *2*
using *n-d* **unfolding** μ_C'*-def* **by** *auto*
moreover
have *conflicting-bj-clss-yet* (*card* (*atms-of-ms* A)) ($add\text{-}cls_{NOT}$ C S)
∗ *2*
+ *card* (*set-mset* ($clauses_{NOT}$ ($add\text{-}cls_{NOT}$ 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** μ_{CDCL}'*-def 1 2 3* **by** *presburger*
next
case ($forget_{NOT}$ C T) **note** T = *this*(*4*)
have [*simp*]: μ_C' A ($remove\text{-}cls_{NOT}$ C S) = μ_C' A S
unfolding μ_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*)
    in-clss-mset-clss order-refl set-mset-minus-replicate-mset(*1*) *state-eq*$_{NOT}$*-clauses*)
ultimately show *?case* **unfolding** μ_{CDCL}*'-def*
  using T ‹μ_C*'* A (*remove-cls*$_{NOT}$ C S) = μ_C*'* A S› **by** (*metis* (*no-types*) *add-le-cancel-left*
    μ_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* (*mset-cls* C) ⊆ A
    using *atms-C atms-clss-S atms-trail-S* **by** *fast*
  then have *simple-clss* (*atms-of* (*mset-cls* C)) ⊆ *simple-clss* A
    by (*simp add*: *simple-clss-mono*)
  then have *mset-cls* C ∈ *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) ⊆ A **and**
    atm-of ‘(*lits-of-l* (*trail* S)) ⊆ A **and**
    n-d: *no-dup* (*trail* S) **and**
    finite: *finite* A
  shows *set-mset* ($clauses_{NOT}$ T) ⊆ *set-mset* ($clauses_{NOT}$ S) ∪ *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 $rtranclp\text{-}cdcl_{NOT}\text{-}inv$ *st inv* **by** *blast*
moreover have *atms-of-mm* ($clauses_{NOT}$ *T*) ⊆ *A* **and** *atm-of* ' *lits-of-l* (*trail T*) ⊆ *A*
using $rtranclp\text{-}cdcl_{NOT}\text{-}trail\text{-}clauses\text{-}bound$[*OF st*] *inv atms-clss-S atms-trail-S n-d* **by** *auto*
moreover have *no-dup* (*trail T*)
using $rtranclp\text{-}cdcl_{NOT}\text{-}no\text{-}dup$[*OF st* ‹*inv S*› *n-d*] **by** *simp*
ultimately have *set-mset* ($clauses_{NOT}$ *U*) ⊆ *set-mset* ($clauses_{NOT}$ *T*) ∪ *simple-clss A*
using $cdcl_{NOT}$ *finite n-d* **by** (*auto simp*: $cdcl_{NOT}\text{-}clauses\text{-}bound$)
then show *?case* **using** *IH* **by** *auto*
qed

lemma $rtranclp\text{-}cdcl_{NOT}\text{-}card\text{-}clauses\text{-}bound$:
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**
finite: *finite A*
shows *card* (*set-mset* ($clauses_{NOT}$ *T*)) ≤ *card* (*set-mset* ($clauses_{NOT}$ *S*)) + *3* ^ (*card A*)
using $rtranclp\text{-}cdcl_{NOT}\text{-}clauses\text{-}bound$[*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 $rtranclp\text{-}cdcl_{NOT}\text{-}card\text{-}clauses\text{-}bound'$:
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**
finite: *finite A*
shows *card* {*C*|*C*. *C* ∈# $clauses_{NOT}$ *T* ∧ (*tautology C* ∨ ¬*distinct-mset C*)}
≤ *card* {*C*|*C*. *C*∈# $clauses_{NOT}$ *S* ∧ (*tautology C* ∨ ¬*distinct-mset C*)} + *3* ^ (*card A*)
(**is** *card ?T* ≤ *card ?S* + -)
using $rtranclp\text{-}cdcl_{NOT}\text{-}clauses\text{-}bound$[*OF assms*] *finite*
proof −
have *?T* ⊆ *?S* ∪ *simple-clss A*
using $rtranclp\text{-}cdcl_{NOT}\text{-}clauses\text{-}bound$[*OF assms*] **by** *force*
then have *card ?T* ≤ *card* (*?S* ∪ *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 $rtranclp\text{-}cdcl_{NOT}\text{-}card\text{-}simple\text{-}clauses\text{-}bound$:
assumes
${cdcl_{NOT}}^{**}$ *S T* **and**
inv S **and**
NA: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *A* **and**
MA: *atm-of* ' (*lits-of-l* (*trail S*)) ⊆ *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 literal multiset 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 $\mu_{CDCL}{}'$ *A U* $\leq$ $\mu_{CDCL}{}'$*-bound A S*
proof −
have $((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ U)$
$\leq (2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A))$
by *auto*
then have $((2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ U)$
$* (1 + 3\ \hat{}\ card\ (atms\text{-}of\text{-}ms\ A)) * 2$
$\leq (2 + card\ (atms\text{-}of\text{-}ms\ A))\ \hat{}\ (1 + card\ (atms\text{-}of\text{-}ms\ A)) * (1 + 3\ \hat{}\ card\ (atms\text{-}of\text{-}ms\ A)) * 2$
using *mult-le-mono1* **by** *blast*
moreover
have *conflicting-bj-clss-yet* (*card* (*atms-of-ms A*)) $T * 2 \leq 2 * 3\ \hat{}\ card\ (atms\text{-}of\text{-}ms\ A)$
by *linarith*
moreover have *card* (*set-mset* (*clauses*$_{NOT}$ *U*))
$\leq card\ \{C.\ C \in\#\ clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3\ \hat{}\ card\ (atms\text{-}of\text{-}ms\ A)$
using *rtranclp-cdcl*$_{NOT}$*-card-simple-clauses-bound*[*OF assms*(*1*−*6*)] *U* **by** *auto*
ultimately show *?thesis*
unfolding $\mu_{CDCL}{}'$*-def* $\mu_{CDCL}{}'$*-bound-def* **by** *linarith*
qed

lemma *rtranclp-cdcl*$_{NOT}$*-*$\mu_{CDCL}{}'$*-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* ≤ 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* ≤ 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* ∈# *clauses*$_{NOT}$ *T* ∧ (*tautology C* ∨ ¬ *distinct-mset C*)} ≤
    card {*C*. *C* ∈# *clauses*$_{NOT}$ *S* ∧ (*tautology C* ∨ ¬ *distinct-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*

16.7 CDCL with restarts

16.7.1 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 $\Longrightarrow$ *cdcl*$_{NOT}$*-raw-restart* S T

end

locale *conflict-driven-clause-learning-with-restarts* =
conflict-driven-clause-learning mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds learn-cond forget-cond
for
mset-cls:: $'cls \Rightarrow 'v$ *clause* **and**
insert-cls :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
remove-lit :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
mset-clss:: $'clss \Rightarrow 'v$ *clauses* **and**
union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
in-clss :: $'cls \Rightarrow 'clss \Rightarrow$ *bool* **and**
insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
remove-from-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
trail :: $'st \Rightarrow ('v, unit, unit)$ *marked-lits* **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v, unit, unit)$ *marked-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-cls$_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls$_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **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 :: $('v, unit, unit)$ *marked-lit* $\Rightarrow 'st \Rightarrow$ *bool* **and**
learn-cond forget-cond :: $'cls \Rightarrow 'st \Rightarrow$ *bool*
begin

lemma *cdcl*$_{NOT}$*-iff-cdcl*$_{NOT}$*-raw-restart-no-restarts*:
cdcl$_{NOT}$ S T $\longleftrightarrow$ *restart-ops.cdcl*$_{NOT}$*-raw-restart cdcl*$_{NOT}$ (λ- -. *False*) S T
(**is** *?C* S T $\longleftrightarrow$ *?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

16.7.2 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* ⇒ *'st* ⇒ *bool* **and**
    $cdcl_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* +
  fixes
    f :: *nat* ⇒ *nat* **and**
    bound-inv :: *'bound* ⇒ *'st* ⇒ *bool* **and**
    μ :: *'bound* ⇒ *'st* ⇒ *nat* **and**
    $cdcl_{NOT}$*-inv* :: *'st* ⇒ *bool* **and**
    μ*-bound* :: *'bound* ⇒ *'st* ⇒ *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
    $(cdcl_{NOT}$^^$n)$ $S\ T$ **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
    $(cdcl_{NOT}$^^$n)$ $S\ T$ **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
($cdcl_{NOT}$^^(*Suc n*)) *S T* **and**
bound-inv A S
$cdcl_{NOT}$-*inv S*
shows μ *A T* $<$ μ *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*: ($cdcl_{NOT}$^^(*Suc n*)) *S U* **and** *U-T*: $cdcl_{NOT}$ *U T* **using** *S-T* **by** *auto*
then have μ *A U* $<$ μ *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 μ *A T* $<$ μ *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* $\wedge$ $cdcl_{NOT}$-*inv S* $\wedge$ *bound-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* ≥ *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 mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ remove-cls$_{NOT}$
for
mset-cls:: *'cls* $\Rightarrow$ *'v clause* **and**
insert-cls :: *'v literal* $\Rightarrow$ *'cls* $\Rightarrow$ *'cls* **and**
remove-lit :: *'v literal* $\Rightarrow$ *'cls* $\Rightarrow$ *'cls* **and**
mset-clss:: *'clss* $\Rightarrow$ *'v clauses* **and**
union-clss :: *'clss* $\Rightarrow$ *'clss* $\Rightarrow$ *'clss* **and**
in-clss :: *'cls* $\Rightarrow$ *'clss* $\Rightarrow$ *bool* **and**
insert-clss :: *'cls* $\Rightarrow$ *'clss* $\Rightarrow$ *'clss* **and**
remove-from-clss :: *'cls* $\Rightarrow$ *'clss* $\Rightarrow$ *'clss* **and**
trail :: *'st* $\Rightarrow$ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* $\Rightarrow$ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
tl-trail :: *'st* $\Rightarrow$*'st* **and**
add-cls$_{NOT}$ :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
remove-cls$_{NOT}$ :: *'cls* $\Rightarrow$ *'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: $\bigwedge A\ T\ V\ n.\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow cdcl_{NOT}\text{-}restart\ (T,\ n)\ (V,\ Suc\ n) \Longrightarrow \mu\ A\ V \leq \mu\text{-}bound\ A\ T$ **and**
$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}\text{-}restart\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\text{-}bound\ A\ V \leq \mu\text{-}bound\ A\ T$
begin

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}\text{-}restart^{**}\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\text{-}bound\ A\ V \leq \mu\text{-}bound\ A\ T$
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*:
$cdcl_{NOT}\text{-}restart\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu\text{-}bound\ A\ T$
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*:
$cdcl_{NOT}\text{-}restart^{**}\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu\text{-}bound\ A\ T$
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*:
$wf\ \{(T,\ S).\ cdcl_{NOT}\text{-}restart\ S\ T \land cdcl_{NOT}\text{-}inv\ (fst\ S)\}$ (**is** *wf ?A*)
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain g **where**
g: $\bigwedge i.\ cdcl_{NOT}\text{-}restart\ (g\ i)\ (g\ (Suc\ i))$ **and**
$cdcl_{NOT}$*-inv-g*: $\bigwedge i.\ cdcl_{NOT}\text{-}inv\ (fst\ (g\ i))$
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*: $\bigwedge T\ Ta\ m.$ ($cdcl_{NOT}$ ^^ *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*. ($cdcl_{NOT}$ ^^ *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* = μ*-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* $\Longrightarrow$ $cdcl_{NOT}$*-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 $cdcl_{NOT}$*-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*)) $\leq$ μ*-bound A* (*fst* (*g 1*))
apply (*rule rtranclp-*$cdcl_{NOT}$*-raw-restart-measure-bound*)
using ‹$cdcl_{NOT}$*-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*)))›
‹$cdcl_{NOT}$*-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*: ($cdcl_{NOT}$ ^^ *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*)) $\leq$ μ*-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 (λ*S T*. $cdcl_{NOT}$ *S T* $\wedge$ $cdcl_{NOT}$*-inv S* $\wedge$ *bound-inv A S*)** *S T* $\longleftrightarrow$ $cdcl_{NOT}{}^{**}$ *S T*
(**is** *?A*** *S T* $\longleftrightarrow$ *?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* (λ*S T*. $cdcl_{NOT}$ *S T* $\wedge$ $cdcl_{NOT}$*-inv S* $\wedge$ *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

16.8 Merging backjump and learning

locale $cdcl_{NOT}$*-merge-bj-learn-ops* =
decide-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ +
forget-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ *forget-cond* +
propagate-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$ *propagate-conds*
for
mset-cls:: $'cls \Rightarrow 'v$ *clause* **and**
insert-cls :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
remove-lit :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
mset-clss:: $'clss \Rightarrow 'v$ *clauses* **and**
union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
in-clss :: $'cls \Rightarrow 'clss \Rightarrow$ *bool* **and**
insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
remove-from-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
trail :: $'st \Rightarrow ('v$, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v$, *unit*, *unit*) *marked-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-cls$_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls$_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
propagate-conds :: $('v$, *unit*, *unit*) *marked-lit* $\Rightarrow 'st \Rightarrow$ *bool* **and**
forget-cond :: $'cls \Rightarrow 'st \Rightarrow$ *bool* +
fixes *backjump-l-cond* :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow$ *bool*
begin
inductive *backjump-l* **where**
backjump-l: *trail S* = *F'* @ *Marked K* () # *F*
$\Longrightarrow$ *no-dup* (*trail S*)
$\Longrightarrow$ *T* ∼ *prepend-trail* (*Propagated L* ()) (*reduce-trail-to*$_{NOT}$ *F* (*add-cls*$_{NOT}$ *C''* *S*))
$\Longrightarrow$ *C* ∈# *clauses*$_{NOT}$ *S*
$\Longrightarrow$ *trail S* ⊨*as CNot C*
$\Longrightarrow$ *undefined-lit F L*
$\Longrightarrow$ *atm-of L* ∈ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
$\Longrightarrow$ *clauses*$_{NOT}$ *S* ⊨*pm C'* + {#*L*#}
$\Longrightarrow$ *mset-cls C''* = *C'* + {#*L*#}
$\Longrightarrow$ *F* ⊨*as CNot C'*
$\Longrightarrow$ *backjump-l-cond C C' L S T*
$\Longrightarrow$ *backjump-l S T*

Avoid (meaningless) simplification:

declare *reduce-trail-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-trail-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 mset-cls insert-cls remove-lit*
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-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
    mset-cls:: *'cls* $\Rightarrow$ *'v clause* **and**
    insert-cls :: *'v literal* $\Rightarrow$ *'cls* $\Rightarrow$ *'cls* **and**
    remove-lit :: *'v literal* $\Rightarrow$ *'cls* $\Rightarrow$ *'cls* **and**
    mset-clss:: *'clss* $\Rightarrow$ *'v clauses* **and**
    union-clss :: *'clss* $\Rightarrow$ *'clss* $\Rightarrow$ *'clss* **and**
    in-clss :: *'cls* $\Rightarrow$ *'clss* $\Rightarrow$ *bool* **and**
    insert-clss :: *'cls* $\Rightarrow$ *'clss* $\Rightarrow$ *'clss* **and**
    remove-from-clss :: *'cls* $\Rightarrow$ *'clss* $\Rightarrow$ *'clss* **and**
    trail :: *'st* $\Rightarrow$ (*'v*, *unit*, *unit*) *marked-lits* **and**
    raw-clauses :: *'st* $\Rightarrow$ *'clss* **and**
    prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    tl-trail :: *'st* $\Rightarrow$*'st* **and**
    add-cls$_{NOT}$ :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    remove-cls$_{NOT}$ :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
    propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
    forget-cond :: *'cls* $\Rightarrow$ *'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* $S = F'$ @ *Marked* 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' @ *Marked 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 mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-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-trail-to*$_{NOT}$ *F* (*add-cls*$_{NOT}$ *D S*))
and
tr-S: *trail S* = F' @ *Marked 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'*
mset-cls D = C' + {#*L*#}
by (*elim backjump-lE*) *metis*
interpret *backjumping-ops mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses 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
} **note** $H = this(1)$
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 mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
propagate-conds forget-cond backjump-l-cond inv
for
mset-cls:: $'cls \Rightarrow 'v$ *clause* **and**
insert-cls :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
remove-lit :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
mset-clss:: $'clss \Rightarrow 'v$ *clauses* **and**
union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
in-clss :: $'cls \Rightarrow 'clss \Rightarrow bool$ **and**
insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
remove-from-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
trail :: $'st \Rightarrow ('v, unit, unit)$ *marked-lits* **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v, unit, unit)$ *marked-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
propagate-conds :: $('v, unit, unit)$ *marked-lit* $\Rightarrow 'st \Rightarrow bool$ **and**
forget-cond :: $'cls \Rightarrow 'st \Rightarrow bool$ **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 mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds
λC -. *distinct-mset* (*mset-cls* C) $\wedge$ $\neg$*tautology* (*mset-cls* C)
forget-cond
by *unfold-locales*
end

locale $cdcl_{NOT}$*-merge-bj-learn* =
$cdcl_{NOT}$*-merge-bj-learn-proxy2 mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
propagate-conds forget-cond backjump-l-cond inv
for
mset-cls:: $'cls \Rightarrow 'v$ *clause* **and**
insert-cls :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
remove-lit :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
mset-clss:: $'clss \Rightarrow 'v$ *clauses* **and**
union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
in-clss :: $'cls \Rightarrow 'clss \Rightarrow bool$ **and**

insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *bool* **and**
forget-cond :: *'cls* ⇒ *'st* ⇒ *bool* **and**
inv :: *'st* ⇒ *bool* +
assumes
dpll-merge-bj-inv: ⋀*S T*. *dpll-bj S T* ⟹ *inv S* ⟹ *inv T* **and**
learn-inv: ⋀*S T*. *learn S T* ⟹ *inv S* ⟹ *inv T*
begin

sublocale
conflict-driven-clause-learning mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds
λ*C* -. *distinct-mset* (*mset-cls C*) ∧ ¬*tautology* (*mset-cls 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* ($add\text{-}cls_{NOT}$ *D S*)
∧ *mset-cls D* = (*C'* + {#*L*#})
∧ *backjump* ($add\text{-}cls_{NOT}$ *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'* @ *Marked K* () # *F* **and**
T: *T* ∼ *prepend-trail* (*Propagated L l*) ($reduce\text{-}trail\text{-}to_{NOT}$ *F* ($add\text{-}cls_{NOT}$ *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 mset-cls D* **and**
D: *mset-cls D* = *C'* + {#*L*#}
F ⊨*as CNot C'* **and**
distinct: *distinct-mset* (*mset-cls D*) **and**
not-tauto: ¬ *tautology* (*mset-cls 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* ($add\text{-}cls_{NOT}$ *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* $\models$*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* $\Longrightarrow$ *inv S* $\Longrightarrow$ *no-dup* (*trail S*) $\Longrightarrow$ *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 literal multiset* **and** *L* :: *'v literal* **and** *D* :: *'cls* **where**
f3: *learn S* (*add-cls*$_{NOT}$ *D S*) $\wedge$
backjump (*add-cls*$_{NOT}$ *D S*) *T* $\wedge$
atms-of (*C'* + {#*L*#}) $\subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\cup$ *atm-of* ‘ *lits-of-l* (*trail S*) **and**
D: *mset-cls 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* $\Longrightarrow$ *inv S* $\Longrightarrow$ *no-dup* (*trail S*) $\Longrightarrow$ *cdcl*$_{NOT}$** *S T* $\wedge$ *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 literal multiset 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 literal multiset 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*))

$- \mu_C$ (*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*))
$- \mu_C$ (*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 in-clss-mset-clss 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*))
$- \mu_C$ (*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*))
$- \mu_C$ (*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: *mset-cls D* = *C'* + {#*L*#}
using *bj-l inv backjump-l-learn-backjump n-d atm-clss atm-trail* **by** *meson*
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*))
$- \mu_C$ (*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*))
$- \mu_C$ (*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*))
$- \mu_C$ (*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*))
$- \mu_C$ (*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-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*$))$
    $\wedge$ *cdcl$_{NOT}$-merged-bj-learn S T*$\}$
  apply $($*rule wfP-if-measure*$[$*of - -* μ_{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-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*$))$
    $\wedge$ *cdcl$_{NOT}$-merged-bj-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-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*$))$
    $\wedge$ *cdcl$_{NOT}$-merged-bj-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* $\Longrightarrow$ ¬*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 literal multiset 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*) ⊆ *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-marked-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*))
  ∨ (*trail S* ⊨*asm clauses*$_{NOT}$ *S* ∧ *satisfiable* (*set-mset* (*clauses*$_{NOT}$ *S*)))
proof −
let *?N* = *set-mset* (*clauses*$_{NOT}$ *S*)
let *?M* = *trail S*
consider
    (*sat*) *satisfiable ?N* **and** *?M* ⊨*as ?N*
  | (*sat'*) *satisfiable ?N* **and** ¬ *?M* ⊨*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* ∈ *?N* **and** ¬*?M* ⊨*a C* **using** *M* **unfolding** *true-annots-def* **by** *auto*
    obtain *I* :: *'v literal set* **where**
      I ⊨*s ?N* **and**
      cons: *consistent-interp I* **and**
      tot: *total-over-m I ?N* **and**
      atm-I-N: *atm-of ‘I* ⊆ *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-marked 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 atms-of-s-def* **unfolding** *total-over-m-def total-over-set-def*
      by (*fastforce simp*: *image-iff*)
    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*: *image-iff lits-of-def dest*!: *is-marked-ex-Marked*)

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 Marked-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* (*Marked* (*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* ⊨*as CNot C*
apply (*rule all-variables-defined-not-imply-cnot*)
  using *atms-N-M* ‹*C* ∈ *?N*› ‹¬ *?M* ⊨*a C*› *atms-of-atms-of-ms-mono*[*OF* ‹*C* ∈ *?N*›]
  by (*auto dest*: *atms-of-atms-of-ms-mono*)
have ∃ *l* ∈ *set ?M*. *is-marked l*
  proof (*rule ccontr*)
    let *?O* = {*unmark L* |*L*. *is-marked 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-marked L* ∧ *L* ∈ *set ?M*}
      = {*unmark L* |*L*. *is-marked 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** *d* :: *unit* **and**
  F F' :: (*'v*, *unit*, *unit*) *marked-lit list* **where**
  M-K: *?M* = *F'* @ *Marked K* () # *F* **and**
  nm: ∀*f*∈*set F'*. ¬*is-marked f*
  unfolding *is-marked-def* **by** (*metis* (*full-types*) *old.unit.exhaust*)
let *?K* = *Marked K* ()::(*'v*, *unit*, *unit*) *marked-lit*
have *?K* ∈ *set ?M*

unfolding *M-K* **by** *auto*
let *?C = image-mset lit-of* {#L∈#mset *?M. is-marked L* ∧ L≠?K#} :: 'v *literal multiset*
let *?C' = set-mset* (*image-mset* (λL::'v *literal*. {#L#}) (*?C + unmark ?K*))
have *?N* ∪ {*unmark L* |*L. is-marked 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-marked L* ∧ *L* ∈ *set ?M*}
unfolding *M-K* **apply** *standard*
apply *force*
using *IntI* **by** *auto*
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-marked a* ∧ *a* ≠ *Marked K* ()} =
set (*trail S*) ∩ {*L. is-marked L* ∧ *L* ≠ *Marked K* ()}
by *auto*
then have *tot'*: *total-over-set I*
(*atm-of* ‘ *lit-of* ‘ (*set ?M* ∩ {*L. is-marked L* ∧ *L* ≠ *Marked K* ()}))
using *tot* **by** (*auto simp add*: *atms-of-uminus-lit-atm-of-lit-of*)
{ **fix** *x* :: ('*v*, *unit*, *unit*) *marked-lit*
assume
a3: *lit-of x* ∉ *I* **and**
a1: *x* ∈ *set ?M* **and**
a4: *is-marked x* **and**
a5: *x* ≠ *Marked 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 $\neg I \models s$ *?C'*
using ⟨*?N* ∪ *?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-mset uminus ?C* + {#− *K*#}
unfolding *true-clss-def true-cls-def Bex-def*
using ⟨$(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$⟩
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-marked L* ∧ *L* ≠ *Marked K* ()#})]
⟨*C*∈*?N*⟩ *n-s* ⟨*?M* $\models$*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 literal multiset 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-marked-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ *T*))
∨ (*trail T* $\models$*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-marked-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

16.8.1 Instantiations

locale $cdcl_{NOT}$*-with-backtrack-and-restarts* =
conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt

mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds learn-restrictions forget-restrictions
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *bool* **and**
learn-restrictions forget-restrictions :: *'cls* ⇒ *'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* ∼ $reduce\text{-}trail\text{-}to_{NOT}$ ([]::*'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* $\lambda S\ T.\ T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S* $cdcl_{NOT}$ *f*
 $\lambda A\ S.$ *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
 $\mu_{CDCL}{}'$ $\lambda S.$ *inv S* $\wedge$ *no-dup* (*trail S*)
 $\mu_{CDCL}{}'$*-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-*$cdcl_{NOT}$*-*$\mu_{CDCL}{}'$*-bound*; *simp*)
   apply (*rule rtranclp-*$\mu_{CDCL}{}'$*-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-*$\mu_{CDCL}{}'$*-le-*$\mu_{CDCL}{}'$*-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*) $\subseteq$ *atms-of-ms A*
   atm-of ‘ *lits-of-l* (*trail T*) $\subseteq$ *atms-of-ms A*
   finite A
 shows $\mu_{CDCL}{}'$ *A V* $\leq$ $\mu_{CDCL}{}'$*-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-*$cdcl_{NOT}$*-*$\mu_{CDCL}{}'$*-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-*$cdcl_{NOT}$*-*$\mu_{CDCL}{}'$*-bound*)
  using *full 2* **unfolding** *full1-def* **by** *force+*
qed

lemma $cdcl_{NOT}$*-with-restart-*$\mu_{CDCL}{}'$*-bound-le-*$\mu_{CDCL}{}'$*-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*) $\subseteq$ *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
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}*' cdcl*$_{NOT}$
λ*S*. *inv S* ∧ *no-dup* (*trail S*)
μ_{CDCL}*'-bound*
apply *unfold-locales*
using *cdcl*$_{NOT}$*-with-restart-*μ_{CDCL}*'-le-*μ_{CDCL}*'-bound* **apply** *simp*
using *cdcl*$_{NOT}$*-with-restart-*μ_{CDCL}*'-bound-le-*μ_{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-marked-decomposition* (*trail* (*fst S*)))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst T*)) (*get-all-marked-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-marked-decomposition* (*trail* (*fst S*)))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst T*)) (*get-all-marked-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*:
  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 *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 literal multiset set* **and** *S T* :: *'st*
  assumes
    full: *full cdcl*$_{NOT}$*-restart* (*S*, *n*) (*T*, *m*) **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**
    fin-A[*simp*]: *finite A* **and**
    inv: *inv S* **and**

decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ *S*) (*get-all-marked-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-marked-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*:⋀*I*. *I* ⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I* ⊨*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* ⊨*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*) ⊨*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. But there is a forget rule.

locale *cdcl*$_{NOT}$*-merge-bj-learn-with-backtrack-restarts* =
cdcl$_{NOT}$*-merge-bj-learn mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cls$_{NOT}$ *remove-cls*$_{NOT}$
λC C' L' S T. *distinct-mset* (*C'* + {#*L'*#}) ∧ *backjump-l-cond C C' L' S T*

propagate-conds forget-conds inv
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *marked-lit* ⇒ *'st* ⇒ *bool* **and**
inv :: *'st* ⇒ *bool* **and**
forget-conds :: *'cls* ⇒ *'st* ⇒ *bool* **and**
backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool*
+
fixes *f* :: *nat* ⇒ *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* A = {#C ∈# A. *tautology* C ∨ ¬*distinct-mset* C#}

lemma *simple-clss-or-not-simplified-cls*:
assumes *atms-of-mm* (*clauses*$_{NOT}$ S) ⊆ *atms-of-ms* A **and**
x ∈# *clauses*$_{NOT}$ S **and** *finite* A
shows x ∈ *simple-clss* (*atms-of-ms* A) ∨ x ∈# *not-simplified-cls* (*clauses*$_{NOT}$ S)
proof −
consider
(*simpl*) ¬*tautology* x **and** *distinct-mset* x
| (*n-simp*) *tautology* x ∨ ¬*distinct-mset* x
by *auto*
then show *?thesis*
proof *cases*
case *simpl*
then have x ∈ *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 ∈# *not-simplified-cls* (*clauses*$_{NOT}$ S)
using ‹x ∈# *clauses*$_{NOT}$ S› **unfolding** *not-simplified-cls-def* **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*) ⊆ *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*
shows *set-mset* (*$clauses_{NOT}$ T*) ⊆ *set-mset* (*not-simplified-cls* (*$clauses_{NOT}$ S*))
  ∪ *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 ‹*backjump-l S T*› *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'* @ *Marked 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: *mset-cls 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-def* **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*))
$\subseteq$ *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 μ_{CDCL}*'-bound* **where**
μ_{CDCL}*'-bound A T* $\equiv$ ((*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*) $\subseteq$ *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 A*
shows μ_{CDCL}*'-merged A T* $\leq$ μ_{CDCL}*'-bound A S*
proof −
have *set-mset* ($clauses_{NOT}$ *T*) $\subseteq$ *set-mset* (*not-simplified-cls*($clauses_{NOT}$ *S*))
$\cup$ *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* $\sim$ *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 dest*!: *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*)) ⊆ *atms-of-ms A* **and**
atm-of ‘ *lits-of-l* (*trail* (*fst T*)) ⊆ *atms-of-ms A* **and**
finite A
shows μ_{CDCL}*'-merged A* (*fst V*) ≤ 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)) $\leq$ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ U)) $\cup$ *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)) $\cup$ *simple-clss* (*atms-of-ms* A) $\subseteq$ *set-mset* (*not-simplified-cls* ($clauses_{NOT}$ S)) $\cup$ *simple-clss* (*atms-of-ms* A)
using *U-S* **by** *auto*
then have *f2*:
card (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ U)) $\cup$ *simple-clss* (*atms-of-ms* A))
$\leq$ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ S)) $\cup$ *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)) $\cup$ *simple-clss* (*atms-of-ms* A))
$\leq$ *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)) $\leq$ *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))
$\leq$ *card* (*set-mset* (*not-simplified-cls* ($clauses_{NOT}$ S))) + *3* ^ *card* (*atms-of-ms* A)
by *linarith*
then show *?case* **unfolding** μ_{CDCL}*'-merged-def* **by** *auto*
qed

lemma $cdcl_{NOT}$*-restart-*μ_{CDCL}*'-bound-le-*μ_{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) $\subseteq$# *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* ⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I* ⊨*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* ⊨*sextm clauses*$_{NOT}$ *T* ⟷ *I* ⊨*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*⊨*sextm clauses*$_{NOT}$ (*fst S*) ⟷ *I* ⊨*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-marked-decomposition* (*trail* (*fst S*)))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst T*))
(*get-all-marked-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-marked-decomposition* (*trail* (*fst S*)))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ (*fst T*))
(*get-all-marked-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-marked-decomposition* (*trail* (*fst* S))) **and**
atms-cls: *atms-of-mm* (*clauses*$_{NOT}$ (*fst* S)) $\subseteq$ *atms-of-ms* A **and**
atms-trail: *atm-of* ' *lits-of-l* (*trail* (*fst* S)) $\subseteq$ *atms-of-ms* A **and**
fin: *finite* A
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst* S)))
$\vee$ *lits-of-l* (*trail* (*fst* T)) $\models$*sextm clauses*$_{NOT}$ (*fst* S) $\wedge$ *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)) $\subseteq$ *atms-of-ms* A **and**
atms-trail-T: *atm-of* ' *lits-of-l* (*trail* (*fst* T)) $\subseteq$ *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-marked-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)))
$\vee$ *trail* (*fst* T) $\models$*asm clauses*$_{NOT}$ (*fst* T) $\wedge$ *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) $\models$*asm clauses*$_{NOT}$ (*fst* T) **and** *satisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst* T)))
by *auto*
then show *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ (*fst* S)))
$\vee$ *lits-of-l* (*trail* (*fst* T)) $\models$*sextm clauses*$_{NOT}$ (*fst* S) $\wedge$ *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*)) $\models$*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*)) $\models$*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*)
$\vee$ *lits-of-l* (*trail* (*fst* *T*)) $\models$*sextm* *N* $\wedge$ *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

17 DPLL as an instance of NOT

17.1 DPLL with simple backtrack

We are using a concrete couple instead of an abstract state.

locale *dpll-with-backtrack*
begin
inductive *backtrack* :: ('*v*, *unit*, *unit*) *marked-lit list* $\times$ '*v clauses*
$\Rightarrow$ ('*v*, *unit*, *unit*) *marked-lit list* $\times$ '*v clauses* $\Rightarrow$ *bool* **where**
backtrack-split (*fst* *S*) = (*M'*, *L* # *M*) $\Longrightarrow$ *is-marked* *L* $\Longrightarrow$ *D* $\in$# *snd* *S*
$\Longrightarrow$ *fst* *S* $\models$*as* *CNot* *D* $\Longrightarrow$ *backtrack* *S* (*Propagated* ($-$ (*lit-of* *L*)) () # *M*, *snd* *S*)

inductive-cases *backtrackE*[*elim*]: *backtrack* (*M*, *N*) (*M'*, *N'*)
lemma *backtrack-is-backjump*:
fixes *M* *M'* :: ('*v*, *unit*, *unit*) *marked-lit list*
assumes
backtrack: *backtrack* (*M*, *N*) (*M'*, *N'*) **and**
no-dup: (*no-dup* $\circ$ *fst*) (*M*, *N*) **and**
decomp: *all-decomposition-implies-m* *N* (*get-all-marked-decomposition* *M*)
shows
$\exists$ *C* *F'* *K* *F* *L* *l* *C'*.
M = *F'* @ *Marked* *K* () # *F* $\wedge$
M' = *Propagated* *L* *l* # *F* $\wedge$ *N* = *N'* $\wedge$ *C* $\in$# *N* $\wedge$ *F'* @ *Marked* *K* *d* # *F* $\models$*as* *CNot* *C* $\wedge$
undefined-lit *F* *L* $\wedge$ *atm-of* *L* $\in$ *atms-of-mm* *N* $\cup$ *atm-of* ‘ *lits-of-l* (*F'* @ *Marked* *K* *d* # *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-marked L and
  D ∈# snd ?S and
  M ⊨as CNot D and
  bt: backtrack ?S (Propagated (− (lit-of L)) P # F, N) and
  M': M' = Propagated (− (lit-of L)) P # F and
  [simp]: N' = N
using backtrackE[OF backtrack] by (metis backtrack fstI sndI)
let ?K = lit-of L
let ?C = image-mset lit-of {#K∈#mset M. is-marked K ∧ K≠L#} :: 'v literal multiset
let ?C' = set-mset (image-mset single (?C+{#?K#}))
obtain K where L: L = Marked K () using ‹is-marked L› by (cases L) auto

have M: M = F' @ Marked K () # F
  using b-sp  by (metis L backtrack-split-list-eq fst-conv snd-conv)
moreover have F' @ Marked K () # F ⊨as CNot D
  using ‹M⊨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-of (−K) ∈ atms-of-mm N ∪ atm-of ‘ lits-of-l (F' @ Marked K d # F)
  by auto
moreover
  have set-mset N ∪ ?C' ⊨ps {{#}}
    proof −
      have A: set-mset N ∪ ?C' ∪ unmark-l M  =
        set-mset  N ∪ unmark-l M
        unfolding M L by auto
      have set-mset  N ∪ {{#lit-of L#} |L. is-marked L ∧ L ∈ set M}
          ⊨ps unmark-l M
        using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
      moreover have C': ?C' = {{#lit-of L#} |L. is-marked L ∧ L ∈ set M}
        unfolding M L apply standard
          apply force
        using IntI by auto
      ultimately have N-C-M: set-mset N ∪ ?C' ⊨ps unmark-l M
        by auto
      have set-mset N ∪ (λL. {#lit-of L#}) ‘ (set M) ⊨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 ⊨ D
            using I-N-M ‹D ∈# snd ?S› unfolding true-clss-def by auto
          moreover have I ⊨s CNot D
            using ‹M ⊨as CNot D› unfolding M by (metis 1(3) ‹M ⊨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 ⊨pm 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* (*set-mset N* ∪ {*image-mset uminus ?C* + {#− *?K*#}})) **and**
cons: *consistent-interp I* **and**
I ⊨*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* ∈ *set M*. *is-marked a* ∧ *a* ≠ *Marked K* ()} =
set M ∩ {*L*. *is-marked L* ∧ *L* ≠ *Marked K* ()}
by *auto*
then have
tI: *total-over-set I* (*atm-of* ‘ *lit-of* ‘ (*set M* ∩ {*L*. *is-marked L* ∧ *L* ≠ *Marked K d*}))
using *tot* **by** (*auto simp add*: *L atms-of-uminus-lit-atm-of-lit-of*)

then have *H*: ⋀*x*.
lit-of x ∉ *I* ⟹ *x* ∈ *set M* ⟹*is-marked x*
⟹ *x* ≠ *Marked K d* ⟹ −*lit-of x* ∈ *I*
proof −
fix *x* :: (′*v*, *unit*, *unit*) *marked-lit*
assume *a1*: *x* ≠ *Marked K d*
assume *a2*: *is-marked x*
assume *a3*: *x* ∈ *set M*
assume *a4*: *lit-of x* ∉ *I*
have *atm-of* (*lit-of x*) ∈ *atm-of* ‘ *lit-of* ‘
(*set M* ∩ {*m*. *is-marked m* ∧ *m* ≠ *Marked K d*})
using *a3 a2 a1* **by** *blast*
then have *Pos* (*atm-of* (*lit-of x*)) ∈ *I* ∨ *Neg* (*atm-of* (*lit-of x*)) ∈ *I*
using *tI* **unfolding** *total-over-set-def* **by** *blast*
then show − *lit-of x* ∈ *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 ¬*I* ⊨*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 ‹$(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$›
unfolding *L* **by** (*auto dest*!: *H*)
qed
moreover
have *set F*′ ∩ {*K*. *is-marked K* ∧ *K* ≠ *L*} = {}
using *backtrack-split-fst-not-marked*[*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*, *unit*) *marked-lit list*
assumes
backtrack: *backtrack S T* **and**

no-dup: (*no-dup* ∘ *fst*) *S* **and**
decomp: *all-decomposition-implies-m* (*snd S*) (*get-all-marked-decomposition* (*fst S*))
shows
∃ *C F′ K F L l C′*.
fst S = *F′* @ *Marked 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*
id λ*L C*. *C* + {#*L*#} *remove1-mset*
id op + *op* ∈# λ*L C*. *C* + {#*L*#} *remove1-mset*
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*
id λ*L C*. *C* + {#*L*#} *remove1-mset*
id op + *op* ∈# λ*L C*. *C* + {#*L*#} *remove1-mset*
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*

lemma *reduce-trail-to*$_{NOT}$*-snd*:
snd (*reduce-trail-to*$_{NOT}$ *F S*) = *snd S*
apply (*induction F S rule*: *reduce-trail-to*$_{NOT}$*.induct*)
by (*cases S*, *rename-tac F Sa*, *case-tac Sa*)
(*simp add*: *less-imp-diff-less reduce-trail-to*$_{NOT}$*.simps*)

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 *auto*
also have (*fst ?R*, *snd ?R*) = *?C*
by (*auto simp*: *trail-reduce-trail-to*$_{NOT}$*-drop reduce-trail-to*$_{NOT}$*-snd*)
finally show *?thesis* **.**
qed

lemma *backtrack-is-backjump′′*:
fixes *M M′* :: (′*v*, *unit*, *unit*) *marked-lit list*
assumes
backtrack: *backtrack S T* **and**
no-dup: (*no-dup* ∘ *fst*) *S* **and**
decomp: *all-decomposition-implies-m* (*snd S*) (*get-all-marked-decomposition* (*fst S*))
shows *backjump S T*
proof −
obtain *C F′ K F L l C′* **where**
1: *fst S* = *F′* @ *Marked K* () # *F* **and**

2: $T = (Propagated\ L\ l\ \#\ F,\ snd\ S)$ **and**
3: $C \in\#\ snd\ S$ **and**
4: $fst\ S \models as\ CNot\ C$ **and**
5: *undefined-lit F L* **and**
6: $atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (snd\ S) \cup atm\text{-}of\ \text{`}\ lits\text{-}of\text{-}l\ (fst\ S)$ **and**
7: $snd\ S \models pm\ C' + \{\#L\#\}$ **and**
8: $F \models as\ CNot\ C'$
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: $fst\ S = F'\ @\ Marked\ K\ d\ \#\ F$ **and**
$C \in\#\ snd\ S$ **and**
C: $fst\ S \models as\ CNot\ C$
shows ¬ *no-step backtrack S*
proof −
obtain *L G' G* **where**
backtrack-split (*fst S*) = (*G'*, *L* # *G*)
unfolding *M* **by** (*induction F' rule*: *marked-lit-list-induct*) *auto*
moreover then have *is-marked L*
by (*metis backtrack-split-snd-hd-marked list.distinct*(*1*) *list.sel*(*1*) *snd-conv*)
ultimately show *?thesis*
using *backtrack.intros*[*of S G' L G C*] ‹$C \in\#\ snd\ S$› *C* **unfolding** *M* **by** *auto*
qed

end

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping-ops*
id $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
id op + *op* $\in\#$ $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
fst snd $\lambda L\ (M,\ N).\ (L\ \#\ M,\ N)$
$\lambda(M,\ N).\ (tl\ M,\ N)$ $\lambda C\ (M,\ N).\ (M,\ \{\#C\#\} + N)$ $\lambda C\ (M,\ N).\ (M,\ removeAll\text{-}mset\ C\ N)$
$\lambda(M,\ N).$ *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-marked-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 id-apply)

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping*
id $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
id op + *op* $\in\#$ $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
fst snd $\lambda L\ (M,\ N).\ (L\ \#\ M,\ N)$
$\lambda(M,\ N).\ (tl\ M,\ N)$ $\lambda C\ (M,\ N).\ (M,\ \{\#C\#\} + N)$ $\lambda C\ (M,\ N).\ (M,\ removeAll\text{-}mset\ C\ N)$
$\lambda(M,\ N).$ *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-marked-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
term *learn*
end

context *dpll-with-backtrack*
begin
lemma *wf-tranclp-dpll-inital-state*:
assumes *fin*: *finite A*
shows *wf* $\{((M'::('v, unit, unit)\ marked\text{-}lits, N'::'v\ clauses), ([], N))|M'\ N'\ N.$
$dpll\text{-}bj^{++}\ ([], N)\ (M', N') \wedge atms\text{-}of\text{-}mm\ N \subseteq atms\text{-}of\text{-}ms\ A\}$
using *wf-tranclp-dpll-bj*[*OF assms*(*1*)] **by** (*rule wf-subset*) *auto*

corollary *full-dpll-final-state-conclusive*:
fixes $M\ M' :: ('v, unit, unit)\ marked\text{-}lit\ list$
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 $M\ M' :: ('v, unit, unit)\ marked\text{-}lit\ list$
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*
id $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
id op + *op* $\in\#$ $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
fst snd $\lambda L\ (M, N).\ (L\ \#\ M, N)$
$\lambda(M, N).\ (tl\ M, N)\ \lambda C\ (M, N).\ (M, \{\#C\#\} + N)\ \lambda C\ (M, N).\ (M, removeAll\text{-}mset\ C\ N)$
$\lambda(M, N).$ *no-dup M* $\wedge$ *all-decomposition-implies-m N* (*get-all-marked-decomposition M*)
λ- - - *S T*. *backtrack S T*
λ- -. *True* λ- -. *False* λ- -. *False*
by *unfold-locales*

interpretation *conflict-driven-clause-learning*
id $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
id op + *op* $\in\#$ $\lambda L\ C.\ C + \{\#L\#\}$ *remove1-mset*
fst snd $\lambda L\ (M, N).\ (L\ \#\ M, N)$
$\lambda(M, N).\ (tl\ M, N)\ \lambda C\ (M, N).\ (M, \{\#C\#\} + N)\ \lambda C\ (M, N).\ (M, removeAll\text{-}mset\ C\ N)$
$\lambda(M, N).$ *no-dup M* $\wedge$ *all-decomposition-implies-m N* (*get-all-marked-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* ⟷ *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* ∧ $cdcl_{NOT}$*-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

17.2 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* ⇒ *nat*
assumes *unbounded*: *unbounded f* **and** *f-ge-1*:⋀*n*. *n*≥ *1* ⟹ *f n* ≥ *1*
begin
sublocale $cdcl_{NOT}$*-increasing-restarts*
id λ*L C*. *C* + {#*L*#} *remove1-mset*
id op + *op* ∈# λ*L C*. *C* + {#*L*#} *remove1-mset*
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*) *f* λ(-, *N*) *S*. *S* = ([], *N*)
λ*A* (*M*, *N*). *atms-of-mm N* ⊆ *atms-of-ms A* ∧ *atm-of* ‘ *lits-of-l M* ⊆ *atms-of-ms A* ∧ *finite A*
∧ *all-decomposition-implies-m N* (*get-all-marked-decomposition M*)
λ*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*) *dpll-bj*
λ(*M*, *N*). *no-dup M* ∧ *all-decomposition-implies-m N* (*get-all-marked-decomposition M*)
λ*A* -. (*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

18 DPLL

18.1 Rules

type-synonym *'a* $dpll_W$*-marked-lit* = (*'a*, *unit*, *unit*) *marked-lit*
type-synonym *'a* $dpll_W$*-marked-lits* = (*'a*, *unit*, *unit*) *marked-lits*
type-synonym *'v* $dpll_W$*-state* = *'v* $dpll_W$*-marked-lits* × *'v clauses*

abbreviation *trail* :: *'v* $dpll_W$*-state* ⇒ *'v* $dpll_W$*-marked-lits* **where**
trail ≡ *fst*
abbreviation *clauses* :: *'v* $dpll_W$*-state* ⇒ *'v clauses* **where**
clauses ≡ *snd*

The definition of DPLL is given in figure 2.13 page 70 of CW.

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* (*Marked L* () # *trail S*, *clauses S*) |
backtrack: *backtrack-split* (*trail S*) = (*M'*, *L* # *M*) ⟹ *is-marked L* ⟹ *D* ∈# *clauses S*
⟹ *trail S* ⊨*as CNot D* ⟹ $dpll_W$ *S* (*Propagated* (− (*lit-of L*)) () # *M*, *clauses S*)

18.2 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** *marked* = *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* $L \notin$ *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) $\notin$ ($\lambda m.$ *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* (($\lambda a.$ {#*lit-of* a#}) ' c) = *atm-of* ' *lit-of* ' c
unfolding *atms-of-ms-def* **using** *image-iff* **by** *force*

Lemma theorem 2.8.2 page 71 of CW

lemma $dpll_W$*-propagate-is-conclusion*:
assumes $dpll_W$ S S'
and *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-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-marked-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* ($\lambda a.$ {#*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-marked-decomposition* (*trail* S) = []
then have *?case* **by** *blast*
}

moreover {
  assume *n*: *get-all-marked-decomposition* (*trail S*) ≠ []
  have *1*: ⋀*a b*. (*a*, *b*) ∈ *set* (*tl* (*get-all-marked-decomposition* (*trail S*)))
    ⟹ (*unmark-l a* ∪ *set-mset* (*clauses S*)) ⊨*ps unmark-l b*
    using *IH* **unfolding** *all-decomposition-implies-def* **by** (*fastforce simp add*: *list.set-sel*(*2*) *n*)
  moreover have *2*: ⋀*a c*. *hd* (*get-all-marked-decomposition* (*trail S*)) = (*a*, *c*)
    ⟹ (*unmark-l a* ∪ *set-mset* (*clauses S*)) ⊨*ps* (*unmark-l c*)
    by (*metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single*
      list.collapse n)
  moreover have *3*: ⋀*a c*. *hd* (*get-all-marked-decomposition* (*trail S*)) = (*a*, *c*)
    ⟹ (*unmark-l a* ∪ *set-mset* (*clauses S*)) ⊨*p* {#*L*#}
    proof −
      fix *a c*
      assume *h*: *hd* (*get-all-marked-decomposition* (*trail S*)) = (*a*, *c*)
      have *h′*: *trail S* = *c* @ *a* **using** *get-all-marked-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-marked-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** *marked* = *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-marked l*
    using *extracted backtrack-split-fst-not-marked*[*of* - *trail S*] **by** *simp*
  have *n*: *get-all-marked-decomposition* (*trail S*) ≠ [] **by** *auto*
  then have *all-decomposition-implies-m* (*clauses S*) ((*L* # *M*, *M′*)
        # *tl* (*get-all-marked-decomposition* (*trail S*)))
    by (*metis* (*no-types*) *IH extracted get-all-marked-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-marked-decomposition*
(*fst* (*Propagated* (− *lit-of L*) *P* # *M*, *clauses S*)))
let *?M'* = *Propagated* (− *lit-of L*) *P* # *M*
let *?hd* = *hd* (*get-all-marked-decomposition ?M'*)
let *?tl* = *tl* (*get-all-marked-decomposition ?M'*)
have *x* = *?hd* ∨ *x* ∈ *set ?tl*
using *x*
by (*cases get-all-marked-decomposition ?M'*)
auto
moreover {
assume *x'*: *x* ∈ *set ?tl*
have *L'*: *Marked* (*lit-of L*) () = *L* **using** *marked* **by** (*cases L*, *auto*)
have *x* ∈ *set* (*get-all-marked-decomposition* (*M'* @ *L* # *M*))
using *x' get-all-marked-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 *marked IH* **by** (*cases L*) (*auto simp add*: *S all-decomposition-implies-def*)
}
moreover {
assume *x'*: *x* = *?hd*
have *tl*: *tl* (*get-all-marked-decomposition* (*M'* @ *L* # *M*)) ≠ []
proof −
have *f1*: ⋀*ms*. *length* (*get-all-marked-decomposition* (*M'* @ *ms*))
= *length* (*get-all-marked-decomposition ms*)
by (*simp add*: *M' get-all-marked-decomposition-remove-unmark-ssed-length*)
have *Suc* (*length* (*get-all-marked-decomposition M*)) ≠ *Suc 0*
by *blast*
then show *?thesis*
using *f1 marked* **by** (*metis* (*no-types*) *get-all-marked-decomposition.simps*(*1*) *length-tl*
list.sel(*3*) *list.size*(*3*) *marked-lit.collapse*(*1*))
qed
obtain *M0' M0* **where**
L0: *hd* (*tl* (*get-all-marked-decomposition* (*M'* @ *L* # *M*))) = (*M0*, *M0'*)
by (*cases hd* (*tl* (*get-all-marked-decomposition* (*M'* @ *L* # *M*))))
have *x''*: *x* = (*M0*, *Propagated* (−*lit-of L*) *P* # *M0'*)
unfolding *x'* **using** *get-all-marked-decomposition-last-choice tl M' L0*
by (*metis marked marked-lit.collapse*(*1*))
obtain *l-get-all-marked-decomposition* **where**
get-all-marked-decomposition (*trail S*) = (*L* # *M*, *M'*) # (*M0*, *M0'*) #
l-get-all-marked-decomposition

using *get-all-marked-decomposition-backtrack-split extracted* **by** (*metis* (*no-types*) *L0 S hd-Cons-tl n tl*)
then have $M = M0' @ M0$ **using** *get-all-marked-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

Lemma theorem 2.8.3 page 72 of CW

theorem *dpll*$_W$*-propagate-is-conclusion-of-decided*:
assumes *dpll*$_W$ *S S'*
and *all-decomposition-implies-m* (*clauses S*) (*get-all-marked-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-marked L* ∧ *L* ∈ *set* (*trail S'*)}
⊨*ps* (λ*a*. {#*lit-of a*#}) ' ⋃(*set* ' *snd* ' *set* (*get-all-marked-decomposition* (*trail S'*)))
using *all-decomposition-implies-trail-is-implied*[*OF dpll*$_W$*-propagate-is-conclusion*[*OF assms*]] **.**

Lemma theorem 2.8.4 page 72 of CW

lemma *only-propagated-vars-unsat*:
assumes *marked*: ∀ *x* ∈ *set M*. ¬ *is-marked x*
and *DN*: *D* ∈ *N* **and** *D*: *M* ⊨*as CNot D*
and *inv*: *all-decomposition-implies N* (*get-all-marked-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-marked L* ∧ *L* ∈ *set M*} = {} **using** *marked* **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*: $I \models s$ *unmark-l M* **by** *auto*
  {
    fix *K*
    assume $K \in\# D$
    then have $-K \in$ *lits-of-l M*
      by (*auto split*: *if-split-asm*
        intro: *allE*[*OF D*[*unfolded true-annots-def Ball-def*], *of* {#−*K*#}])
    then have $-K \in I$ **using** *IM true-clss-singleton-lit-of-implies-incl* **by** *fastforce*
  }
  then have $\neg\ I \models D$ **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-marked-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-marked-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-marked-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-marked-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-marked-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-marked-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'') $\subseteq$ *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* $S \equiv$
(*all-decomposition-implies-m* (*clauses* S) (*get-all-marked-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-marked-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* (λM. *Marked* M ()) M, N)
using *assms*
proof (*induct* M)
case *Nil*
then show *?case* **by** *auto*
next

case (*Cons L M*)
then have *undefined-lit* (*map* (λM. *Marked M* ()) *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* (λM. *Marked M* ()) *M*, *N*) (*map* (λM. *Marked M* ()) (*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-marked L*)
$\wedge$ ($\exists$ *C* $\in$# *clauses S*. *trail S* $\models$*as CNot C*)))

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* (λM. *Marked M* ()) *M*, *N*)
and *conclusive-*$dpll_W$*-state* (*map* (λM. *Marked M* ()) *M*, *N*)
proof −
show *rtranclp* $dpll_W$ ([], *N*) (*map* (λM. *Marked M* ()) *M*, *N*) **using** $dpll_W$*-can-do-step assms* **by** *auto*
have *map* (λM. *Marked M* ()) *M* $\models$*asm N* **using** *assms*(*1*) *true-annots-marked-true-cls* **by** *auto*
then show *conclusive-*$dpll_W$*-state* (*map* (λM. *Marked M* ()) *M*, *N*)
unfolding *conclusive-*$dpll_W$*-state-def* **by** *auto*
qed

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-marked-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-marked 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*: $\exists$ *l* $\in$ *set M. is-marked l*
then have *False*
using *backtrack*[*of* (*M*, *N*) - - - *D*] *DN MD assms*(*2*)
backtrack-split-some-is-marked-then-snd-has-hd[*OF l*]
by (*metis backtrack-split-snd-hd-marked fst-conv list.distinct*(*1*) *list.sel*(*1*) *snd-conv*)
}
ultimately have *False* **by** *blast*
}
ultimately show *False* **by** *blast*
qed
qed

18.3 Termination

definition $dpll_W$*-mes M n* =
map (λ*l. if is-marked l then 2 else* (*1::nat*)) (*rev M*) @ *replicate* (*n* $-$ *length M*) *3*

lemma *length-*$dpll_W$*-mes*:
assumes *length M* $\leq$ *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'*) $\leq$ *card vars*
and *length* (*trail S*) $\leq$ *card vars*
shows ($dpll_W$*-mes* (*trail S'*) (*card vars*), $dpll_W$*-mes* (*trail S*) (*card vars*))
$\in$ *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-marked l then 2 else 1*) (*rev* (*trail S*))

@ *replicate* (*card vars* − *length* (*trail S*)) *3*
= *map* (*λl. if is-marked 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* (*λl. if is-marked l then 2 else 1*) (*rev* (*trail S*))
@ *replicate* (*card vars* − *length* (*trail S*)) *3*
= *map* (*λl. if is-marked 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-marked 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

Proposition theorem 2.8.7 page 73 of CW

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*). $dpll_W$*-all-inv S* ∧ $dpll_W$ *S* S'}$^+$ = {(S', *S*). $dpll_W$*-all-inv S* ∧ *tranclp* $dpll_W$ *S* S'}
(**is** *?A* = *?B*)
proof
{ **fix** *S* S'
assume (*S*, S') ∈ *?A*
then have (*S*, S') ∈ *?B*
by (*induct rule*: *trancl.induct*, *auto*)
}
then show *?A* ⊆ *?B* **by** *blast*
{ **fix** *S* S'
assume (*S*, S') ∈ *?B*
then have $dpll_W{}^{++}$ S' *S* **and** $dpll_W$*-all-inv* S' **by** *auto*
then have (*S*, S') ∈ *?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*) ∈ {*a*. *case a of* (S', *S*) ⇒ $dpll_W$*-all-inv S* ∧ $dpll_W$ *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') ∈ {(*pa*, *p*). $dpll_W$*-all-inv p* ∧ $dpll_W$ *p pa*}$^+$
using ‹$dpll_W$*-all-inv* S'› *trancl-into-trancl.hyps*(*3*) **by** *blast*
then show *?case*
using ‹(S', *S*) ∈ {*a*. *case a of* (S', *S*) ⇒ $dpll_W$*-all-inv S* ∧ $dpll_W$ *S* S'}$^+$› **by** *auto*
qed
}
then show *?B* ⊆ *?A* **by** *blast*
qed

lemma $dpll_W$*-wf-tranclp*: *wf* {(S', *S*). $dpll_W$*-all-inv S* ∧ $dpll_W{}^{++}$ *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*

18.4 Final States

```
lemma dpll_W-no-more-step-is-a-conclusive-state:
  assumes ∀ S'. ¬dpll_W S S'
  shows conclusive-dpll_W-state S
proof −
  have vars: ∀ s ∈ atms-of-mm (clauses S). s ∈ atm-of ` lits-of-l (trail S)
    proof (rule ccontr)
      assume ¬ (∀ s∈atms-of-mm (clauses S). s ∈ atm-of ` lits-of-l (trail S))
      then obtain L where
        L-in-atms: L ∈ atms-of-mm (clauses S) and
        L-notin-trail: L ∉ atm-of ` lits-of-l (trail S) by metis
      obtain L' where L': atm-of L' = L by (meson literal.sel(2))
      then have undefined-lit (trail S) L'
        unfolding Marked-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-marked L) ∨ (∀ C∈#clauses S. ¬trail S ⊨as CNot C)
        unfolding conclusive-dpll_W-state-def by auto
      moreover {
        assume ∃ L∈set (trail S). is-marked L
        then obtain L M' M where L: backtrack-split (trail S) = (M', L # M)
          using backtrack-split-some-is-marked-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-marked L
          using L by (metis backtrack-split-snd-hd-marked 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'* $\models$*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 *no-mark*: $\forall$ *L*$\in$*set M*. $\neg$ *is-marked L* $\exists$ *C* $\in\#$ *N*. *M* $\models$*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

18.5 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

18.5.1 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.

fun *get-rev-level* :: (*'v*, *nat*, *'a*) *marked-lits* ⇒ *nat* ⇒ *'v literal* ⇒ *nat* **where**
get-rev-level [] - - = *0* |
get-rev-level (*Marked l level* # *Ls*) *n L* =
(*if atm-of l* = *atm-of L then level else get-rev-level Ls level L*) |
get-rev-level (*Propagated l -* # *Ls*) *n L* =
(*if atm-of l* = *atm-of L then n else get-rev-level Ls n L*)

abbreviation *get-level M L* ≡ *get-rev-level* (*rev M*) *0 L*

lemma *get-rev-level-uminus*[*simp*]: *get-rev-level M n*(−*L*) = *get-rev-level M n L*
by (*induct arbitrary*: *n rule*: *get-rev-level.induct*) *auto*

lemma *atm-of-notin-get-rev-level-eq-0*:
assumes *atm-of L* ∉ *atm-of* ‘ *lits-of-l M*
shows *get-rev-level M n L* = *0*
using *assms* **by** (*induct M arbitrary*: *n rule*: *marked-lit-list-induct*) *auto*

lemma *get-rev-level-ge-0-atm-of-in*:
assumes *get-rev-level M n L > n*
shows *atm-of L ∈ atm-of ' lits-of-l M*
using *assms* **by** (*induct M arbitrary*: *n rule*: *marked-lit-list-induct*)
(*fastforce simp*: *atm-of-notin-get-rev-level-eq-0*)+

In *get-rev-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-rev-level* (*M @ Marked K i # M'*) *n L* = *get-rev-level* (*Marked K i # M'*) *i L*
using *assms* **by** (*induct M arbitrary*: *n i rule*: *marked-lit-list-induct*) *auto*

lemma *get-rev-level-notin-end*[*simp*]:
assumes *atm-of L ∉ atm-of ' lits-of-l M'*
shows *get-rev-level* (*M @ M'*) *n L* = *get-rev-level M n L*
using *assms* **by** (*induct M arbitrary*: *n rule*: *marked-lit-list-induct*)
(*auto simp*: *atm-of-notin-get-rev-level-eq-0*)

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-rev-level* (*M @ M'*) *n L* = *get-rev-level M n L*
using *assms* **by** (*induct arbitrary*: *n rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-beginning*:
assumes *atm-of L' ≠ 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-marked-rev*:
assumes *atm-of L ∉ atm-of ' lit-of '*(*set S*)
and ∀ *s*∈*set S*. ¬*is-marked s*
shows *get-level* (*M @ rev S*) *L* = *get-level M L*
using *assms* **by** (*induction S rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-beginning-not-marked*[*simp*]:
assumes *atm-of L ∉ atm-of ' lit-of '*(*set S*)
and ∀ *s*∈*set S*. ¬*is-marked s*
shows *get-level* (*M @ S*) *L* = *get-level M L*
using *get-level-skip-beginning-not-marked-rev*[*of L rev S M*] *assms* **by** *auto*

lemma *get-rev-level-skip-beginning-not-marked*[*simp*]:
assumes *atm-of L ∉ atm-of ' lit-of '*(*set S*)
and ∀ *s*∈*set S*. ¬*is-marked s*
shows *get-rev-level* (*rev S @ rev M*) *0 L* = *get-level M L*
using *get-level-skip-beginning-not-marked-rev*[*of L rev S M*] *assms* **by** *auto*

lemma *get-level-skip-in-all-not-marked*:
fixes *M* :: (*'a*, *nat*, *'b*) *marked-lit list* **and** *L* :: *'a literal*
assumes ∀ *m*∈*set M*. ¬ *is-marked m*
and *atm-of L ∈ atm-of ' lit-of '* (*set M*)
shows *get-rev-level M n L = n*
using *assms* **by** (*induction M rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-all-not-marked*[*simp*]:
  fixes *M*
  defines $M' \equiv rev\ M$
  assumes $\forall m \in set\ M.\ \neg\ is\text{-}marked\ m$
  shows $get\text{-}level\ M\ L = 0$
proof −
  have *M*: $M = rev\ M'$
    unfolding *M′-def* **by** *auto*
  show *?thesis*
    using *assms* **unfolding** *M* **by** (*induction M′ rule*: *marked-lit-list-induct*) *auto*
qed

abbreviation $MMax\ M \equiv Max\ (set\text{-}mset\ M)$

the $\{\#0::'a\#\}$ is there to ensures that the set is not empty.

definition *get-maximum-level* :: $('a,\ nat,\ 'b)\ marked\text{-}lit\ list \Rightarrow\ 'a\ literal\ multiset \Rightarrow nat$
  where
$get\text{-}maximum\text{-}level\ M\ D = MMax\ (\{\#0\#\} + image\text{-}mset\ (get\text{-}level\ M)\ D)$

lemma *get-maximum-level-ge-get-level*:
  $L \in\#\ D \Longrightarrow get\text{-}maximum\text{-}level\ M\ D \geq get\text{-}level\ M\ L$
  unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-empty*[*simp*]:
  $get\text{-}maximum\text{-}level\ M\ \{\#\} = 0$
  unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-exists-lit-of-max-level*:
  $D \neq \{\#\} \Longrightarrow \exists L \in\#\ D.\ get\text{-}level\ M\ L = get\text{-}maximum\text{-}level\ M\ D$
  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*]:
  $get\text{-}maximum\text{-}level\ []\ D = 0$
  unfolding *get-maximum-level-def* **by** (*simp add*: *image-constant-conv*)

lemma *get-maximum-level-single*[*simp*]:
  $get\text{-}maximum\text{-}level\ M\ \{\#L\#\} = get\text{-}level\ M\ L$
  unfolding *get-maximum-level-def* **by** *simp*

lemma *get-maximum-level-plus*:
  $get\text{-}maximum\text{-}level\ M\ (D + D') = max\ (get\text{-}maximum\text{-}level\ M\ D)\ (get\text{-}maximum\text{-}level\ M\ D')$
  by (*induct D*) (*auto simp add*: *get-maximum-level-def*)

lemma *get-maximum-level-exists-lit*:
  assumes *n*: $n > 0$
  and *max*: $get\text{-}maximum\text{-}level\ M\ D = n$
  shows $\exists L \in\#D.\ get\text{-}level\ M\ L = n$
proof −
  have *f*: $finite\ (insert\ 0\ ((\lambda L.\ get\text{-}level\ M\ L)\ `\ set\text{-}mset\ D))$ **by** *auto*
  then have $n \in ((\lambda L.\ get\text{-}level\ M\ L)\ `\ set\text{-}mset\ D)$
    using *n max Max-in*[*OF f*] **unfolding** *get-maximum-level-def* **by** *simp*

then show $\exists L \in\# D$. *get-level M L* $= n$ **by** *auto*
qed

lemma *get-maximum-level-skip-first*[*simp*]:
assumes *atm-of L* $\notin$ *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 marked-lit.sel*(*2*)
multiset.map-cong0)

lemma *get-maximum-level-skip-beginning*:
assumes *DH*: *atms-of D* $\subseteq$ *atm-of ‘lits-of-l H*
shows *get-maximum-level* (*c* @ *Marked Kh i* # *H*) *D* = *get-maximum-level H D*
proof −
have (*get-rev-level* (*rev H* @ *Marked Kh i* # *rev c*) *0*) ‘ *set-mset D*
= (*get-rev-level* (*rev H*) *0*) ‘ *set-mset D*
using *DH* **unfolding** *atms-of-def*
by (*metis* (*no-types, lifting*) *get-rev-level-skip-end image-cong image-subset-iff set-rev*)
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*
proof −
have *A*: *insert 0* ((λL. *0*) ‘ (*set-mset D* $\cap$ {*L*. *atm-of x21* = *atm-of L*})
$\cup$ (λL. *0*) ‘ (*set-mset D* $\cap$ {*L*. *atm-of x21* $\neq$ *atm-of L*})) = {*0*}
by *auto*
show *?thesis* **unfolding** *get-maximum-level-def* **by** (*simp add*: *A*)
qed

lemma *get-maximum-level-skip-notin*:
assumes *D*: $\forall$ *L*$\in\#$*D*. *atm-of L* $\in$ *atm-of ‘lits-of-l M*
shows *get-maximum-level M D* = *get-maximum-level* (*Propagated x21 x22* # *M*) *D*
proof −
have *A*: (*get-rev-level* (*rev M* @ [*Propagated x21 x22*]) *0*) ‘ *set-mset D*
= (*get-rev-level* (*rev M*) *0*) ‘ *set-mset D*
using *D* **by** (*auto intro*!: *image-cong simp add*: *lits-of-def*)
show *?thesis* **unfolding** *get-maximum-level-def* **by** (*auto simp*: *A*)
qed

lemma *get-maximum-level-skip-un-marked-not-present*:
assumes $\forall$ *L*$\in\#$*D*. *atm-of L* $\in$ *atm-of* ‘ *lits-of-l aa* **and**
$\forall$ *m*$\in$*set M*. $\neg$ *is-marked m*
shows *get-maximum-level aa D* = *get-maximum-level* (*M* @ *aa*) *D*
using *assms* **by** (*induction M rule*: *marked-lit-list-induct*)
(*auto intro*!: *get-maximum-level-skip-notin*[*of D - @ aa*] *simp add*: *image-Un*)

lemma *get-maximum-level-union-mset*:
get-maximum-level M (*A* #$\cup$ *B*) = *get-maximum-level M* (*A* + *B*)
unfolding *get-maximum-level-def* **by** (*auto simp*: *image-Un*)

fun *get-maximum-possible-level*:: (*'b*, *nat*, *'c*) *marked-lit list* $\Rightarrow$ *nat* **where**
get-maximum-possible-level [] = *0* |
get-maximum-possible-level (*Marked K i* # *l*) = *max i* (*get-maximum-possible-level l*) |

get-maximum-possible-level (*Propagated* - - # *l*) = *get-maximum-possible-level l*

lemma *get-maximum-possible-level-append*[*simp*]:
  get-maximum-possible-level (*M*@*M′*)
    = *max* (*get-maximum-possible-level M*) (*get-maximum-possible-level M′*)
  by (*induct M rule*: *marked-lit-list-induct*) *auto*

lemma *get-maximum-possible-level-rev*[*simp*]:
  get-maximum-possible-level (*rev M*) = *get-maximum-possible-level M*
  by (*induct M rule*: *marked-lit-list-induct*) *auto*

lemma *get-maximum-possible-level-ge-get-rev-level*:
  max (*get-maximum-possible-level M*) *i* ≥ *get-rev-level M i L*
  by (*induct M arbitrary*: *i rule*: *marked-lit-list-induct*) (*auto simp add*: *le-max-iff-disj*)

lemma *get-maximum-possible-level-ge-get-level*[*simp*]:
  get-maximum-possible-level M ≥ *get-level M L*
  using *get-maximum-possible-level-ge-get-rev-level*[*of rev - 0*] **by** *auto*

lemma *get-maximum-possible-level-ge-get-maximum-level*[*simp*]:
  get-maximum-possible-level M ≥ *get-maximum-level M D*
  using *get-maximum-level-exists-lit-of-max-level* **unfolding** *Bex-def*
  by (*metis get-maximum-level-empty get-maximum-possible-level-ge-get-level le0*)

fun *get-all-mark-of-propagated* **where**
get-all-mark-of-propagated [] = [] |
get-all-mark-of-propagated (*Marked* - - # *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*: *marked-lit-list-induct*) *auto*

18.5.2 Properties about the levels

fun *get-all-levels-of-marked* :: (*′b*, *′a*, *′c*) *marked-lit list* ⇒ *′a list* **where**
get-all-levels-of-marked [] = [] |
get-all-levels-of-marked (*Marked l level* # *Ls*) = *level* # *get-all-levels-of-marked Ls* |
get-all-levels-of-marked (*Propagated* - - # *Ls*) = *get-all-levels-of-marked Ls*

lemma *get-all-levels-of-marked-nil-iff-not-is-marked*:
  get-all-levels-of-marked xs = [] ⟷ (∀ *x* ∈ *set xs*. ¬*is-marked x*)
  using *assms* **by** (*induction xs rule*: *marked-lit-list-induct*) *auto*

lemma *get-all-levels-of-marked-cons*:
  get-all-levels-of-marked (*a* # *b*) =
    (*if is-marked a then* [*level-of a*] *else* []) @ *get-all-levels-of-marked b*
  by (*cases a*) *simp-all*

lemma *get-all-levels-of-marked-append*[*simp*]:
  get-all-levels-of-marked (*a* @ *b*) = *get-all-levels-of-marked a* @ *get-all-levels-of-marked b*
  by (*induct a*) (*simp-all add*: *get-all-levels-of-marked-cons*)

lemma *in-get-all-levels-of-marked-iff-decomp*:
  i ∈ *set* (*get-all-levels-of-marked M*) ⟷ (∃ *c K c′*. *M* = *c* @ *Marked K i* # *c′*) (**is** *?A* ⟷ *?B*)
proof

assume *?B*
then show *?A* **by** *auto*
next
assume *?A*
then show *?B*
apply (*induction M rule*: *marked-lit-list-induct*)
apply *auto*[]
apply (*metis append-Cons append-Nil get-all-levels-of-marked.simps*(*2*) *set-ConsD*)
by (*metis append-Cons get-all-levels-of-marked.simps*(*3*))
qed

lemma *get-rev-level-less-max-get-all-levels-of-marked*:
get-rev-level M n L ≤ *Max* (*set* (*n* # *get-all-levels-of-marked M*))
by (*induct M arbitrary*: *n rule*: *get-all-levels-of-marked.induct*)
(*simp-all add*: *max.coboundedI2*)

lemma *get-rev-level-ge-min-get-all-levels-of-marked*:
assumes *atm-of L* ∈ *atm-of* ' *lits-of-l M*
shows *get-rev-level M n L* ≥ *Min* (*set* (*n* # *get-all-levels-of-marked M*))
using *assms* **by** (*induct M arbitrary*: *n rule*: *get-all-levels-of-marked.induct*)
(*auto simp add*: *min-le-iff-disj*)

lemma *get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked*[*simp*]:
get-all-levels-of-marked (*rev M*) = *rev* (*get-all-levels-of-marked M*)
by (*induct M rule*: *get-all-levels-of-marked.induct*)
(*simp-all add*: *max.coboundedI2*)

lemma *get-maximum-possible-level-max-get-all-levels-of-marked*:
get-maximum-possible-level M = *Max* (*insert 0* (*set* (*get-all-levels-of-marked M*)))
by (*induct M rule*: *marked-lit-list-induct*) (*auto simp*: *insert-commute*)

lemma *get-rev-level-in-levels-of-marked*:
get-rev-level M n L ∈ {*0*, *n*} ∪ *set* (*get-all-levels-of-marked M*)
by (*induction M arbitrary*: *n rule*: *marked-lit-list-induct*) (*force simp add*: *atm-of-eq-atm-of*)+

lemma *get-rev-level-in-atms-in-levels-of-marked*:
atm-of L ∈ *atm-of* ' (*lits-of-l M*) ⟹
get-rev-level M n L ∈ {*n*} ∪ *set* (*get-all-levels-of-marked M*)
by (*induction M arbitrary*: *n rule*: *marked-lit-list-induct*) (*auto simp add*: *atm-of-eq-atm-of*)

lemma *get-all-levels-of-marked-no-marked*:
(∀ *l*∈*set Ls*. ¬ *is-marked l*) ⟷ *get-all-levels-of-marked Ls* = []
by (*induction Ls*) (*auto simp add*: *get-all-levels-of-marked-cons*)

lemma *get-level-in-levels-of-marked*:
get-level M L ∈ {*0*} ∪ *set* (*get-all-levels-of-marked M*)
using *get-rev-level-in-levels-of-marked*[*of rev M 0 L*] **by** *auto*

The zero is here to avoid empty-list issues with *last*:

lemma *get-level-get-rev-level-get-all-levels-of-marked*:
assumes *atm-of L* ∉ *atm-of* ' (*lits-of-l M*)
shows
get-level (*K* @ *M*) *L* = *get-rev-level* (*rev K*) (*last* (*0* # *get-all-levels-of-marked* (*rev M*))) *L*
using *assms*
proof (*induct M arbitrary*: *K*)

case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a M*)
then have *H*: $\bigwedge$*K*. *get-level* (*K* @ *M*) *L*
= *get-rev-level* (*rev K*) (*last* (*0* # *get-all-levels-of-marked* (*rev M*))) *L*
by *auto*
have *get-level* ((*K* @ [*a*])@ *M*) *L*
= *get-rev-level* (*a* # *rev K*) (*last* (*0* # *get-all-levels-of-marked* (*rev M*))) *L*
using *H*[*of K* @ [*a*]] **by** *simp*
then show *?case* **using** *Cons*(*2*) **by** (*cases a*) *auto*
qed

lemma *get-rev-level-can-skip-correctly-ordered*:
assumes
no-dup M **and**
atm-of L $\notin$ *atm-of* ‘ (*lits-of-l M*) **and**
get-all-levels-of-marked M = *rev* [*Suc 0*..<*Suc* (*length* (*get-all-levels-of-marked M*))]
shows *get-rev-level* (*rev M* @ *K*) *0 L* = *get-rev-level K* (*length* (*get-all-levels-of-marked M*)) *L*
using *assms*
proof (*induct M arbitrary*: *K rule*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** *simp*
next
case (*marked L' i M K*)
then have
i: *i* = *Suc* (*length* (*get-all-levels-of-marked M*)) **and**
get-all-levels-of-marked M = *rev* [*Suc 0*..<*Suc* (*length* (*get-all-levels-of-marked M*))]
by *auto*
then have *get-rev-level* (*rev M* @ (*Marked L' i* # *K*)) *0 L*
= *get-rev-level* (*Marked L' i* # *K*) (*length* (*get-all-levels-of-marked M*)) *L*
using *marked* **by** *auto*
then show *?case* **using** *marked* **unfolding** *i* **by** *auto*
next
case (*proped L' D M K*)
then have *get-all-levels-of-marked M* = *rev* [*Suc 0*..<*Suc* (*length* (*get-all-levels-of-marked M*))]
by *auto*
then have *get-rev-level* (*rev M* @ (*Propagated L' D* # *K*)) *0 L*
= *get-rev-level* (*Propagated L' D* # *K*) (*length* (*get-all-levels-of-marked M*)) *L*
using *proped* **by** *auto*
then show *?case* **using** *proped* **by** *auto*
qed

lemma *get-level-skip-beginning-hd-get-all-levels-of-marked*:
assumes *atm-of L* $\notin$ *atm-of* ‘ *lits-of-l S* **and** *get-all-levels-of-marked S* $\neq$ []
shows *get-level* (*M*@ *S*) *L* = *get-rev-level* (*rev M*) (*hd* (*get-all-levels-of-marked S*)) *L*
using *assms*
proof (*induction S arbitrary*: *M rule*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** (*auto simp add*: *lits-of-def*)
next
case (*marked K m*) **note** *notin* = *this*(*2*)
then show *?case* **by** (*auto simp add*: *lits-of-def*)
next
case (*proped L l*) **note** *IH* = *this*(*1*) **and** *L* = *this*(*2*) **and** *neq* = *this*(*3*)

show *?case* **using** *IH*[*of M*@[*Propagated L l*]] *L neq* **by** (*auto simp add*: *atm-of-eq-atm-of*)
qed

end
theory *CDCL-W*
imports *CDCL-Abstract-Clause-Representation List-More CDCL-W-Level Wellfounded-More*

begin

19 Weidenbach's CDCL

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

19.1 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 $state_W$*-ops* =
raw-clss mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
+
raw-ccls-union mset-ccls union-ccls insert-ccls remove-clit
for
— Clause
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**

— Multiset of Clauses
mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**

mset-ccls:: *'ccls* ⇒ *'v clause* **and**
union-ccls :: *'ccls* ⇒ *'ccls* ⇒ *'ccls* **and**
insert-ccls :: *'v literal* ⇒ *'ccls* ⇒ *'ccls* **and**
remove-clit :: *'v literal* ⇒ *'ccls* ⇒ *'ccls*
+
fixes
ccls-of-cls :: *'cls* ⇒ *'ccls* **and**
cls-of-ccls :: *'ccls* ⇒ *'cls* **and**

trail :: *'st* ⇒ (*'v*, *nat*, *'v clause*) *marked-lits* **and**
hd-raw-trail :: *'st* ⇒ (*'v*, *nat*, *'cls*) *marked-lit* **and**
raw-init-clss :: *'st* ⇒ *'clss* **and**
raw-learned-clss :: *'st* ⇒ *'clss* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conflicting :: *'st* ⇒ *'ccls option* **and**

cons-trail :: (*'v*, *nat*, *'cls*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**

add-init-cls :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
add-learned-cls :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
remove-cls :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
update-backtrack-lvl :: *nat* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**
update-conflicting :: *'ccls option* $\Rightarrow$ *'st* $\Rightarrow$ *'st* **and**

init-state :: *'clss* $\Rightarrow$ *'st* **and**
restart-state :: *'st* $\Rightarrow$ *'st*
assumes
mset-ccls-ccls-of-cls[*simp*]:
mset-ccls (*ccls-of-cls* C) = *mset-cls* C **and**
mset-cls-cls-of-ccls[*simp*]:
mset-cls (*cls-of-ccls* D) = *mset-ccls* D **and**
ex-mset-cls: $\exists$ *a*. *mset-cls a* = E
begin
fun *mmset-of-mlit* :: (*'a*, *'b*, *'cls*) *marked-lit* $\Rightarrow$ (*'a*, *'b*, *'v clause*) *marked-lit*
where
mmset-of-mlit (*Propagated L C*) = *Propagated L* (*mset-cls C*) |
mmset-of-mlit (*Marked L i*) = *Marked L i*

lemma *lit-of-mmset-of-mlit*[*simp*]:
lit-of (*mmset-of-mlit a*) = *lit-of a*
by (*cases a*) *auto*

lemma *lit-of-mmset-of-mlit-set-lit-of-l*[*simp*]:
lit-of ‘ *mmset-of-mlit* ‘ *set M'* = *lits-of-l M'*
by (*induction M'*) *auto*

lemma *map-mmset-of-mlit-true-annots-true-cls*[*simp*]:
map mmset-of-mlit M' $\models$*as C* $\longleftrightarrow$ *M'* $\models$*as C*
by (*simp add*: *true-annots-true-cls lits-of-def*)

abbreviation *init-clss* $\equiv$ λS. *mset-clss* (*raw-init-clss S*)
abbreviation *learned-clss* $\equiv$ λS. *mset-clss* (*raw-learned-clss S*)
abbreviation *conflicting* $\equiv$ λS. *map-option mset-ccls* (*raw-conflicting S*)

notation *insert-cls* (**infix** !++ *50*)

notation *in-clss* (**infix** !$\in$! *50*)
notation *union-clss* (**infix** $\oplus$ *50*)
notation *insert-clss* (**infix** !++! *50*)

notation *union-ccls* (**infix** !$\cup$ *50*)

definition *raw-clauses* :: *'st* $\Rightarrow$ *'clss* **where**
raw-clauses S = *union-clss* (*raw-init-clss S*) (*raw-learned-clss S*)

abbreviation *clauses* :: *'st* $\Rightarrow$ *'v clauses* **where**
clauses S $\equiv$ *mset-clss* (*raw-clauses S*)

end

locale $state_W$ =
$state_W$*-ops*
— functions for clauses:

mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:
mset-ccls union-ccls insert-ccls remove-clit

— Conversion between conflicting and non-conflicting
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
— changing state:
cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl update-conflicting

— get state:
init-state
restart-state
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**

mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**

mset-ccls:: *'ccls* ⇒ *'v clause* **and**
union-ccls :: *'ccls* ⇒ *'ccls* ⇒ *'ccls* **and**
insert-ccls :: *'v literal* ⇒ *'ccls* ⇒ *'ccls* **and**
remove-clit :: *'v literal* ⇒ *'ccls* ⇒ *'ccls* **and**

ccls-of-cls :: *'cls* ⇒ *'ccls* **and**
cls-of-ccls :: *'ccls* ⇒ *'cls* **and**

trail :: *'st* ⇒ (*'v*, *nat*, *'v clause*) *marked-lits* **and**
hd-raw-trail :: *'st* ⇒ (*'v*, *nat*, *'cls*) *marked-lit* **and**
raw-init-clss :: *'st* ⇒ *'clss* **and**
raw-learned-clss :: *'st* ⇒ *'clss* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conflicting :: *'st* ⇒ *'ccls option* **and**

cons-trail :: (*'v*, *nat*, *'cls*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-init-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
add-learned-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'ccls option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'clss* ⇒ *'st* **and**
restart-state :: *'st* ⇒ *'st* +

assumes
hd-raw-trail: $\textit{trail } S \neq [] \Longrightarrow \textit{mmset-of-mlit } (\textit{hd-raw-trail } S) = \textit{hd } (\textit{trail } S)$ **and**
trail-cons-trail[*simp*]:
$\bigwedge L\ st.\ \textit{undefined-lit } (\textit{trail } st)\ (\textit{lit-of } L) \Longrightarrow$
$\textit{trail } (\textit{cons-trail } L\ st) = \textit{mmset-of-mlit } L\ \#\ \textit{trail } st$ **and**
trail-tl-trail[*simp*]: $\bigwedge st.\ \textit{trail } (\textit{tl-trail } st) = \textit{tl } (\textit{trail } st)$ **and**
trail-add-init-cls[*simp*]:
$\bigwedge st\ C.\ \textit{no-dup } (\textit{trail } st) \Longrightarrow \textit{trail } (\textit{add-init-cls } C\ st) = \textit{trail } st$ **and**
trail-add-learned-cls[*simp*]:
$\bigwedge C\ st.\ \textit{no-dup } (\textit{trail } st) \Longrightarrow \textit{trail } (\textit{add-learned-cls } C\ st) = \textit{trail } st$ **and**
trail-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{trail } (\textit{remove-cls } C\ st) = \textit{trail } st$ **and**
trail-update-backtrack-lvl[*simp*]: $\bigwedge st\ C.\ \textit{trail } (\textit{update-backtrack-lvl } C\ st) = \textit{trail } st$ **and**
trail-update-conflicting[*simp*]: $\bigwedge C\ st.\ \textit{trail } (\textit{update-conflicting } C\ st) = \textit{trail } st$ **and**

init-clss-cons-trail[*simp*]:
$\bigwedge M\ st.\ \textit{undefined-lit } (\textit{trail } st)\ (\textit{lit-of } M) \Longrightarrow$
$\textit{init-clss } (\textit{cons-trail } M\ st) = \textit{init-clss } st$
and
init-clss-tl-trail[*simp*]:
$\bigwedge st.\ \textit{init-clss } (\textit{tl-trail } st) = \textit{init-clss } st$ **and**
init-clss-add-init-cls[*simp*]:
$\bigwedge st\ C.\ \textit{no-dup } (\textit{trail } st) \Longrightarrow \textit{init-clss } (\textit{add-init-cls } C\ st) = \{\#\textit{mset-cls } C\#\} + \textit{init-clss } st$
and
init-clss-add-learned-cls[*simp*]:
$\bigwedge C\ st.\ \textit{no-dup } (\textit{trail } st) \Longrightarrow \textit{init-clss } (\textit{add-learned-cls } C\ st) = \textit{init-clss } st$ **and**
init-clss-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{init-clss } (\textit{remove-cls } C\ st) = \textit{removeAll-mset } (\textit{mset-cls } C)\ (\textit{init-clss } st)$ **and**
init-clss-update-backtrack-lvl[*simp*]:
$\bigwedge st\ C.\ \textit{init-clss } (\textit{update-backtrack-lvl } C\ st) = \textit{init-clss } st$ **and**
init-clss-update-conflicting[*simp*]:
$\bigwedge C\ st.\ \textit{init-clss } (\textit{update-conflicting } C\ st) = \textit{init-clss } st$ **and**

learned-clss-cons-trail[*simp*]:
$\bigwedge M\ st.\ \textit{undefined-lit } (\textit{trail } st)\ (\textit{lit-of } M) \Longrightarrow$
$\textit{learned-clss } (\textit{cons-trail } M\ st) = \textit{learned-clss } st$ **and**
learned-clss-tl-trail[*simp*]:
$\bigwedge st.\ \textit{learned-clss } (\textit{tl-trail } st) = \textit{learned-clss } st$ **and**
learned-clss-add-init-cls[*simp*]:
$\bigwedge st\ C.\ \textit{no-dup } (\textit{trail } st) \Longrightarrow \textit{learned-clss } (\textit{add-init-cls } C\ st) = \textit{learned-clss } st$ **and**
learned-clss-add-learned-cls[*simp*]:
$\bigwedge C\ st.\ \textit{no-dup } (\textit{trail } st) \Longrightarrow$
$\textit{learned-clss } (\textit{add-learned-cls } C\ st) = \{\#\textit{mset-cls } C\#\} + \textit{learned-clss } st$ **and**
learned-clss-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{learned-clss } (\textit{remove-cls } C\ st) = \textit{removeAll-mset } (\textit{mset-cls } C)\ (\textit{learned-clss } st)$ **and**
learned-clss-update-backtrack-lvl[*simp*]:
$\bigwedge st\ C.\ \textit{learned-clss } (\textit{update-backtrack-lvl } C\ st) = \textit{learned-clss } st$ **and**
learned-clss-update-conflicting[*simp*]:
$\bigwedge C\ st.\ \textit{learned-clss } (\textit{update-conflicting } C\ st) = \textit{learned-clss } st$ **and**

backtrack-lvl-cons-trail[*simp*]:
$\bigwedge M\ st.\ \textit{undefined-lit } (\textit{trail } st)\ (\textit{lit-of } M) \Longrightarrow$
$\textit{backtrack-lvl } (\textit{cons-trail } M\ st) = \textit{backtrack-lvl } st$ **and**
backtrack-lvl-tl-trail[*simp*]:
$\bigwedge st.\ \textit{backtrack-lvl } (\textit{tl-trail } st) = \textit{backtrack-lvl } st$ **and**

backtrack-lvl-add-init-cls[*simp*]:
$\bigwedge$*st C*. *no-dup* (*trail st*) $\Longrightarrow$ *backtrack-lvl* (*add-init-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-add-learned-cls[*simp*]:
$\bigwedge$*C st*. *no-dup* (*trail st*) $\Longrightarrow$ *backtrack-lvl* (*add-learned-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-remove-cls[*simp*]:
$\bigwedge$*C st*. *backtrack-lvl* (*remove-cls C st*) = *backtrack-lvl st* **and**
backtrack-lvl-update-backtrack-lvl[*simp*]:
$\bigwedge$*st k*. *backtrack-lvl* (*update-backtrack-lvl k st*) = *k* **and**
backtrack-lvl-update-conflicting[*simp*]:
$\bigwedge$*C st*. *backtrack-lvl* (*update-conflicting C st*) = *backtrack-lvl st* **and**

conflicting-cons-trail[*simp*]:
$\bigwedge$*M st*. *undefined-lit* (*trail st*) (*lit-of M*) $\Longrightarrow$
conflicting (*cons-trail M st*) = *conflicting st* **and**
conflicting-tl-trail[*simp*]:
$\bigwedge$*st*. *conflicting* (*tl-trail st*) = *conflicting st* **and**
conflicting-add-init-cls[*simp*]:
$\bigwedge$*st C*. *no-dup* (*trail st*) $\Longrightarrow$ *conflicting* (*add-init-cls C st*) = *conflicting st* **and**
conflicting-add-learned-cls[*simp*]:
$\bigwedge$*C st*. *no-dup* (*trail st*) $\Longrightarrow$ *conflicting* (*add-learned-cls C st*) = *conflicting st*
and
conflicting-remove-cls[*simp*]:
$\bigwedge$*C st*. *conflicting* (*remove-cls C st*) = *conflicting st* **and**
conflicting-update-backtrack-lvl[*simp*]:
$\bigwedge$*st C*. *conflicting* (*update-backtrack-lvl C st*) = *conflicting st* **and**
conflicting-update-conflicting[*simp*]:
$\bigwedge$*C st*. *raw-conflicting* (*update-conflicting C st*) = *C* **and**

init-state-trail[*simp*]: $\bigwedge$*N*. *trail* (*init-state N*) = [] **and**
init-state-clss[*simp*]: $\bigwedge$*N*. (*init-clss* (*init-state N*)) = *mset-clss N* **and**
init-state-learned-clss[*simp*]: $\bigwedge$*N*. *learned-clss* (*init-state N*) = {#} **and**
init-state-backtrack-lvl[*simp*]: $\bigwedge$*N*. *backtrack-lvl* (*init-state N*) = *0* **and**
init-state-conflicting[*simp*]: $\bigwedge$*N*. *conflicting* (*init-state N*) = *None* **and**

trail-restart-state[*simp*]: *trail* (*restart-state S*) = [] **and**
init-clss-restart-state[*simp*]: *init-clss* (*restart-state S*) = *init-clss S* **and**
learned-clss-restart-state[*intro*]:
learned-clss (*restart-state S*) ⊆# *learned-clss S* **and**
backtrack-lvl-restart-state[*simp*]: *backtrack-lvl* (*restart-state S*) = *0* **and**
conflicting-restart-state[*simp*]: *conflicting* (*restart-state S*) = *None*
begin

lemma
shows
clauses-cons-trail[*simp*]:
undefined-lit (*trail S*) (*lit-of M*) $\Longrightarrow$ *clauses* (*cons-trail M S*) = *clauses S* **and**

clss-tl-trail[*simp*]: *clauses* (*tl-trail S*) = *clauses S* **and**
clauses-add-learned-cls-unfolded:
no-dup (*trail S*) $\Longrightarrow$ *clauses* (*add-learned-cls U S*) =
{#*mset-cls U*#} + *learned-clss S* + *init-clss S*
and
clauses-add-init-cls[*simp*]:
no-dup (*trail S*) $\Longrightarrow$
clauses (*add-init-cls N S*) = {#*mset-cls N*#} + *init-clss S* + *learned-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* (*mset-cls C*) (*clauses S*) **and**
clauses-add-learned-cls[*simp*]:
no-dup (*trail S*) $\Longrightarrow$ *clauses* (*add-learned-cls C S*) = {#*mset-cls C*#} + *clauses S* **and**
clauses-restart[*simp*]: *clauses* (*restart-state S*) ⊆# *clauses S* **and**
clauses-init-state[*simp*]: $\bigwedge N$. *clauses* (*init-state N*) = *mset-clss N*
prefer *9* **using** *raw-clauses-def learned-clss-restart-state* **apply** *fastforce*
by (*auto simp*: *ac-simps replicate-mset-plus raw-clauses-def intro*: *multiset-eqI*)

abbreviation *state* :: *'st* ⇒ (*'v*, *nat*, *'v clause*) *marked-lit list* × *'v clauses* × *'v clauses* × *nat* × *'v clause option* **where**
state S ≡ (*trail S*, *init-clss S*, *learned-clss S*, *backtrack-lvl S*, *conflicting S*)

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* $\Longrightarrow$ *T* ∼ *U* $\Longrightarrow$ *S* ∼ *U*
unfolding *state-eq-def* **by** *auto*

lemma
shows
state-eq-trail: *S* ∼ *T* $\Longrightarrow$ *trail S* = *trail T* **and**
state-eq-init-clss: *S* ∼ *T* $\Longrightarrow$ *init-clss S* = *init-clss T* **and**
state-eq-learned-clss: *S* ∼ *T* $\Longrightarrow$ *learned-clss S* = *learned-clss T* **and**
state-eq-backtrack-lvl: *S* ∼ *T* $\Longrightarrow$ *backtrack-lvl S* = *backtrack-lvl T* **and**
state-eq-conflicting: *S* ∼ *T* $\Longrightarrow$ *conflicting S* = *conflicting T* **and**
state-eq-clauses: *S* ∼ *T* $\Longrightarrow$ *clauses S* = *clauses T* **and**
state-eq-undefined-lit: *S* ∼ *T* $\Longrightarrow$ *undefined-lit* (*trail S*) *L* = *undefined-lit* (*trail T*) *L*
unfolding *state-eq-def raw-clauses-def* **by** *auto*

lemma *state-eq-raw-conflicting-None*:
S ∼ *T* $\Longrightarrow$ *conflicting T* = *None* $\Longrightarrow$ *raw-conflicting S* = *None*
unfolding *state-eq-def raw-clauses-def* **by** *auto*

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-raw-conflicting-None*

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*[*intro*]:
x ∈ *atms-of-mm* (*learned-clss* (*restart-state S*)) $\Longrightarrow$ *x* ∈ *atms-of-mm* (*learned-clss S*)
by (*meson atms-of-ms-mono learned-clss-restart-state set-mset-mono subsetCE*)

function *reduce-trail-to* :: *'a list* ⇒ *'st* ⇒ *'st* **where**
reduce-trail-to F S =
  (*if length* (*trail S*) = *length F* ∨ *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* = [] ⟹ *reduce-trail-to F S* = *S* **and**
    reduce-trail-to-eq-length[*simp*]: *length* (*trail S*) = *length F* ⟹ *reduce-trail-to F S* = *S*
  by (*auto simp*: *reduce-trail-to.simps*)

lemma *reduce-trail-to-length-ne*:
  length (*trail S*) ≠ *length F* ⟹ *trail S* ≠ [] ⟹
    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*, *nat*, *'v clause*) *marked-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 *raw-conflicting-reduce-trail-to*[*simp*]:
raw-conflicting (*reduce-trail-to F S*) = *None* ⟷ *raw-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'* @ *Marked K d* # *F* ⟹ (*trail* (*reduce-trail-to F S*)) = *F*
apply (*rule reduce-trail-to-skip-beginning*[*of - F'* @ *Marked K d* # []])
by (*cases F'*) (*auto simp add*:*tl-append reduce-trail-to-skip-beginning*)

lemma *reduce-trail-to-add-learned-cls*[*simp*]:
no-dup (*trail S*) ⟹
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-add-init-cls*[*simp*]:
no-dup (*trail S*) ⟹
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 *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 C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *in-get-all-marked-decomposition-marked-or-empty*:
assumes (*a*, *b*) ∈ *set* (*get-all-marked-decomposition M*)
shows *a* = [] ∨ (*is-marked* (*hd a*))
using *assms*
proof (*induct M arbitrary*: *a b*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons m M*)
show *?case*
proof (*cases m*)
case (*Marked l mark*)
then show *?thesis* **using** *Cons* **by** *auto*
next
case (*Propagated l mark*)
then show *?thesis* **using** *Cons* **by** (*cases get-all-marked-decomposition M*) *force+*
qed
qed

lemma *reduce-trail-to-length*:
length M = *length M′* ⟹ *reduce-trail-to M S* = *reduce-trail-to M′ S*
apply (*induction M S arbitrary*: *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-marked-decomposition-trail-update-trail*[*simp*]:
assumes *H*: (*L* # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*))
shows *trail* (*reduce-trail-to M1 S*) = *M1*
proof −
obtain *K mark* **where**
L: *L* = *Marked K mark*

using *H* **by** (*cases L*) (*auto dest*!: *in-get-all-marked-decomposition-marked-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 mark*])
(*auto simp*: *tr-S L*)
qed

lemma *raw-conflicting-cons-trail*[*simp*]:
assumes *undefined-lit* (*trail S*) (*lit-of L*)
shows
raw-conflicting (*cons-trail L S*) = *None* ⟷ *raw-conflicting S* = *None*
using *assms conflicting-cons-trail*[*of S L*] *map-option-is-None* **by** *fastforce+*

lemma *raw-conflicting-add-init-cls*[*simp*]:
no-dup (*trail S*) ⟹
raw-conflicting (*add-init-cls C S*) = *None* ⟷ *raw-conflicting S* = *None*
using *map-option-is-None conflicting-add-init-cls*[*of S C*] **by** *fastforce+*

lemma *raw-conflicting-add-learned-cls*[*simp*]:
no-dup (*trail S*) ⟹
raw-conflicting (*add-learned-cls C S*) = *None* ⟷ *raw-conflicting S* = *None*
using *map-option-is-None conflicting-add-learned-cls*[*of S C*] **by** *fastforce+*

lemma *raw-conflicting-update-backtracl-lvl*[*simp*]:
raw-conflicting (*update-backtrack-lvl k S*) = *None* ⟷ *raw-conflicting S* = *None*
using *map-option-is-None conflicting-update-backtrack-lvl*[*of k S*] **by** *fastforce+*

end — end of $state_W$ locale

19.2 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 clauses:
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:
mset-ccls union-ccls insert-ccls remove-clit

— conversion
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
— changing state:
cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state

restart-state
for
mset-cls:: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**

mset-clss:: *'clss* ⇒ *'v clauses* **and**
union-clss :: *'clss* ⇒ *'clss* ⇒ *'clss* **and**
in-clss :: *'cls* ⇒ *'clss* ⇒ *bool* **and**
insert-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**
remove-from-clss :: *'cls* ⇒ *'clss* ⇒ *'clss* **and**

mset-ccls:: *'ccls* ⇒ *'v clause* **and**
union-ccls :: *'ccls* ⇒ *'ccls* ⇒ *'ccls* **and**
insert-ccls :: *'v literal* ⇒ *'ccls* ⇒ *'ccls* **and**
remove-clit :: *'v literal* ⇒ *'ccls* ⇒ *'ccls* **and**

ccls-of-cls :: *'cls* ⇒ *'ccls* **and**
cls-of-ccls :: *'ccls* ⇒ *'cls* **and**

trail :: *'st* ⇒ (*'v*, *nat*, *'v clause*) *marked-lits* **and**
hd-raw-trail :: *'st* ⇒ (*'v*, *nat*, *'cls*) *marked-lit* **and**
raw-init-clss :: *'st* ⇒ *'clss* **and**
raw-learned-clss :: *'st* ⇒ *'clss* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conflicting :: *'st* ⇒ *'ccls option* **and**

cons-trail :: (*'v*, *nat*, *'cls*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-init-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
add-learned-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'ccls option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'clss* ⇒ *'st* **and**
restart-state :: *'st* ⇒ *'st*
begin

inductive *propagate* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate-rule: *conflicting S* = *None* ⟹
E !∈! *raw-clauses S* ⟹
L ∈# *mset-cls E* ⟹
trail S ⊨*as CNot* (*mset-cls* (*remove-lit L E*)) ⟹
undefined-lit (*trail S*) *L* ⟹
T ∼ *cons-trail* (*Propagated L E*) *S* ⟹
propagate S T

inductive-cases *propagateE*: *propagate S T*

inductive *conflict* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
conflict-rule:
conflicting S = *None* ⟹
D !∈! *raw-clauses S* ⟹
trail S ⊨*as CNot* (*mset-cls D*) ⟹

$T \sim$ *update-conflicting* (*Some* (*ccls-of-cls* D)) $S \Longrightarrow$
conflict S T

inductive-cases *conflictE*: *conflict* S T

inductive *backtrack* :: $'st \Rightarrow 'st \Rightarrow$ *bool* **for** S :: $'st$ **where**
backtrack-rule:
raw-conflicting S = *Some* $D \Longrightarrow$
$L \in\#$ *mset-ccls* $D \Longrightarrow$
(*Marked* K (i+1) # $M1$, $M2$) $\in$ *set* (*get-all-marked-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) (*mset-ccls* D) $\Longrightarrow$
get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* L D)) $\equiv i \Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* L (*cls-of-ccls* D))
(*reduce-trail-to* $M1$
(*add-learned-cls* (*cls-of-ccls* D)
(*update-backtrack-lvl* i
(*update-conflicting* *None* S)))) $\Longrightarrow$
backtrack S T

inductive-cases *backtrackE*: *backtrack* S T
thm *backtrackE*

inductive *decide* :: $'st \Rightarrow 'st \Rightarrow$ *bool* **for** S :: $'st$ **where**
decide-rule:
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* (*Marked* L (*backtrack-lvl* S +1)) (*incr-lvl* S) $\Longrightarrow$
decide S T

inductive-cases *decideE*: *decide* S T

inductive *skip* :: $'st \Rightarrow 'st \Rightarrow$ *bool* **for** S :: $'st$ **where**
skip-rule:
trail S = *Propagated* L C' # $M \Longrightarrow$
raw-conflicting S = *Some* $E \Longrightarrow$
$-L \notin\#$ *mset-ccls* $E \Longrightarrow$
mset-ccls $E \neq \{\#\} \Longrightarrow$
$T \sim$ *tl-trail* $S \Longrightarrow$
skip S T

inductive-cases *skipE*: *skip* S T

get-maximum-level (*Propagated* L (C + {#L#}) # M) $D = k \vee k = 0$ (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* :: $'st \Rightarrow 'st \Rightarrow$ *bool* **for** S :: $'st$ **where**
resolve-rule: *trail* $S \neq [] \Longrightarrow$
hd-raw-trail S = *Propagated* L $E \Longrightarrow$
$L \in\#$ *mset-cls* $E \Longrightarrow$
raw-conflicting S = *Some* $D' \Longrightarrow$
$-L \in\#$ *mset-ccls* $D' \Longrightarrow$
get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* ($-L$) D')) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting* (*Some* (*union-ccls* (*remove-clit* ($-L$) D') (*ccls-of-cls* (*remove-lit* L E))))

$(tl\text{-}trail\ S) \Longrightarrow$
resolve S T

inductive-cases *resolveE*: *resolve S T*

inductive *restart* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
restart: $state\ S = (M,\ N,\ U,\ k,\ None) \Longrightarrow \neg M \models asm\ clauses\ S$
$\Longrightarrow T \sim restart\text{-}state\ S$
$\Longrightarrow restart\ S\ T$

inductive-cases *restartE*: *restart S T*

We add the condition $C \notin\# \ init\text{-}clss\ S$, to maintain consistency even without the strategy.

inductive *forget*:: *'st* ⇒ *'st* ⇒ *bool* **where**
forget-rule:
$conflicting\ S = None \Longrightarrow$
$C\ !\in!\ raw\text{-}learned\text{-}clss\ S \Longrightarrow$
$\neg(trail\ S) \models asm\ clauses\ S \Longrightarrow$
$mset\text{-}cls\ C \notin set\ (get\text{-}all\text{-}mark\text{-}of\text{-}propagated\ (trail\ S)) \Longrightarrow$
$mset\text{-}cls\ C \notin\# \ init\text{-}clss\ S \Longrightarrow$
$T \sim remove\text{-}cls\ C\ S \Longrightarrow$
forget S T

inductive-cases *forgetE*: *forget S T*

inductive $cdcl_W\text{-}rf$ :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
restart: $restart\ S\ T \Longrightarrow cdcl_W\text{-}rf\ S\ T$ |
forget: $forget\ S\ T \Longrightarrow cdcl_W\text{-}rf\ S\ T$

inductive $cdcl_W\text{-}bj$ :: *'st* ⇒ *'st* ⇒ *bool* **where**
skip: $skip\ S\ S' \Longrightarrow cdcl_W\text{-}bj\ S\ S'$ |
resolve: $resolve\ S\ S' \Longrightarrow cdcl_W\text{-}bj\ S\ S'$ |
backtrack: $backtrack\ S\ S' \Longrightarrow cdcl_W\text{-}bj\ S\ S'$

inductive-cases $cdcl_W\text{-}bjE$: $cdcl_W\text{-}bj\ S\ T$

inductive $cdcl_W\text{-}o$:: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
decide: $decide\ S\ S' \Longrightarrow cdcl_W\text{-}o\ S\ S'$ |
bj: $cdcl_W\text{-}bj\ S\ S' \Longrightarrow cdcl_W\text{-}o\ S\ S'$

inductive $cdcl_W$ :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate: $propagate\ S\ S' \Longrightarrow cdcl_W\ S\ S'$ |
conflict: $conflict\ S\ S' \Longrightarrow cdcl_W\ S\ S'$ |
other: $cdcl_W\text{-}o\ S\ S' \Longrightarrow cdcl_W\ S\ S'$|
rf: $cdcl_W\text{-}rf\ S\ S' \Longrightarrow cdcl_W\ S\ S'$

lemma $rtranclp\text{-}propagate\text{-}is\text{-}rtranclp\text{-}cdcl_W$:
$propagate^{**}\ S\ S' \Longrightarrow cdcl_W^{**}\ S\ S'$
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
apply (*frule propagate*)
using *rtranclp-trans*[*of* $cdcl_W$] **by** *blast*

lemma $cdcl_W\text{-}all\text{-}rules\text{-}induct$[*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 !∈! *raw-clauses* S $\Longrightarrow$
L ∈# *mset-cls* C $\Longrightarrow$
trail S ⊨*as* *CNot* (*remove1-mset* L (*mset-cls* C)) $\Longrightarrow$
undefined-lit (*trail* S) L $\Longrightarrow$
T ∼ *cons-trail* (*Propagated* L C) S $\Longrightarrow$
P S T **and**
conflictH: $\bigwedge D\ T$. *conflicting* S = *None* $\Longrightarrow$
D !∈! *raw-clauses* S $\Longrightarrow$
trail S ⊨*as* *CNot* (*mset-cls* D) $\Longrightarrow$
T ∼ *update-conflicting* (*Some* (*ccls-of-cls* D)) S $\Longrightarrow$
P S T **and**
forgetH: $\bigwedge C\ U\ T$. *conflicting* S = *None* $\Longrightarrow$
C !∈! *raw-learned-clss* S $\Longrightarrow$
¬(*trail* S) ⊨*asm* *clauses* S $\Longrightarrow$

$mset\text{-}cls\ C \notin set\ (get\text{-}all\text{-}mark\text{-}of\text{-}propagated\ (trail\ S)) \Longrightarrow$
$mset\text{-}cls\ C \notin\#\ init\text{-}clss\ S \Longrightarrow$
$T \sim remove\text{-}cls\ C\ S \Longrightarrow$
$P\ S\ T$ **and**
restartH: $\bigwedge T.\ \neg trail\ S \models asm\ clauses\ S \Longrightarrow$
$conflicting\ S = None \Longrightarrow$
$T \sim restart\text{-}state\ S \Longrightarrow$
$P\ S\ T$ **and**
decideH: $\bigwedge L\ T.\ conflicting\ S = None \Longrightarrow$
$undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow$
$atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (init\text{-}clss\ S) \Longrightarrow$
$T \sim cons\text{-}trail\ (Marked\ L\ (backtrack\text{-}lvl\ S\ +1))\ (incr\text{-}lvl\ S) \Longrightarrow$
$P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
$trail\ S = Propagated\ L\ C'\ \#\ M \Longrightarrow$
$raw\text{-}conflicting\ S = Some\ E \Longrightarrow$
$-L \notin\#\ mset\text{-}ccls\ E \Longrightarrow\ \ mset\text{-}ccls\ E \neq \{\#\} \Longrightarrow$
$T \sim tl\text{-}trail\ S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
$trail\ S = Propagated\ L\ (mset\text{-}cls\ E)\ \#\ M \Longrightarrow$
$L \in\#\ mset\text{-}cls\ E \Longrightarrow$
$hd\text{-}raw\text{-}trail\ S\ = Propagated\ L\ E \Longrightarrow$
$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$-L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ (remove\text{-}clit\ (-L)\ D)) = backtrack\text{-}lvl\ S \Longrightarrow$
$T \sim update\text{-}conflicting$
$(Some\ (union\text{-}ccls\ (remove\text{-}clit\ (-L)\ D)\ (ccls\text{-}of\text{-}cls\ (remove\text{-}lit\ L\ E))))\ (tl\text{-}trail\ S) \Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge L\ D\ K\ i\ M1\ M2\ T.$
$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$(Marked\ K\ (i{+}1)\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S)) \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = backtrack\text{-}lvl\ S \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ D) \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (remove1\text{-}mset\ L\ (mset\text{-}ccls\ D)) \equiv i \Longrightarrow$
$T \sim cons\text{-}trail\ (Propagated\ L\ (cls\text{-}of\text{-}ccls\ D))$
$(reduce\text{-}trail\text{-}to\ M1$
$(add\text{-}learned\text{-}cls\ (cls\text{-}of\text{-}ccls\ D)$
$(update\text{-}backtrack\text{-}lvl\ i$
$(update\text{-}conflicting\ None\ S)))) \Longrightarrow$
$P\ S\ T$
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*
using *hd-raw-trail*[*of S*] **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* (*Marked L* (*backtrack-lvl S* +*1*)) (*incr-lvl S*)
$\Longrightarrow$ *P S T* **and**
skipH: $\bigwedge$*L C' M E T*.
trail S = *Propagated L C'* # *M* $\Longrightarrow$
raw-conflicting S = *Some E* $\Longrightarrow$
$-$*L* $\notin$# *mset-ccls E* $\Longrightarrow$ *mset-ccls E* $\neq$ {#} $\Longrightarrow$
T $\sim$ *tl-trail S* $\Longrightarrow$
P S T **and**
resolveH: $\bigwedge$*L E M D T*.
trail S = *Propagated L* (*mset-cls E*) # *M* $\Longrightarrow$
L $\in$# *mset-cls E* $\Longrightarrow$
hd-raw-trail S = *Propagated L E* $\Longrightarrow$
raw-conflicting S = *Some D* $\Longrightarrow$
$-$*L* $\in$# *mset-ccls D* $\Longrightarrow$
get-maximum-level (*trail S*) (*mset-ccls* (*remove-clit* ($-$*L*) *D*)) = *backtrack-lvl S* $\Longrightarrow$
T $\sim$ *update-conflicting*
(*Some* (*union-ccls* (*remove-clit* ($-$*L*) *D*) (*ccls-of-cls* (*remove-lit L E*)))) (*tl-trail S*) $\Longrightarrow$
P S T **and**
backtrackH: $\bigwedge$*L D K i M1 M2 T*.
raw-conflicting S = *Some D* $\Longrightarrow$
L $\in$# *mset-ccls D* $\Longrightarrow$
(*Marked K* (*i*+*1*) # *M1*, *M2*) $\in$ *set* (*get-all-marked-decomposition* (*trail S*)) $\Longrightarrow$
get-level (*trail S*) *L* = *backtrack-lvl S* $\Longrightarrow$
get-level (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls D*) $\Longrightarrow$
get-maximum-level (*trail S*) (*remove1-mset L* (*mset-ccls D*)) $\equiv$ *i* $\Longrightarrow$
T $\sim$ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls 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*)
using *hd-raw-trail*[*of S*] **apply** (*cases trail S*; *auto elim*!: *resolveE intro*!: *resolveH*)
apply (*frule backtrackH*; *simp-all del*: *state-simp add*: *state-eq-def*)
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*)

19.3 Invariants

19.3.1 Properties of the trail

We here establish that: * the marks are exactly 1..k where k is the level * 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: (*Marked K* (*i* + *1*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*)) **and**
no-dup: *no-dup* (*trail S*) **and**
bt-l: *backtrack-lvl S* = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
order: *get-all-levels-of-marked* (*trail S*)
= *rev* [*1*..<(*1*+*length* (*get-all-levels-of-marked* (*trail S*)))]
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* @ *Marked K* (*i* + *1*) # *M1*
using *M1* **by** *blast*

have *atm-of L* $\notin$ *atm-of ' lits-of-l c*
using *L-in-M1 no-dup* **unfolding** *Mc lits-of-def* **by** *force*
have *g-M-eq-g-M1*: *get-level ?M L* = *get-level M1 L*
using *L-in-M1* **unfolding** *Mc* **by** *auto*
have *g*: *get-all-levels-of-marked M1* = *rev* [*1*..<*Suc i*]
using *order* **unfolding** *Mc* **by** (*auto simp del*: *upt-simps simp*: *rev-swap*[*symmetric*]
dest: *append-cons-eq-upt-length-i*)
then have *Max* (*set* (*0* # *get-all-levels-of-marked* (*rev M1*))) < *Suc i* **by** *auto*
then have *get-level M1 L* < *Suc i*
using *get-rev-level-less-max-get-all-levels-of-marked*[*of rev M1 0 L*] **by** *linarith*
moreover have *Suc i* $\leq$ *backtrack-lvl S* **using** *bt-l* **by** (*simp add*: *Mc g*)
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**
backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
get-all-levels-of-marked (*trail S*) = *rev* [*1*..<*1*+*length* (*get-all-levels-of-marked* (*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** *T* = *this*(*7*) **and**
n-d = *this*(*8*)
obtain *c* **where** *Mc*: *trail S* = *c* @ *M2* @ *Marked K* (*i* + *1*) # *M1*
using *decomp* **by** *auto*
have *no-dup* (*M2* @ *Marked K* (*i* + *1*) # *M1*)
using *Mc n-d* **by** *fastforce*
moreover have *atm-of L* $\notin$ (λ*l. atm-of* (*lit-of l*)) ' *set M1*
using *backtrack-lit-skiped*[*of S L K i M1 M2*] *L decomp backtrack.prems*
by (*fastforce simp*: *lits-of-def*)
moreover then have *undefined-lit M1 L*
by (*simp add*: *defined-lit-map*)
ultimately show *?case* **using** *decomp T n-d* **by** *simp*
qed (*auto simp*: *defined-lit-map*)

lemma $cdcl_W$*-consistent-inv-2*:
assumes
$cdcl_W$ *S S'* **and**
no-dup (*trail S*) **and**
backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
get-all-levels-of-marked (*trail S*) = *rev* [*1*..<*1*+*length* (*get-all-levels-of-marked* (*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 = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
get-all-levels-of-marked (*trail S*) =
rev ([*1*..<(*1*+*length* (*get-all-levels-of-marked* (*trail S*)))]) **and**
n-d[*simp*]: *no-dup* (*trail S*)
shows *backtrack-lvl S'* = *length* (*get-all-levels-of-marked* (*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** *T* = *this*(*7*) **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* @ *Marked K* (*i* + *1*) # *M1* **using** *decomp* **by** *auto*
have *rev* (*get-all-levels-of-marked* (*trail S*))
= [*1*..<*1*+ (*length* (*get-all-levels-of-marked* (*trail S*)))]
using *level* **by** (*auto simp*: *rev-swap*[*symmetric*])
moreover have *atm-of L* $\notin$ (λ*l*. *atm-of* (*lit-of l*)) ‘ *set M1*
using *backtrack-lit-skiped*[*of S L K i M1 M2*] *backtrack*(*4*,*8*,*9*) *decomp*
by (*fastforce simp add*: *lits-of-def*)
moreover then have *undefined-lit M1 L*
by (*simp add*: *defined-lit-map*)
moreover then have *no-dup* (*trail T*)
using *T decomp n-d* **by** (*auto simp*: *defined-lit-map M*)
ultimately show *?case*
using *T n-d* **unfolding** *M* **by** (*auto dest*!: *append-cons-eq-upt-length simp del*: *upt-simps*)
qed *auto*

lemma *cdcl*$_W$*-rf-bt*:
assumes
cdcl$_W$*-rf S S'* **and**
backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
get-all-levels-of-marked (*trail S*) = *rev* [*1*..<(*1*+*length* (*get-all-levels-of-marked* (*trail S*)))]
shows *backtrack-lvl S'* = *length* (*get-all-levels-of-marked* (*trail S'*))
using *assms* **by** (*induct rule*: *cdcl*$_W$*-rf.induct*) (*auto elim*: *restartE forgetE*)

lemma *cdcl*$_W$*-bt*:
assumes
cdcl$_W$ *S S'* **and**
backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
get-all-levels-of-marked (*trail S*)
= *rev* ([*1*..<(*1*+*length* (*get-all-levels-of-marked* (*trail S*)))]) **and**
no-dup (*trail S*)
shows *backtrack-lvl S'* = *length* (*get-all-levels-of-marked* (*trail S'*))
using *assms* **by** (*induct rule*: *cdcl*$_W$*.induct*) (*auto simp add*: *cdcl*$_W$*-o-bt cdcl*$_W$*-rf-bt*
elim: *conflictE propagateE*)

lemma *cdcl*$_W$*-bt-level'*:
assumes
cdcl$_W$ *S S'* **and**
backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**
get-all-levels-of-marked (*trail S*)
= *rev* ([*1*..<(*1*+*length* (*get-all-levels-of-marked* (*trail S*)))]) **and**
n-d: *no-dup* (*trail S*)
shows *get-all-levels-of-marked* (*trail S'*)
= *rev* [*1*..<*1*+*length* (*get-all-levels-of-marked* (*trail S'*))]
using *assms*
proof (*induct rule*: *cdcl*$_W$*-all-induct*)
case (*decide L T*) **note** *undef* = *this*(*2*) **and** *T* = *this*(*4*)
let *?k* = *backtrack-lvl S*
let *?M* = *trail S*
let *?M'* = *Marked L* (*?k* + *1*) # *trail S*
have *H*: *get-all-levels-of-marked ?M* = *rev* [*Suc 0*..<*1*+*length* (*get-all-levels-of-marked ?M*)]
using *decide.prems* **by** *simp*

have *k*: *?k = length (get-all-levels-of-marked ?M)*
  using *decide.prems* by *auto*
have *get-all-levels-of-marked ?M' = Suc ?k # get-all-levels-of-marked ?M* by *simp*
then have *get-all-levels-of-marked ?M' = Suc ?k #*
  rev [Suc 0..<1+length (get-all-levels-of-marked ?M)]
  using *H* by *auto*
moreover have *... = rev [Suc 0..< Suc (1+length (get-all-levels-of-marked ?M))]*
  unfolding *k* by *simp*
finally show *?case* using *T undef* by *(auto simp add: defined-lit-map)*
next
case *(backtrack L D K i M1 M2 T)* note *decomp = this(3)* and *confli = this(1)* and *T = this(7)*
and
  all-marked = this(9) and *bt-lvl = this(8)*
have *atm-of L ∉ atm-of ` lits-of-l M1*
  using *backtrack-lit-skiped[of S L K i M1 M2] backtrack(4,8−10) decomp*
  by *(fastforce simp add: lits-of-def)*
moreover then have *undefined-lit M1 L*
  by *(auto simp: defined-lit-map lits-of-def)*
then have *[simp]: trail T = Propagated L (mset-ccls D) # M1*
  using *T decomp n-d* by *auto*
obtain *c* where *M: trail S = c @ M2 @ Marked K (i + 1) # M1* using *decomp* by *auto*
have *get-all-levels-of-marked (rev (trail S))*
  = [Suc 0..<2+length (get-all-levels-of-marked c) + (length (get-all-levels-of-marked M2)
  + length (get-all-levels-of-marked M1))]
  using *all-marked bt-lvl* unfolding *M* by *(auto simp: rev-swap[symmetric] simp del: upt-simps)*
then show *?case*
  using *T* by *(auto simp: rev-swap M simp del: upt-simps dest!: append-cons-eq-upt(1))*
qed *auto*

We write *1 + length (get-all-levels-of-marked (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 = length (get-all-levels-of-marked (trail S))
  ∧ get-all-levels-of-marked (trail S)
  = rev [1..<1+length (get-all-levels-of-marked (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* $cdcl_W$*-bt-level'*
  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*
proof −
  have *get-all-levels-of-marked* (*trail S*) = *rev* [*1..<1* + *backtrack-lvl S*]
    using *inv* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
  then show *?thesis*
    using *get-rev-level-less-max-get-all-levels-of-marked*[*of rev* (*trail S*) *0 L*]
    by (*auto simp*: *Max-n-upt*)
qed

lemma *backtrack-ex-decomp*:
  assumes
    M-l: $cdcl_W$*-M-level-inv S* **and**
    i-S: *i* < *backtrack-lvl S*
  shows ∃ *K M1 M2*. (*Marked K* (*i* + *1*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*))
proof −
  let *?M* = *trail S*
  have
    g: *get-all-levels-of-marked* (*trail S*) = *rev* [*Suc 0..<Suc* (*backtrack-lvl S*)]
    using *M-l* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *simp-all*
  then have *i+1* ∈ *set* (*get-all-levels-of-marked* (*trail S*))
    using *i-S* **by** *auto*

  then obtain *c K c'* **where** *tr-S*: *trail S* = *c* @ *Marked K* (*i* + *1*) # *c'*
    using *in-get-all-levels-of-marked-iff-decomp*[*of i+1 trail S*] **by** *auto*

  obtain *M1 M2* **where** (*Marked K* (*i* + *1*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*))
    using *Marked-cons-in-get-all-marked-decomposition-append-Marked-cons* **unfolding** *tr-S* **by** *fast*
  then show *?thesis* **by** *blast*
qed

19.3.2 Better-Suited Induction Principle

We generalise the induction principle defined previously: the induction case for *backtrack* now includes the assumption that *undefined-lit M1 L*. This helps the simplifier and thus the automation.

lemma *backtrack-induction-lev*[*consumes 1*, *case-names M-devel-inv backtrack*]:
  assumes
    bt: *backtrack S T* **and**
    inv: $cdcl_W$*-M-level-inv S* **and**
    backtrackH: $\bigwedge$*K i M1 M2 L D T*.
      raw-conflicting S = *Some D* $\Longrightarrow$
      L ∈# *mset-ccls D* $\Longrightarrow$
      (*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*)) $\Longrightarrow$
      get-level (*trail S*) *L* = *backtrack-lvl S* $\Longrightarrow$
      get-level (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls D*) $\Longrightarrow$
      get-maximum-level (*trail S*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* $\Longrightarrow$
      undefined-lit M1 L $\Longrightarrow$
      T ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
              (*reduce-trail-to M1*
                (*add-learned-cls* (*cls-of-ccls D*)
                  (*update-backtrack-lvl i*
                    (*update-conflicting None S*)))) $\Longrightarrow$
      P S T
  shows *P S T*
proof −
  obtain *K i M1 M2 L D* **where**
    decomp: (*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*)) **and**
    L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
    confl: *raw-conflicting S* = *Some D* **and**
    LD: *L* ∈# *mset-ccls D* **and**
    lev-L: *get-level* (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls D*) **and**
    lev-D: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* **and**
    T: *T* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
              (*reduce-trail-to M1*
                (*add-learned-cls* (*cls-of-ccls D*)
                  (*update-backtrack-lvl i*
                    (*update-conflicting None S*))))
    using *bt* **by** (*elim backtrackE*) *metis*

  have *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*
    using *backtrack-lit-skiped*[*of S L K i M1 M2*] *L decomp bt confl lev-L lev-D inv*
    unfolding $cdcl_W$*-M-level-inv-def* **by** *force*
  then have *undefined-lit M1 L*
    by (*auto simp*: *defined-lit-map lits-of-def*)
  then show *?thesis*
    using *backtrackH*[*OF confl LD decomp L lev-L lev-D - T*] **by** *simp*
qed

lemmas *backtrack-induction-lev2* = *backtrack-induction-lev*[*consumes 2*, *case-names backtrack*]

lemma $cdcl_W$*-all-induct-lev-full*:
  fixes *S* :: *'st*
  assumes
    $cdcl_W$: $cdcl_W$ *S S'* **and**
    inv[*simp*]: $cdcl_W$*-M-level-inv S* **and**
    propagateH: $\bigwedge$*C L T*. *conflicting S* = *None* $\Longrightarrow$
      C !∈! *raw-clauses S* $\Longrightarrow$
      L ∈# *mset-cls C* $\Longrightarrow$
      trail S ⊨*as CNot* (*remove1-mset L* (*mset-cls C*)) $\Longrightarrow$

$\quad$ *undefined-lit* (*trail* S) $L \Longrightarrow$
$\quad T \sim$ *cons-trail* (*Propagated* L C) $S \Longrightarrow$
$\quad P\ S\ T$ **and**
conflictH: $\bigwedge D\ T.$ *conflicting* S = *None* $\Longrightarrow$
$\quad D$!∈! *raw-clauses* $S \Longrightarrow$
$\quad$ *trail* $S \models$*as* *CNot* (*mset-cls* D) $\Longrightarrow$
$\quad T \sim$ *update-conflicting* (*Some* (*ccls-of-cls* D)) $S \Longrightarrow$
$\quad P\ S\ T$ **and**
forgetH: $\bigwedge C\ T.$ *conflicting* S = *None* $\Longrightarrow$
$\quad C$!∈! *raw-learned-clss* $S \Longrightarrow$
$\quad \neg$(*trail* S) $\models$*asm* *clauses* $S \Longrightarrow$
$\quad$ *mset-cls* $C \notin$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\Longrightarrow$
$\quad$ *mset-cls* $C \notin\#$ *init-clss* $S \Longrightarrow$
$\quad T \sim$ *remove-cls* C $S \Longrightarrow$
$\quad P\ S\ T$ **and**
restartH: $\bigwedge T.$ $\neg$*trail* $S \models$*asm* *clauses* $S \Longrightarrow$
$\quad$ *conflicting* S = *None* $\Longrightarrow$
$\quad T \sim$ *restart-state* $S \Longrightarrow$
$\quad P\ S\ T$ **and**
decideH: $\bigwedge L\ T.$ *conflicting* S = *None* $\Longrightarrow$
$\quad$ *undefined-lit* (*trail* S) $L \Longrightarrow$
$\quad$ *atm-of* $L \in$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
$\quad T \sim$ *cons-trail* (*Marked* L (*backtrack-lvl* S +*1*)) (*incr-lvl* S) $\Longrightarrow$
$\quad P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
$\quad$ *trail* S = *Propagated* L C' # $M \Longrightarrow$
$\quad$ *raw-conflicting* S = *Some* $E \Longrightarrow$
$\quad -L \notin\#$ *mset-ccls* $E \Longrightarrow$ *mset-ccls* $E \neq \{\#\} \Longrightarrow$
$\quad T \sim$ *tl-trail* $S \Longrightarrow$
$\quad P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
$\quad$ *trail* S = *Propagated* L (*mset-cls* E) # $M \Longrightarrow$
$\quad L \in\#$ *mset-cls* $E \Longrightarrow$
$\quad$ *hd-raw-trail* S = *Propagated* L $E \Longrightarrow$
$\quad$ *raw-conflicting* S = *Some* $D \Longrightarrow$
$\quad -L \in\#$ *mset-ccls* $D \Longrightarrow$
$\quad$ *get-maximum-level* (*trail* S) (*mset-ccls* (*remove-clit* ($-L$) D)) = *backtrack-lvl* $S \Longrightarrow$
$\quad T \sim$ *update-conflicting*
$\qquad$ (*Some* (*union-ccls* (*remove-clit* ($-L$) D) (*ccls-of-cls* (*remove-lit* L E)))) (*tl-trail* S) $\Longrightarrow$
$\quad P\ S\ T$ **and**
backtrackH: $\bigwedge K\ i\ M1\ M2\ L\ D\ T.$
$\quad$ *raw-conflicting* S = *Some* $D \Longrightarrow$
$\quad L \in\#$ *mset-ccls* $D \Longrightarrow$
$\quad$ (*Marked* K (*Suc* i) # *M1*, *M2*) $\in$ *set* (*get-all-marked-decomposition* (*trail* S)) $\Longrightarrow$
$\quad$ *get-level* (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
$\quad$ *get-level* (*trail* S) L = *get-maximum-level* (*trail* S) (*mset-ccls* D) $\Longrightarrow$
$\quad$ *get-maximum-level* (*trail* S) (*remove1-mset* L (*mset-ccls* D)) $\equiv i \Longrightarrow$
$\quad$ *undefined-lit* *M1* $L \Longrightarrow$
$\quad T \sim$ *cons-trail* (*Propagated* L (*cls-of-ccls* D))
$\qquad\qquad$ (*reduce-trail-to* *M1*
$\qquad\qquad\quad$ (*add-learned-cls* (*cls-of-ccls* D)
$\qquad\qquad\qquad$ (*update-backtrack-lvl* i
$\qquad\qquad\qquad\quad$ (*update-conflicting* *None* S)))) $\Longrightarrow$
$\quad P\ S\ T$
shows $P\ S\ S'$

using $cdcl_W$
proof (*induct* 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*
apply (*induction rule*: *backtrack-induction-lev*)
apply (*rule inv*)
by (*rule backtrackH*;
fastforce simp del: *state-simp simp add*: *state-eq-def dest*!: *HOL.meta-eq-to-obj-eq*)
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*
using *hd-raw-trail*[*of S*] **by** (*cases trail S*) (*auto elim*!: *resolveE intro*!: *resolveH*)
qed

lemmas $cdcl_W$*-all-induct-lev2* = $cdcl_W$*-all-induct-lev-full*[*consumes 2*, *case-names propagate conflict forget restart decide skip resolve backtrack*]

lemmas $cdcl_W$*-all-induct-lev* = $cdcl_W$*-all-induct-lev-full*[*consumes 1*, *case-names lev-inv propagate conflict forget restart decide skip resolve backtrack*]

thm $cdcl_W$*-o-induct*
lemma $cdcl_W$*-o-induct-lev*[*consumes 1*, *case-names M-lev decide skip resolve backtrack*]:
fixes S :: *'st*
assumes
$cdcl_W$: $cdcl_W$*-o S T* **and**
inv[*simp*]: $cdcl_W$*-M-level-inv S* **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* (*Marked L* (*backtrack-lvl S +1*)) (*incr-lvl S*) $\Longrightarrow$
P S T **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
trail S = Propagated L C' # M $\Longrightarrow$

$raw\text{-}conflicting\ S = Some\ E \Longrightarrow$
$-L \notin\# \ mset\text{-}ccls\ E \Longrightarrow \ mset\text{-}ccls\ E \neq \{\#\} \Longrightarrow$
$T \sim tl\text{-}trail\ S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
$trail\ S = Propagated\ L\ (mset\text{-}cls\ E)\ \#\ M \Longrightarrow$
$L \in\#\ mset\text{-}cls\ E \Longrightarrow$
$hd\text{-}raw\text{-}trail\ S\ = Propagated\ L\ E \Longrightarrow$
$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$-L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ (remove\text{-}clit\ (-L)\ D)) = backtrack\text{-}lvl\ S \Longrightarrow$
$T \sim update\text{-}conflicting$
$(Some\ (union\text{-}ccls\ (remove\text{-}clit\ (-L)\ D)\ (ccls\text{-}of\text{-}cls\ (remove\text{-}lit\ L\ E))))\ (tl\text{-}trail\ S) \Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge K\ i\ M1\ M2\ L\ D\ T.$
$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$(Marked\ K\ (Suc\ i)\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S)) \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = backtrack\text{-}lvl\ S \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ D) \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (remove1\text{-}mset\ L\ (mset\text{-}ccls\ D)) \equiv i \Longrightarrow$
$undefined\text{-}lit\ M1\ L \Longrightarrow$
$T \sim cons\text{-}trail\ (Propagated\ L\ (cls\text{-}of\text{-}ccls\ D))$
$(reduce\text{-}trail\text{-}to\ M1$
$(add\text{-}learned\text{-}cls\ (cls\text{-}of\text{-}ccls\ D)$
$(update\text{-}backtrack\text{-}lvl\ i$
$(update\text{-}conflicting\ None\ S)))) \Longrightarrow$
$P\ S\ T$
shows $P\ S\ T$
using $cdcl_W$
proof (*induct S T rule*: $cdcl_W$*-o-all-rules-induct*)
case (*decide T*)
then show *?case*
by (*auto elim*!: *decideE intro*!: *decideH*)
next
case (*backtrack S'*)
then show *?case*
apply (*induction rule*: *backtrack-induction-lev*)
apply (*rule inv*)
by (*rule backtrackH*;
fastforce simp del: *state-simp simp add*: *state-eq-def dest*!: *HOL.meta-eq-to-obj-eq*)
next
case (*skip S'*)
then show *?case* **by** (*auto elim*!: *skipE intro*!: *skipH*)
next
case (*resolve S'*)
then show *?case*
using *hd-raw-trail*[*of S*] **by** (*cases trail S*) (*auto elim*!: *resolveE intro*!: *resolveH*)
qed

lemmas $cdcl_W$*-o-induct-lev2* = $cdcl_W$*-o-induct-lev*[*consumes 2*, *case-names decide skip resolve backtrack*]

19.3.3 Compatibility with $op \sim$

lemma *propagate-state-eq-compatible*:

assumes
  propa: *propagate S T* **and**
  SS': $S \sim S'$ **and**
  TT': $T \sim T'$
shows *propagate S' T'*
proof –
  obtain *C L* **where**
    conf: *conflicting S = None* **and**
    C: *C* !∈! *raw-clauses S* **and**
    L: *L* ∈# *mset-cls C* **and**
    tr: *trail S* ⊨*as CNot* (*remove1-mset L* (*mset-cls C*)) **and**
    undef: *undefined-lit* (*trail S*) *L* **and**
    T: $T \sim$ *cons-trail* (*Propagated L C*) *S*
  using *propa* **by** (*elim propagateE*) *auto*

  obtain *C'* **where**
    CC': *mset-cls C' = mset-cls C* **and**
    C': *C'* !∈! *raw-clauses S'*
    using *SS' C*
    in-mset-clss-exists-preimage[*of mset-cls C raw-learned-clss S'*]
    in-mset-clss-exists-preimage[*of mset-cls C raw-init-clss S'*]
    apply –
    apply (*frule in-clss-mset-clss*)
    by (*auto simp*: *state-eq-def raw-clauses-def simp del*: *state-simp dest*: *in-clss-mset-clss*)

  show *?thesis*
    apply (*rule propagate-rule*[*of - C'*])
    using *state-eq-sym*[*of S S'*] *SS' conf C' CC' 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': $T \sim T'$ **and**
    SS': $S \sim S'$
  shows *conflict S' T'*
proof –
  obtain *D* **where**
    conf: *conflicting S = None* **and**
    D: *D* !∈! *raw-clauses S* **and**
    tr: *trail S* ⊨*as CNot* (*mset-cls D*) **and**
    T: $T \sim$ *update-conflicting* (*Some* (*ccls-of-cls D*)) *S*
  using *confl* **by** (*elim conflictE*) *auto*

  obtain *D'* **where**
    DD': *mset-cls D' = mset-cls D* **and**
    D': *D'* !∈! *raw-clauses S'*
    using *D SS' in-mset-clss-exists-preimage* **by** *fastforce*

  show *?thesis*
    apply (*rule conflict-rule*[*of - D'*])
    using *state-eq-sym*[*of S S'*] *SS' conf D' DD' tr TT' T*
    by (*auto simp*: *state-eq-def simp del*: *state-simp*)
qed

lemma *backtrack-levE*[*consumes 2*]:
backtrack S S′ $\Longrightarrow$ *cdcl*$_W$*-M-level-inv S* $\Longrightarrow$
($\bigwedge$*K i M1 M2 L D*.
raw-conflicting S = *Some D* $\Longrightarrow$
L ∈# *mset-ccls D* $\Longrightarrow$
(*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*)) $\Longrightarrow$
get-level (*trail S*) *L* = *backtrack-lvl S* $\Longrightarrow$
get-level (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls D*) $\Longrightarrow$
get-maximum-level (*trail S*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* $\Longrightarrow$
undefined-lit M1 L $\Longrightarrow$
S′ ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls D*)
(*update-backtrack-lvl i*
(*update-conflicting None S*)))) $\Longrightarrow$ *P*) $\Longrightarrow$
P
using *assms* **by** (*induction rule*: *backtrack-induction-lev2*) *metis*
thm *allI*

lemma *backtrack-state-eq-compatible*:
assumes
bt: *backtrack S T* **and**
SS′: *S* ∼ *S′* **and**
TT′: *T* ∼ *T′* **and**
inv: *cdcl*$_W$*-M-level-inv S*
shows *backtrack S′ T′*
proof −
obtain *D L K i M1 M2* **where**
conf: *raw-conflicting S* = *Some D* **and**
L: *L* ∈# *mset-ccls D* **and**
decomp: (*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-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*) (*mset-ccls D*) **and**
max-D: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* **and**
undef: *undefined-lit M1 L* **and**
T: *T* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls D*)
(*update-backtrack-lvl i*
(*update-conflicting None S*))))
using *bt inv* **by** (*elim backtrack-levE*) *metis*
obtain *D′* **where**
D′: *raw-conflicting S′* = *Some D′*
using *SS′ conf* **by** (*cases raw-conflicting S′*) *auto*
have [*simp*]: *mset-ccls D* = *mset-ccls D′*
using *SS′ D′ conf* **by** (*auto simp*: *state-eq-def simp del*: *state-simp*)[]

have *T′*: *T′* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D′*))
(*reduce-trail-to M1* (*add-learned-cls* (*cls-of-ccls D′*)
(*update-backtrack-lvl i* (*update-conflicting None S′*))))
using *TT′* **unfolding** *state-eq-def*
using *decomp undef 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 undef T*
apply (*auto simp*: *state-eq-def simp del*: *state-simp*)[]
using *decomp SS' lev SS' max-D max T'* **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 raw-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: *raw-conflicting S* = *Some E* **and**
L: $-L \notin\#$ *mset-ccls E* **and**
E: *mset-ccls E* $\neq$ {#} **and**
T: $T \sim$ *tl-trail S*
using *skip* **by** (*elim skipE*) *simp*
obtain *E'* **where** *E'*: *raw-conflicting S'* = *Some E'*
using *SS' raw* **by** (*cases raw-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-raw-trail S* = *Propagated L E* **and**
L: $L \in\#$ *mset-cls E* **and**
raw: *raw-conflicting S* = *Some D* **and**
LD: $-L \in\#$ *mset-ccls D* **and**
i: *get-maximum-level* (*trail S*) (*mset-ccls* (*remove-clit* ($-L$) *D*)) = *backtrack-lvl S* **and**

T: $T \sim$ *update-conflicting* (*Some* (*union-ccls* (*remove-clit* $(-L)$ D)
(*ccls-of-cls* (*remove-lit* L E)))) (*tl-trail* S)
using *assms* **by** (*elim resolveE*) *simp*

obtain E' **where**
E': *hd-raw-trail* $S' =$ *Propagated* L E'
using SS' *hd* **by** (*metis* ‹*trail* $S \neq []$› *hd-raw-trail is-proped-def marked-lit.disc*(*3*)
marked-lit.inject(*2*) *mmset-of-mlit.elims state-eq-trail*)
have [*simp*]: *mset-cls* $E =$ *mset-cls* E'
using *hd-raw-trail*[*of* S] *tr* *hd-raw-trail*[*of* S'] *tr* SS' *hd* E'
by (*metis marked-lit.inject*(*2*) *mmset-of-mlit.simps*(*1*) *state-eq-trail*)
obtain D' **where**
D': *raw-conflicting* $S' =$ *Some* D'
using SS' *raw* **by** *fastforce*
have [*simp*]: *mset-ccls* $D =$ *mset-ccls* 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 E' **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 !∈! *raw-learned-clss* S **and**
tr: $\neg$(*trail* S) $\models$*asm clauses* S **and**
C1: *mset-cls* $C \notin$ *set* (*get-all-mark-of-propagated* (*trail* S)) **and**
C2: *mset-cls* $C \notin\#$ *init-clss* S **and**
T: $T \sim$ *remove-cls* C S
using *forget* **by** (*elim forgetE*) *simp*

obtain C' **where**
C': C' !∈! *raw-learned-clss* S' **and**
[*simp*]: *mset-cls* $C' =$ *mset-cls* C
using ‹C !∈! *raw-learned-clss* S› SS' *in-mset-clss-exists-preimage* **by** *fastforce*
show *?thesis*
apply (*rule forget-rule*)
using SS' *conf* **apply** *simp*
using C' **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 ‹$U \sim T'$› $cdcl_W$*-bj-state-eq-compatible*[*of* T U] ‹$cdcl_W$*-bj* T U› **by** *auto*
then show *?case*
using *IH*[*of* T] **by** *auto*
qed

19.3.4 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-lev2*) (*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** (*induct rule*: $cdcl_W$*-all-induct-lev2*)
(*auto simp*: *inv* $cdcl_W$*-M-level-inv-decomp not-in-iff*)

lemma *rtranclp-*$cdcl_W$*-init-clss*:
$cdcl_W{}^{**}$ S T $\Longrightarrow$ $cdcl_W$*-M-level-inv* S $\Longrightarrow$ *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 $\Longrightarrow$ $cdcl_W$*-M-level-inv* S $\Longrightarrow$ *init-clss* S = *init-clss* T
using *rtranclp-*$cdcl_W$*-init-clss*[*of* S T] **unfolding** *rtranclp-unfold* **by** *auto*

19.3.5 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. A more precise version would be to show that either these marked are learned or are in the set of clauses

definition $cdcl_W$*-learned-clause* (S:: *'st*) $\longleftrightarrow$
(*init-clss* S $\models$*psm learned-clss* S
$\wedge$ ($\forall$ T. *conflicting* S = *Some* T $\longrightarrow$ *init-clss* S $\models$*pm* T)
$\wedge$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\subseteq$ *set-mset* (*clauses* S))

lemma $cdcl_W$*-learned-clause-S0-*$cdcl_W$[*simp*]:
$cdcl_W$*-learned-clause* (*init-state* N)
unfolding $cdcl_W$*-learned-clause-def* **by** *auto*

lemma $cdcl_W$*-learned-clss*:
assumes
$cdcl_W$ 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-lev2*)
case (*backtrack K i M1 M2 L D T*) **note** *decomp* = *this*(*3*) **and** *confl* = *this*(*1*) **and** *undef* = *this*(*7*)
and T = *this*(*8*)
show *?case*
using *decomp confl learned undef T* **unfolding** $cdcl_W$*-learned-clause-def*
by (*auto dest*!: *get-all-marked-decomposition-exists-prepend*
simp: *raw-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 raw-clauses-def* **by** *auto*
then have *init-clss S* $\models$*pm mset-cls C* + {#*L*#}
using *trail learned* **unfolding** $cdcl_W$*-learned-clause-def raw-clauses-def*
by (*auto dest*: *true-clss-clss-in-imp-true-clss-cls*)
moreover have *remove1-mset* (− *L*) (*mset-ccls D*) + {#− *L*#} = *mset-ccls D*
using *DL* **by** (*auto simp*: *multiset-eq-iff*)
moreover have *remove1-mset L* (*mset-cls C*) + {#*L*#} = *mset-cls 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 raw-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 learned-clss-restart-state*[*of T*]
by (*auto*
simp: *raw-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 raw-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 raw-clauses-def split*: *if-split-asm*)
qed (*auto simp*: $cdcl_W$*-learned-clause-def raw-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*)

19.3.6 No alien atom in the state

This invariant means that all the literals are in the set of clauses.

definition *no-strange-atm* $S' \longleftrightarrow$ (
    ($\forall$ T. *conflicting* S' = *Some* T $\longrightarrow$ *atms-of* $T \subseteq$ *atms-of-mm* (*init-clss* S'))
  $\wedge$ ($\forall$ 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'))

lemma *no-strange-atm-decomp*:
  assumes *no-strange-atm* S
  shows *conflicting* S = *Some* T $\Longrightarrow$ *atms-of* $T \subseteq$ *atms-of-mm* (*init-clss* S)
  and ($\forall$ L *mark*. *Propagated* L *mark* $\in$ *set* (*trail* S)
    $\longrightarrow$ *atms-of* (*mark*) $\subseteq$ *atms-of-mm* (*init-clss* S))
  and *atms-of-mm* (*learned-clss* S) $\subseteq$ *atms-of-mm* (*init-clss* S)
  and *atm-of* ‘ (*lits-of-l* (*trail* S)) $\subseteq$ *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#} $\in$# A $\Longrightarrow$ $x \in$ *atms-of* C $\Longrightarrow$ $x \in$ *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* (*mset-cls* C) $\subseteq$ *atms-of-mm* (*init-clss* S)
    using C *alien* **unfolding** *no-strange-atm-def*
    by (*auto simp*: *raw-clauses-def atms-of-ms-def dest*!:*in-clss-mset-clss*)
  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 raw-clauses-def* **apply** *auto* **by** *blast*
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*:
$x \in$ *atms-of* (*remove1-mset L C*) $\Longrightarrow$ $x \in$ *atms-of C*
using *in-diffD* **unfolding** *atms-of-def* **by** *fastforce*

lemma *cdcl$_W$-no-strange-atm-explicit*:
assumes
cdcl$_W$ S S' **and**
lev: *cdcl$_W$-M-level-inv S* **and**
conf: $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *atms-of T* $\subseteq$ *atms-of-mm* (*init-clss S*) **and**
marked: $\forall$ *L mark*. *Propagated L mark* $\in$ *set* (*trail S*)
$\longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss S*) **and**
learned: *atms-of-mm* (*learned-clss S*) $\subseteq$ *atms-of-mm* (*init-clss S*) **and**
trail: *atm-of* ‘ (*lits-of-l* (*trail S*)) $\subseteq$ *atms-of-mm* (*init-clss S*)
shows
($\forall$ *T*. *conflicting S'* = *Some T* $\longrightarrow$ *atms-of T* $\subseteq$ *atms-of-mm* (*init-clss S'*)) $\wedge$
($\forall$ *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-lev2*)
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 marked learned trail **unfolding** *no-strange-atm-def* **by** *presburger*
next
case (*decide L*)
then show *?case* **using** *learned marked conf trail* **unfolding** *raw-clauses-def* **by** *auto*
next
case (*skip L C M D*)
then show *?case* **using** *learned marked conf trail* **by** *auto*
next
case (*conflict D T*) **note** *D-S* = *this*(*2*) **and** *T* = *this*(*4*)
have *D*: *atm-of* ‘ *set-mset* (*mset-cls 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* (*mset-cls D*) $\subseteq$ ($\bigcup$*x*$\in$*set-mset* (*init-clss S*). *atms-of x*)

∪ (⋃*x*∈*set-mset* (*learned-clss S*). *atms-of x*)
assume *a2*:
(⋃*x*∈*set-mset* (*learned-clss S*). *atms-of x*) ⊆ (⋃*x*∈*set-mset* (*init-clss S*). *atms-of x*)
assume *xa* ∈# *mset-cls D*
then have *atm-of xa* ∈ *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 ∃ *m*∈*set-mset* (*init-clss S*). *atm-of xa* ∈ *atms-of m*
by *blast*
} **note** *H* = *this*
ultimately show *?case* **using** *conflict.prems T learned marked conf trail*
unfolding *atms-of-def atms-of-ms-def raw-clauses-def*
by (*auto simp add*: *H*)
next
case (*restart T*)
then show *?case* **using** *learned marked conf trail* **by** *auto*
next
case (*forget C T*) **note** *confl* = *this*(*1*) **and** *C* = *this*(*4*) **and** *C-le* = *this*(*5*) **and**
T = *this*(*6*)
have *H*: ⋀*L mark*. *Propagated L mark* ∈ *set* (*trail S*) ⟹ *atms-of mark* ⊆ *atms-of-mm* (*init-clss S*)
using *marked* **by** *simp*
show *?case* **unfolding** *raw-clauses-def* **apply** (*intro conjI*)
using *conf confl T trail C* **unfolding** *raw-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*: *raw-clauses-def lits-of-def*)[]
done
next
case (*backtrack K i M1 M2 L D T*) **note** *confl* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *decomp* = *this*(*3*)
and
undef = *this*(*7*) **and** *T* = *this*(*8*)
have *?C T*
using *conf T decomp undef lev* **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 *marked conf undef confl T decomp lev*
by (*auto simp*: *image-subset-iff raw-clauses-def* $cdcl_W$*-M-level-inv-decomp*)
moreover have *?U T*
using *learned decomp conf confl T undef lev* **unfolding** *raw-clauses-def*
by (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
moreover have *?V T*
using *M conf confl trail T undef decomp lev LD*
by (*auto simp*: $cdcl_W$*-M-level-inv-decomp atms-of-def*
dest!: *get-all-marked-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* ((*remove-clit* (−*L*) *D*) !∪ *ccls-of-cls* ((*remove-lit L C*))))
(*tl-trail S*)
have *?C ?T*
using *confl trail-S conf marked* **by** (*auto dest*!: *in-atms-of-remove1-mset-in-atms-of*)
moreover have *?M ?T*
using *confl trail-S conf marked* **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*)

19.3.7 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 moreover.

definition *distinct-*$cdcl_W$*-state* (*S*::*'st*)
$\longleftrightarrow$ ((∀ *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 ∀ *T*. *conflicting S* = *Some T* ⟶ *distinct-mset T*
and *distinct-mset-mset* (*learned-clss S*)
and *distinct-mset-mset* (*init-clss S*)
and ∀ *L mark*. (*Propagated L mark* ∈ *set* (*trail S*) ⟶ *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*)
shows *conflicting S* = *Some T* ⟹ *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 (*mset-clss N*) ⟹ *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-lev2*)
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-marked-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 raw-clauses-def*
using *learned-clss-restart-state*[*of S*] **by** *auto*
next
case *resolve*
then show *?case*
by (*auto simp add*: *distinct-cdcl$_W$-state-def distinct-mset-set-def raw-clauses-def*
distinct-mset-single-add
intro!: *distinct-mset-union-mset*)
qed (*auto simp*: *distinct-cdcl$_W$-state-def distinct-mset-set-def raw-clauses-def*
dest!: *in-clss-mset-clss 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+*

19.3.8 Conflicts and co

This invariant shows that each mark contains a contradiction only related to the previously defined variable.

abbreviation *every-mark-is-a-conflict* :: *'st ⇒ bool* **where**
every-mark-is-a-conflict S ≡
∀ *L mark a b. a @ Propagated L mark # b = (trail S)*
⟶ (*b* ⊨*as CNot* (*mark* − {#*L*#}) ∧ *L* ∈# *mark*)

definition *cdcl$_W$-conflicting S* ≡
(∀ *T. conflicting S = Some T* ⟶ *trail S* ⊨*as CNot T*)
∧ *every-mark-is-a-conflict S*

lemma *backtrack-atms-of-D-in-M1*:
fixes *M1* :: (*'v, nat, 'v clause*) *marked-lits*
assumes
inv: *cdcl$_W$-M-level-inv S* **and**
undef: *undefined-lit M1 L* **and**
i: *get-maximum-level* (*trail S*) (*mset-ccls* (*remove-clit L D*)) ≡ *i* **and**
decomp: (*Marked K* (*Suc i*) # *M1*, *M2*)
∈ *set* (*get-all-marked-decomposition* (*trail S*)) **and**
S-lvl: *backtrack-lvl S = get-maximum-level* (*trail S*) (*mset-ccls D*) **and**
S-confl: *raw-conflicting S = Some D* **and**
undef: *undefined-lit M1 L* **and**
T: *T* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls D*)
(*update-backtrack-lvl i*
(*update-conflicting None S*)))) **and**
confl: ∀ *T. conflicting S = Some T* ⟶ *trail S* ⊨*as CNot T*
shows *atms-of* (*mset-ccls* (*remove-clit L D*)) ⊆ *atm-of* ‘ *lits-of-l* (*tl* (*trail T*))
proof (*rule ccontr*)
let *?k* = *get-maximum-level* (*trail S*) (*mset-ccls D*)

let *?D = mset-ccls D*
let *?D′ = mset-ccls (remove-clit L D)*
have *trail S ⊨as CNot ?D* **using** *confl S-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*)

obtain *M0* **where** *M*: *trail S = M0 @ M2 @ Marked K (Suc i) # M1*
using *decomp* **by** *auto*

have *max*: *?k = length (get-all-levels-of-marked (M0 @ M2 @ Marked K (Suc i) # M1))*
using *inv* **unfolding** *cdcl*$_W$*-M-level-inv-def S-lvl M* **by** *simp*
assume *a*: ¬ *?thesis*
then obtain *L′* **where**
L′: *L′ ∈ atms-of ?D′* **and**
L′-notin-M1: *L′ ∉ atm-of ' lits-of-l M1*
using *T undef decomp inv* **by** (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
then have *L′-in*: *L′ ∈ atm-of ' lits-of-l (M0 @ M2 @ Marked K (i + 1) # [])*
using *vars-of-D* **unfolding** *M* **by** (*auto dest*: *in-atms-of-remove1-mset-in-atms-of*)
then obtain *L″* **where**
L″ ∈# ?D′ **and**
L″: *L′ = atm-of L″*
using *L′ L′-notin-M1* **unfolding** *atms-of-def* **by** *auto*
have *lev-L″*:
get-level (trail S) L″ = get-rev-level (Marked K (Suc i) # rev M2 @ rev M0) (Suc i) L″
using *L′-notin-M1 L″ M* **by** (*auto simp del*: *get-rev-level.simps*)
have *get-all-levels-of-marked (trail S) = rev [1..<1+?k]*
using *inv S-lvl* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have *get-all-levels-of-marked (M0 @ M2) = rev [Suc (Suc i)..<Suc ?k]*
unfolding *M* **by** (*auto simp:rev-swap[symmetric] dest!*: *append-cons-eq-upt-length-i-end*)

then have *M*: *get-all-levels-of-marked M0 @ get-all-levels-of-marked M2*
= rev [Suc (Suc i)..<Suc (length (get-all-levels-of-marked (M0 @ M2 @ Marked K (Suc i) # M1)))]
unfolding *max* **unfolding** *M* **by** *simp*

have *get-rev-level (Marked K (Suc i) # rev (M0 @ M2)) (Suc i) L″*
≥ Min (set ((Suc i) # get-all-levels-of-marked (Marked K (Suc i) # rev (M0 @ M2))))
using *get-rev-level-ge-min-get-all-levels-of-marked[of L″*
rev (M0 @ M2 @ [Marked K (Suc i)]) Suc i] L′-in
unfolding *L″* **by** (*fastforce simp add*: *lits-of-def*)
also have *Min (set ((Suc i) # get-all-levels-of-marked (Marked K (Suc i) # rev (M0 @ M2))))*
= Min (set ((Suc i) # get-all-levels-of-marked (rev (M0 @ M2)))) **by** *auto*
also have ... *= Min (set ((Suc i) # get-all-levels-of-marked M0 @ get-all-levels-of-marked M2))*
by (*simp add*: *Un-commute*)
also have ... *= Min (set ((Suc i) # [Suc (Suc i)..<2 + length (get-all-levels-of-marked M0)*
+ (length (get-all-levels-of-marked M2) + length (get-all-levels-of-marked M1))]))
unfolding *M* **by** (*auto simp add*: *Un-commute*)
also have ... *= Suc i* **by** (*auto intro*: *Min-eqI*)
finally have *get-rev-level (Marked K (Suc i) # rev (M0 @ M2)) (Suc i) L″ ≥ Suc i* **.**
then have *get-level (trail S) L″ ≥ i + 1*
using *lev-L″* **by** *simp*
then have *get-maximum-level (trail S) ?D′ ≥ 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* $D \subseteq$ *atm-of* ‘ *lits-of-l* M' **and**
    a3: $x \in$ *atms-of* D
  shows $x \notin$ *atm-of* ‘ *lits-of-l* M
proof −
  have *ff1*: $\bigwedge l$ *ms. undefined-lit ms l* $\vee$ *atm-of l*
    $\in$ *set* (*map* (λm. *atm-of* (*lit-of* (m::($'a$, $'b$, $'c$) *marked-lit*))) *ms*)
    by (*simp add*: *defined-lit-map*)
  have *ff2*: $\bigwedge a$. $a \notin$ *atms-of* $D \vee a \in$ *atm-of* ‘ *lits-of-l* M'
    using *a2* **by** (*meson subsetCE*)
  have *ff3*: $\bigwedge a$. $a \notin$ *set* (*map* (λm. *atm-of* (*lit-of m*)) M')
    $\vee$ $a \notin$ *set* (*map* (λm. *atm-of* (*lit-of m*)) M)
    using *a1* **by** (*metis* (*lifting*) *IntI distinct-append empty-iff map-append*)
  have $\forall L\ a\ f.\ \exists l.$ ((a::$'a$) $\notin f$ ‘ L $\vee$ (l::$'a$ *literal*) $\in L$) $\wedge$ ($a \notin f$ ‘ $L \vee f\ l = a$)
    by *blast*
  then show $x \notin$ *atm-of* ‘ *lits-of-l* M
    using *ff3 ff2 ff1 a3* **by** (*metis* (*no-types*) *Marked-Propagated-in-iff-in-lits-of-l*)
qed

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-marked-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-marked-decomposition* (*trail* S'))
  using *assms*(*1*,*2*)
proof (*induct rule*: $cdcl_W$*-all-induct-lev2*)
  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-marked-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-marked-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-marked-decomposition M*)
    = *insert* (*hd* (*get-all-marked-decomposition M*)) (*set* (*tl* (*get-all-marked-decomposition M*)))
    by (*cases get-all-marked-decomposition M*) *auto*
  show *?case*

```
      using decomp tr T unfolding all-decomposition-implies-def
      by (cases hd (get-all-marked-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-marked-decomposition (trail S)) = (a, y)
    by (cases hd (get-all-marked-decomposition (trail S)))
  then have M: trail S = y @ a using get-all-marked-decomposition-decomp by blast
  have M': set (get-all-marked-decomposition (trail S))
    = insert (a, y) (set (tl (get-all-marked-decomposition (trail S))))
    using ay by (cases get-all-marked-decomposition (trail S)) auto
  have unmark-l a ∪ set-mset (init-clss S) ⊨ps unmark-l y
    using decomp ay unfolding all-decomposition-implies-def
    by (cases get-all-marked-decomposition (trail S)) fastforce+
  then have a-Un-N-M: unmark-l a ∪ set-mset (init-clss S)
    ⊨ps unmark-l (trail S)
    unfolding M by (auto simp add: all-in-true-clss-clss image-Un)

  have unmark-l a ∪ set-mset (init-clss S) ⊨p {#L#} (is ?I ⊨p -)
    proof (rule true-clss-cls-plus-CNot)
      show ?I ⊨p remove1-mset L (mset-cls C) + {#L#}
        apply (rule true-clss-clss-in-imp-true-clss-cls[of -
            set-mset (init-clss S) ∪ set-mset (learned-clss S)])
        using learned propa L by (auto simp: raw-clauses-def cdcl_W-learned-clause-def
          true-annot-CNot-diff)
    next
      have unmark-l (trail S) ⊨ps CNot (remove1-mset L (mset-cls C))
        using ‹(trail S) ⊨as CNot (remove1-mset L (mset-cls C))› true-annots-true-clss-clss
        by blast
      then show ?I ⊨ps CNot (remove1-mset L (mset-cls C))
        using a-Un-N-M true-clss-clss-left-right true-clss-clss-union-l-r by blast
    qed
  moreover have ⋀aa b.
      ∀ (Ls, seen)∈set (get-all-marked-decomposition (y @ a)).
        unmark-l Ls ∪ set-mset (init-clss S) ⊨ps unmark-l seen
    ⟹ (aa, b) ∈ set (tl (get-all-marked-decomposition (y @ a)))
    ⟹ unmark-l aa ∪ set-mset (init-clss S) ⊨ps unmark-l b
    by (metis (no-types, lifting) case-prod-conv get-all-marked-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 ∪ set-mset (init-clss S) ⊨ps unmark-l y›
     ay by auto
next
  case (backtrack K i M1 M2 L D  T) note conf = this(1) and LD = this(2) and decomp' = this(3)
and
    lev-L = this(4) and undef = this(7) and T = this(8)
  let ?D = mset-ccls D
  let ?D' = mset-ccls (remove-clit L D)
  have ∀ l ∈ set M2. ¬is-marked l
    using get-all-marked-decomposition-snd-not-marked decomp' by blast
```

obtain *M0* **where** *M*: *trail S* = *M0* @ *M2* @ *Marked K* (*i* + *1*) # *M1*
  using *decomp'* **by** *auto*
show *?case* **unfolding** *all-decomposition-implies-def*
  proof
    fix *x*
    assume *x* ∈ *set* (*get-all-marked-decomposition* (*trail T*))
    then have *x*: *x* ∈ *set* (*get-all-marked-decomposition* (*Propagated L ?D* # *M1*))
      using *T decomp' undef inv* **by** (*simp add*: $cdcl_W$*-M-level-inv-decomp*)
    let *?m* = *get-all-marked-decomposition* (*Propagated L ?D* # *M1*)
    let *?hd* = *hd ?m*
    let *?tl* = *tl ?m*
    consider
        (*hd*) *x* = *?hd*
      | (*tl*) *x* ∈ *set ?tl*
      using *x* **by** (*cases ?m*) *auto*
    then show *case x of* (*Ls*, *seen*) ⇒ *unmark-l Ls* ∪ *set-mset* (*init-clss T*)
      ⊨*ps unmark-l seen*
      proof *cases*
        case *tl*
        then have *x* ∈ *set* (*get-all-marked-decomposition* (*trail S*))
          using *tl-get-all-marked-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-marked-decomposition M1*) = (*M1'*, *M1''*)
          by (*cases hd* (*get-all-marked-decomposition M1*))
        then have *x'*: *x* = (*M1'*, *Propagated L ?D* # *M1''*)
          using ‹*x* = *?hd*› **by** *auto*
        have (*M1'*, *M1''*) ∈ *set* (*get-all-marked-decomposition* (*trail S*))
          using *M1*[*symmetric*] *hd-get-all-marked-decomposition-skip-some*[*OF M1*[*symmetric*],
            of M0 @ *M2* - *i*+*1*] **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 M1 L D i K 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* @ *Marked K* (*i* + *1*) # [])
            using *vars-of-D distinct-atms-of-incl-not-in-other*[*of*
              M0 @*M2* @ *Marked K* (*i* + *1*) # [] *M1*] **unfolding** *M* **by** *auto*
          have *trail S* ⊨*as CNot* (*remove1-mset L* (*mset-ccls 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* @ *Marked K* (*i* + *1*) # []
              M1 CNot ?D'] *conf confl* **unfolding** *M lits-of-def* **by** *simp*
          have *M1* = *M1''* @ *M1'* **by** (*simp add*: *M1 get-all-marked-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* $\models$*pm ?D'* + {#*L*#}
using *conf learned confl LD* **unfolding** *cdcl*$_W$*-learned-clause-def* **by** *auto*
then have *T'*: *unmark-l M1'* $\cup$ *set-mset* (*init-clss S*) $\models$*p ?D'* + {#*L*#} **by** *auto*
have *atms-of* (*?D'* + {#*L*#}) $\subseteq$ *atms-of-mm* (*clauses S*)
using *alien conf LD* **unfolding** *no-strange-atm-def raw-clauses-def* **by** *auto*
then have *unmark-l M1'* $\cup$ *set-mset* (*init-clss S*) $\models$*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-marked-decomposition* (*trail S*)) **and**
confl: $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*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-lev2*)
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* = *mset-cls C* **and** *b* = *trail S*
| (*tl*) *tl a* @ *Propagated L' mark* # *b* = *trail S*
using *T undef* **by** (*cases a*) *fastforce+*
then show *b* $\models$*as CNot* (*mark* − {#*L'*#}) $\wedge$ *L'* $\in$# *mark*
using *mark-confl confl LC* **by** *cases auto*
qed
next
case (*decide L*) **note** *undef*[*simp*] = *this*(*2*) **and** *T* = *this*(*4*)
have $\bigwedge$*a La mark b*. *a* @ *Propagated La mark* # *b* = *Marked L* (*backtrack-lvl S*+*1*) # *trail S*
$\Longrightarrow$ *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 $\forall$ *La mark a b*. *a* @ *Propagated La mark* # *b* = *Propagated L C'* # *M*
$\longrightarrow$ *b* $\models$*as CNot* (*mark* − {#*La*#}) $\wedge$ *La* $\in$# *mark*

using *mark-confl* **unfolding** *skip.hyps*(*1*) **by** *simp*
ultimately show $b \models as\ CNot\ (mark - \{\#L'\#\}) \wedge L' \in\#\ \ mark$ **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* (*mset-cls* (*L* !++ *C*)) # *a*) @ *Propagated* L' *mark* # *b*
= *Propagated L* (*mset-cls* (*L* !++ *C*)) # *M*
using *T tr-S* **by** *auto*
then show $b \models as\ CNot\ (mark - \{\#L'\#\}) \wedge L' \in\#\ \ mark$
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 K i M1 M2 L D T*) **note** *conf* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *decomp* = *this*(*3*)
and
undef = *this*(*7*) **and** *T* = *this*(*8*)
have $\forall\, l \in set\ M2.\ \neg is\text{-}marked\ l$
using *get-all-marked-decomposition-snd-not-marked decomp* **by** *blast*
obtain *M0* **where** *M*: *trail S* = *M0* @ *M2* @ *Marked K* (*i* + *1*) # *M1*
using *decomp* **by** *auto*
have [*simp*]: *trail* (*reduce-trail-to M1* (*add-learned-cls* (*cls-of-ccls* (*insert-ccls L D*))
(*update-backtrack-lvl i* (*update-conflicting None S*)))) = *M1*
using *decomp lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
let *?D* = *mset-ccls D*
let *?D'* = *mset-ccls* (*remove-clit L D*)
show *?case*
proof (*intro allI impI*)
fix *La* :: *'v literal* **and** *mark* :: *'v literal multiset* **and**
a b :: (*'v*, *nat*, *'v literal multiset*) *marked-lit list*
assume *a* @ *Propagated La mark* # *b* = *trail T*
then consider
(*hd-tr*) *a* = [] **and**
(*Propagated La mark* :: (*'v*, *nat*, *'v literal multiset*) *marked-lit*)
= *Propagated L ?D* **and**
b = *M1*
| (*tl-tr*) *tl a* @ *Propagated La mark* # *b* = *M1*
using *M T decomp undef lev* **by** (*cases a*) (*auto simp*: $cdcl_W$*-M-level-inv-def*)
then show $b \models as\ CNot\ (mark - \{\#La\#\}) \wedge La \in\#\ \ mark$
proof *cases*
case *hd-tr* **note** *A* = *this*(*1*) **and** *P* = *this*(*2*) **and** *b* = *this*(*3*)
have *trail S* $\models$*as CNot ?D* **using** *conf 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*)
have *vars-of-D*: *atms-of ?D'* ⊆ *atm-of ' lits-of-l M1*
using *backtrack-atms-of-D-in-M1*[*of S M1 L D i K 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* @ *Marked K* (*i* + *1*) # [])
using *vars-of-D distinct-atms-of-incl-not-in-other*[*of*
M0 @ *M2* @ *Marked K* (*i* + *1*) # [] *M1*] **unfolding** *M* **by** *auto*
then have *M1* ⊨*as CNot ?D'*
using *true-annots-remove-if-notin-vars*[*of M0* @ *M2* @ *Marked K* (*i* + *1*) # []
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**
marked-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-lev2*)
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*)
let *?D* = *mset-ccls D*
have *D*: *Propagated L C'* # *M* ⊨*as CNot* (*mset-ccls D*) **using** *assms skip* **by** *auto*
moreover
have *L* ∉# *?D*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have − *L* ∈ *lits-of-l M*
using *in-CNot-implies-uminus*(*2*)[*of L ?D Propagated L C'* # *M*]
‹*Propagated L C'* # *M* ⊨*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 marked-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* (*mset-cls C*)
let *?D* = *remove1-mset* (−*L*) (*mset-ccls D*)
show *?case*
proof (*intro allI impI*)
fix T'
have *tl* (*trail S*) ⊨*as CNot ?C* **using** *tr marked-confl* **by** *fastforce*
moreover
have *distinct-mset* (*?D* + {#− *L*#}) **using** *confl dist LD*
unfolding *distinct-cdcl*$_W$*-state-def* **by** *auto*
then have −*L* ∉# *?D* **unfolding** *distinct-mset-def*
by (*meson* ‹*distinct-mset* (*?D* + {#− *L*#})› *distinct-mset-single-add*)
have *M* ⊨*as CNot ?D*
proof −
have *Propagated L* (*?C* + {#*L*#}) # *M* ⊨*as CNot ?D* ∪ *CNot* {#− *L*#}
using *confl tr confl-inv LC* **by** (*metis CNot-plus LD insert-DiffM2 option.simps*(*9*))
then show *?thesis*
using *M-lev* ‹− *L* ∉# *?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* ⊨*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 ∀ *T*. *conflicting S* = *Some T* ⟶ *trail S* ⊨*as CNot T*
and ∀ *L mark a b*. *a* @ *Propagated L mark* # *b* = (*trail S*)
⟶ (*b* ⊨*as CNot* (*mark* − {#*L*#}) ∧ *L* ∈# *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* ⊨*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*

19.3.9 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-marked-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-marked-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-marked-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-marked-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-marked-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* (*mset-clss N*)
shows
all-decomposition-implies-m (*init-clss* (*init-state N*))
(*get-all-marked-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*

lemma $cdcl_W$*-only-propagated-vars-unsat*:
assumes
marked: $\forall x \in$ *set M*. $\neg$ *is-marked x* **and**
DN: *D* $\in$# *clauses S* **and**
D: *M* $\models$*as CNot D* **and**
inv: *all-decomposition-implies-m N* (*get-all-marked-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* $\cup$ *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*: *raw-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* $\models$*psm U* **using** *learned-cl state* **unfolding** $cdcl_W$*-learned-clause-def* **by** *auto*
ultimately have *I-D*: *I* $\models$ *D*
using *I DN cons state* **unfolding** *true-clss-clss-def true-clss-def Ball-def*
by (*auto simp add*: *raw-clauses-def*)

have *l0*: {*unmark L* |*L*. *is-marked L* $\wedge$ *L* $\in$ *set M*} = {} **using** *marked* **by** *auto*
have *atms-of-ms* (*set-mset N* $\cup$ *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* $\cup$ *unmark-l M*)
using *tot* **unfolding** *total-over-m-def* **by** *auto*

then have $I \models s$ *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*: $I \models s$ *unmark-l M* **by** *auto*
{
fix *K*
assume $K \in\# D$
then have $-K \in$ *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 $-K \in I$ **using** *IM true-clss-singleton-lit-of-implies-incl lits-of-def* **by** *fastforce* **}**
then have $\neg\ I \models D$ **using** *cons* **unfolding** *true-cls-def true-lit-def consistent-interp-def* **by** *auto*
then show *False* **using** *I-D* **by** *blast*
qed

We have actually a much stronger theorem, namely *all-decomposition-implies ?N* (*get-all-marked-decomposition ?M*) $\Longrightarrow$ *?N* $\cup$ {*unmark L* |*L. is-marked L* $\wedge$ *L* $\in$ *set ?M*} $\models ps$ *unmark-l ?M*, that show that the only choices we made are marked in the formula

lemma
assumes *all-decomposition-implies-m N* (*get-all-marked-decomposition M*)
and $\forall\, m \in$ *set M*. $\neg$*is-marked m*
shows *set-mset N* $\models ps$ *unmark-l M*
proof $-$
have *T*: {*unmark L* |*L. is-marked L* $\wedge$ *L* $\in$ *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

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

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*

19.3.10 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: ∀ *s* ∈# *learned-clss S*. ¬*tautology s*
  shows ∀ *s* ∈# *learned-clss S'*. ¬*tautology s*
  using *assms*
proof (*induct rule*: $cdcl_W$*-all-induct-lev2*)
  case (*backtrack K i M1 M2 L D 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* ⊨*as CNot* (*mset-ccls D*)
      using *conflicting confl* **unfolding** $cdcl_W$*-conflicting-def* **by** *auto*
    then have *lits-of-l* (*trail S*) ⊨*s CNot* (*mset-ccls D*)
      using *true-annots-true-cls* **by** *blast*
  ultimately have ¬*tautology* (*mset-ccls 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** *learned-clss-restart-state state-eq-learned-clss no-tauto*
    by (*metis* (*no-types*, *lifting*) *set-mset-mono subsetCE*)
qed (*auto dest*!: *in-diffD*)

definition *final-*$cdcl_W$*-state* (*S*:: *'st*)
  ⟷ (*trail S* ⊨*asm init-clss S*
    ∨ ((∀ *L* ∈ *set* (*trail S*). ¬*is-marked L*) ∧
       (∃ *C* ∈# *init-clss S*. *trail S* ⊨*as CNot C*)))

definition *termination-*$cdcl_W$*-state* (*S*:: *'st*)
   ⟷ (*trail S* ⊨*asm init-clss S*
     ∨ ((∀ *L* ∈ *atms-of-mm* (*init-clss S*). *L* ∈ *atm-of* ‘ *lits-of-l* (*trail S*))
        ∧ (∃ *C* ∈# *init-clss S*. *trail S* ⊨*as CNot C*)))

19.4 CDCL Strong Completeness

fun *mapi* :: (*'a* ⇒ *nat* ⇒ *'b*) ⇒ *nat* ⇒ *'a list* ⇒ *'b list* **where**
mapi - - [] = [] |
mapi f n (*x* # *xs*) = *f x n* # *mapi f* (*n* − *1*) *xs*

lemma *mark-not-in-set-mapi*[*simp*]: *L* ∉ *set M* ⟹ *Marked L k* ∉ *set* (*mapi Marked i M*)
  by (*induct M arbitrary*: *i*) *auto*

lemma *propagated-not-in-set-mapi*[*simp*]: *L* ∉ *set M* ⟹ *Propagated L k* ∉ *set* (*mapi Marked i M*)
  by (*induct M arbitrary*: *i*) *auto*

lemma *image-set-mapi*:
  f ‘ *set* (*mapi g i M*) = *set* (*mapi* (λ*x i*. *f* (*g x i*)) *i M*)
  by (*induction M arbitrary*: *i*) *auto*

lemma *mapi-map-convert*:
  ∀ *x i j*. *f x i* = *f x j* ⟹ *mapi f i M* = *map* (λ*x*. *f x 0*) *M*
  by (*induction M arbitrary*: *i*) *auto*

lemma *defined-lit-mapi*: *defined-lit* (*mapi Marked i M*) *L* ⟷ *atm-of L* ∈ *atm-of* ‘ *set M*

by (*induction M*) (*auto simp*: *defined-lit-map image-set-mapi mapi-map-convert*)

lemma *cdcl*$_W$*-can-do-step*:
assumes
consistent-interp (*set M*) **and**
distinct M **and**
atm-of ' (*set M*) $\subseteq$ *atms-of-mm* (*mset-clss N*)
shows $\exists$*S*. *rtranclp cdcl*$_W$ (*init-state N*) *S*
$\wedge$ *state S* = (*mapi Marked* (*length M*) *M*, *mset-clss 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* (*mset-clss 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* = (*mapi Marked* (*length M*) *M*, *mset-clss N*, {#}, *length M*, *None*)
using *IH* **by** *blast*
let *?S*$_0$ = *incr-lvl* (*cons-trail* (*Marked L* (*length M* +*1*)) *S*)
have *undefined-lit* (*mapi Marked* (*length M*) *M*) *L*
using *Cons.prems*(*1*,*2*) **unfolding** *defined-lit-def consistent-interp-def* **by** *fastforce*
moreover have *init-clss S* = *mset-clss N*
using *S* **by** *blast*
moreover have *atm-of L* $\in$ *atms-of-mm* (*mset-clss 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-mapi 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

lemma *cdcl*$_W$*-strong-completeness*:
assumes
MN: *set M* $\models$*sm mset-clss N* **and**
cons: *consistent-interp* (*set M*) **and**
dist: *distinct M* **and**
atm: *atm-of* ' (*set M*) $\subseteq$ *atms-of-mm* (*mset-clss N*)
obtains *S* **where**
state S = (*mapi Marked* (*length M*) *M*, *mset-clss 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* = (*mapi Marked* (*length M*) *M*, *mset-clss N*, {#}, *length M*, *None*)
using *cdcl*$_W$*-can-do-step*[*OF cons dist atm*] **by** *auto*
have *lits-of-l* (*mapi Marked* (*length M*) *M*) = *set M*
by (*induct M*, *auto*)

then have *mapi Marked* (*length M*) *M* $\models$*asm mset-clss 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

19.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

19.5.1 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*) *option.simps*(*8*,*9*) *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*) *option.simps*(*9*) *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* ∧ $cdcl_W$*-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 ≠ *None* ⟹ *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 ⟹ *skip S T'* ⟹ *T* ∼ *T'*
by (*fastforce simp*: *state-eq-def simp del*: *state-simp elim*: *skipE*)

lemma *resolve-unique*:
resolve S T ⟹ *resolve S T'* ⟹ *T* ∼ *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 ⟹ ¬*conflict T U*
by (*metis None-eq-map-option-iff conflictE conflicting-update-conflicting option.distinct*(*1*) *state-simp*(*5*))

lemma *no-propagate-after-conflict*:
conflict S T ⟹ ¬*propagate T U*
by (*metis conflictE conflicting-update-conflicting map-option-is-None 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* ∧ *conflicting U* = *None*)
∨ (∃ *T D*. *propagate*** *S T* ∧ *conflict T U* ∧ *conflicting U* = *Some D*)

proof −
  have *propagate*$^{++}$ *S U* ∨ (∃ *T*. *propagate*** *S T* ∧ *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* ⟹ *conflicting U* = *None*
      unfolding *tranclp-unfold-end* **by** (*auto elim*!: *propagateE*)
  moreover
    have ⋀*T*. *conflict T U* ⟹ ∃ *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* ⟹ ¬*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 *full cdcl*$_W$*-cp S' S''*

inductive *cdcl*$_W$*-stgy* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
conflict': *full1 cdcl*$_W$*-cp S S'* ⟹ *cdcl*$_W$*-stgy S S'* |
other': *cdcl*$_W$*-o S S'* ⟹ *no-step cdcl*$_W$*-cp S* ⟹ *full cdcl*$_W$*-cp S' S''* ⟹ *cdcl*$_W$*-stgy S S''*

19.5.2 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** ($\forall$ *l* $\in$ *set M*. $\neg$*is-marked 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*, *nat*, *'v clause*) *marked-lit list* **where**
    trail S' = *M* @ *trail S* **and** $\forall$ *l* $\in$ *set M*. $\neg$*is-marked 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 $\exists$ *M*. *trail S'* = *M* @ *trail S* $\wedge$ ($\forall$ *l* $\in$ *set M*. $\neg$*is-marked l*)
  using *assms* **by** *induction* (*fastforce elim*: *conflictE propagateE*)+

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail*:
  assumes *cdcl$_W$-cp*** *S S'*
  shows $\exists$ *M*. *trail S'* = *M* @ *trail S* $\wedge$ ($\forall$ *l* $\in$ *set M*. $\neg$*is-marked 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*) $\leq$ *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* = λa. ($cdcl_W$*-M-level-inv a* ∧ *no-strange-atm a*)
obtain *T* **where** *T*: *full* (λa *b*. *?inv a* ∧ $cdcl_W$*-cp a b*) *S T*
using $cdcl_W$*-cp-wf wf-exists-normal-form*[*of* λa *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: $\exists S'$. *propagate S S'*
  then obtain S' **where** *propagate S S'* **by** *blast*
  then obtain *E L* **where**
    S: *conflicting S = None* **and**
    E: *E !∈! raw-clauses S* **and**
    LE: *L ∈# mset-cls E* **and**
    tr: *trail S ⊨as CNot (mset-cls (remove-lit L E))* **and**
    undef: *undefined-lit (trail S) L* **and**
    S': $S' \sim$ *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** *raw-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 !∈! raw-clauses S* **and**
      LE: *L ∈# mset-cls E* **and**
      tr: *trail S ⊨as CNot (mset-cls (remove-lit L E))* **and**
      undef: *undefined-lit (trail S) L* **and**
      S': $S' \sim$ *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*: *raw-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'' ∧ (∀ 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

19.5.3 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** *raw-conflicting S* = *None* **and** *trail S* ⊨*as CNot D*
moreover then obtain D' **where**
mset-cls D' = *D* **and**
D' !∈! *raw-clauses S*
using *in-mset-clss-exists-preimage* **unfolding** *raw-clauses-def* **by** *blast*
ultimately show *False*
using *conflict-rule*[*of S* D' *update-conflicting* (*Some* (*ccls-of-cls* 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*) *option.simps*(*9*) *state-eq-conflicting*)

lemma *rtranclp-cdcl$_W$-cp-propa-or-propa-confl*:
  assumes *cdcl$_W$-cp** S U*
  shows *propagate** S U* $\vee$ ($\exists$ *T*. *propagate** S T* $\wedge$ *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* $\vee$ *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: $D \neq \{\#\}$
  consider (*CT*) *conflicting S = None* | (*SD*) *D′* **where** *conflicting S = Some D′*
    by (*cases conflicting S*) *auto*
  then show $\exists L \in\# D$. *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**
        raw-conflicting T = None **and**
        D′: *D′* !∈! *raw-clauses T* **and**
        tr: *trail T* ⊨*as CNot* (*mset-cls D′*) **and**
        U: *U* ∼ *update-conflicting* (*Some* (*ccls-of-cls 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*: *raw-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-marked m*
          by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *full-def rtranclp-cdcl*$_W$*-cp-dropWhile-trail*)

have *trail U* $\models$*as CNot D*
using *tr confl U* **by** (*auto elim*!: *conflictE*)
ultimately obtain *L* **where** $L \in\#$ *D* **and** $-L \in$ *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*, *nat*, ′*v clause*) *marked-lit* **and**
xb :: (′*v*, *nat*, ′*v clause*) *marked-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*
$\cap$ (λ*l*. *atm-of* (*lit-of l*)) ‘ *set* (*trail S*) = {}
moreover assume *a4*: *x* $\in$ *set M*
moreover assume *a5*: *xb* $\in$ *set* (*trail S*)
moreover have *atm-of* (− *L*) = *atm-of L*
by *auto*
ultimately have *False*
by *auto*
}
then have *LS*: *atm-of L* $\notin$ *atm-of* ‘ *lits-of-l* (*trail S*)
using ‹−*L* $\in$ *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 get-all-levels-of-marked* (*trail S*) $\neq$ [], *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-rev-level* (*rev M*) *0 L* = *0*
using *nm* **by** *auto*
ultimately show *?thesis* **using** *LS ne US* **unfolding** *tr-U*
by (*simp add*: *get-all-levels-of-marked-nil-iff-not-is-marked lits-of-def*)
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 *hd* (*get-all-levels-of-marked* (*trail S*)) = *backtrack-lvl S*
using *ne lev* **unfolding** *cdcl*$_W$*-M-level-inv-def*
by (*cases get-all-levels-of-marked* (*trail S*)) *auto*
moreover have *atm-of L* $\in$ *atm-of* ‘ *lits-of-l M*
using ‹−*L* $\in$ *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-beginning-hd-get-all-levels-of-marked*[*OF LS*, *of M*]
get-level-skip-in-all-not-marked[*of rev M L backtrack-lvl S*]
unfolding *lits-of-def US tr-U*
by *auto*
qed

then show $\exists L \in \# D.$ *get-level* (*trail* U) L = *backtrack-lvl* U
using ‹$L \in \# D$› **by** *blast*
qed
qed

19.5.4 Literal of highest level in marked literals

definition *mark-is-false-with-level* :: *'st* $\Rightarrow$ *bool* **where**
mark-is-false-with-level $S' \equiv$
$\forall D\ M1\ M2\ L.\ M1$ @ *Propagated* $L\ D$ # $M2$ = *trail* $S' \longrightarrow D - \{\#L\#\} \neq \{\#\}$
$\longrightarrow (\exists L.\ L \in\#\ D \wedge$ *get-level* (*trail* S') L = *get-maximum-possible-level* $M1$)

definition *no-more-propagation-to-do*:: *'st* $\Rightarrow$ *bool* **where**
no-more-propagation-to-do $S \equiv$
$\forall D\ M\ M'\ L.\ D + \{\#L\#\} \in\#$ *clauses* $S \longrightarrow$ *trail* $S = M'$ @ $M \longrightarrow M \models$*as CNot* D
$\longrightarrow$ *undefined-lit* $M\ L \longrightarrow$ *get-maximum-possible-level* $M <$ *backtrack-lvl* S
$\longrightarrow (\exists L.\ L \in\#\ D \wedge$ *get-level* (*trail* S) L = *get-maximum-possible-level* 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 !∈! *raw-clauses* S **and**
LE: $L \in\#$ *mset-cls* E **and**
tr: *trail* $S \models$*as CNot* (*mset-cls* (*remove-lit* $L\ E$)) **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 (*mset-cls* 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: *get-maximum-possible-level* $M1 <$ *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 (*mset-cls* 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** *raw-clauses-def* **by** *auto*
moreover have *get-maximum-possible-level* $M1 <$ *backtrack-lvl* S
using *get-max* $S\ S'$ *undefL* **by** *auto*
ultimately obtain L' **where** $L' \in\#\ D$ **and**
get-level (*trail* S) L' = *get-maximum-possible-level* $M1$
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'* ∈ *atm-of* ' (*lits-of-l M1*)
   using ‹*L'* ∈# *D*› ‹*M1* ⊨*as CNot D*› **by** (*metis atm-of-uminus image-eqI in-CNot-implies-uminus*(*2*))
  then have *atm-of L'* ∈ *atm-of* ' (*lits-of-l* (*trail S*))
   using ‹*tl M2* @ *M1* = *trail S*›[*symmetric*] *S undefL* **by** *auto*
 ultimately have *atm-of L* ≠ *atm-of L'* **unfolding** *lits-of-def* **by** *auto*
}
ultimately have ∃ *L'* ∈# *D*. *get-level* (*trail S'*) *L'* = *get-maximum-possible-level M1*
 using *S S' undefL* **by** *auto*
}
moreover {
 assume *M2* = [] **and** *M1*: *M1* = *Propagated L* (*mset-cls 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 *get-all-levels-of-marked* (*trail S'*) = *rev* [*Suc 0..<*(*Suc 0+backtrack-lvl S*)]
  using *S' undefL* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
 then have *get-maximum-possible-level M1* = *backtrack-lvl S'*
  using *get-maximum-possible-level-max-get-all-levels-of-marked*[*of M1*] *S' M1 undefL*
  by (*auto intro*: *Max-eqI*)
 then have *False* **using** *get-max* **by** *auto*
}
ultimately show ∃ *L*. *L* ∈# *D* ∧ *get-level* (*trail S'*) *L* = *get-maximum-possible-level 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: *raw-conflicting S = Some E* **and**
LE: *L* ∈# *mset-ccls E* **and**
L: *get-level* (*trail S*) *L = backtrack-lvl S* **and**
D: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls E*)) < *backtrack-lvl S* **and**
bt: *backtrack-lvl S = get-maximum-level* (*trail S*) (*mset-ccls 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*) (*mset-ccls E*)
using *L D bt* **by** (*simp add*: *get-maximum-level-plus*)
let *?i = get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls E*))
obtain *K M1 M2* **where**
K: (*Marked K* (*?i + 1*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*))
using *backtrack-ex-decomp*[*OF M-L*, *of ?i*] *D S* **by** *auto*
show *?thesis* **using** *backtrack-rule*[*OF S LE K L*] *bt L bj* $cdcl_W$*-bj.simps* **by** *auto*
qed

lemma $cdcl_W$*-stgy-final-state-conclusive*:
assumes
termi: $\forall S'$. ¬$cdcl_W$*-stgy S S'* **and**
decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-marked-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* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S*)))
∨ (*conflicting S = None* ∧ *trail S* ⊨*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*) *raw-conflicting S = None*
| (*Some-Empty*) *E* **where** *raw-conflicting S = Some E* **and** *mset-ccls E* = {#}
| (*Some*) *E'* **where** *raw-conflicting S = Some E'* **and**
conflicting S = Some (*mset-ccls E'*) **and** *mset-ccls E'* ≠ {#}
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* ⊨*asm ?N*
      have *atm-of* ' (*lits-of-l ?M*) = *atms-of-mm ?N* (**is** *?A* = *?B*)
        proof
          show *?A* ⊆ *?B* **using** *alien* **unfolding** *no-strange-atm-def* **by** *auto*
          show *?B* ⊆ *?A*
            proof (*rule ccontr*)
              assume ¬*?B* ⊆ *?A*
              then obtain *l* **where** *l* ∈ *?B* **and** *l* ∉ *?A* **by** *auto*
              then have *undefined-lit ?M* (*Pos l*)
                using ‹*l* ∉ *?A*› **unfolding** *lits-of-def* **by** (*auto simp add*: *defined-lit-map*)
              moreover have *conflicting S* = *None*
                using *None* **by** *auto*
              ultimately have ∃ *S'*. $cdcl_W$*-o S S'*
                using $cdcl_W$*-o.decide decide-rule* ‹*l* ∈ *?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** ¬ *?M* ⊨*a D* **and** *D* ∈# *?N*
        using ‹¬*?M* ⊨*asm ?N*› **unfolding** *lits-of-def true-annots-def Ball-def* **by** *auto*
      have *atms-of D* ⊆ *atm-of* ' (*lits-of-l ?M*)
        using ‹*D* ∈# *?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* ⊆ *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* ⊆ *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* ⊨*as CNot D*
        using *total-not-true-cls-true-clss-CNot* ‹¬ *trail S* ⊨*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* ∈# *init-clss S*› *None* ‹*trail S* ⊨*as CNot D*›
            raw-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** *raw-conf* = *this*(*1*) **and** *LD* = *this*(*2*) **and** *nempty* = *this*(*3*)
then obtain *L D* **where**
E'[*simp*]: *mset-ccls 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-a-l*: *get-all-levels-of-marked ?M* = *rev* [*1*..<*1* + *?k*]
using *level-inv lev-L M* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
have *g-k*: *get-maximum-level* (*trail S*) *D* $\leq$ *?k*
using *get-maximum-possible-level-ge-get-maximum-level*[*of ?M*]
get-maximum-possible-level-max-get-all-levels-of-marked[*of ?M*]
by (*auto simp add*: *Max-n-upt g-a-l*)
{
assume *marked*: *is-marked* (*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' l'* **where** *L'*: *hd ?M* = *Marked L' l'* **using** *marked* **by** (*cases hd ?M*) *auto*
have *marked-hd-tl*: *get-all-levels-of-marked* (*hd* (*trail S*) # *tl* (*trail S*))
= *rev* [*1*..<*1* + *length* (*get-all-levels-of-marked ?M*)]
using *level-inv lev-L M* **unfolding** *cdcl*$_W$*-M-level-inv-def M*[*symmetric*]
by *blast*
then have *l'-tl*: *l'* # *get-all-levels-of-marked* (*tl ?M*)
= *rev* [*1*..<*1* + *length* (*get-all-levels-of-marked ?M*)] **unfolding** *L'* **by** *simp*
moreover have . . . = *length* (*get-all-levels-of-marked ?M*)
rev [*1*..<*length* (*get-all-levels-of-marked ?M*)]
using *M Suc-le-mono calculation* **by** (*fastforce simp add*: *upt.simps*(*2*))
finally have
l'-cons: *l'* # *get-all-levels-of-marked* (*tl* (*trail S*)) =
length (*get-all-levels-of-marked* (*trail S*))
rev [*1*..<*length* (*get-all-levels-of-marked* (*trail S*))] **and**
l' = *?k* **and**
g-r: *get-all-levels-of-marked* (*tl* (*trail S*))
= *rev* [*1*..<*length* (*get-all-levels-of-marked* (*trail S*))]
using *level-inv lev-L M* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *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 $\neg$ *?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*] *L' M atm-of-eq-atm-of*
unfolding *lits-of-def consistent-interp-def*
by (*metis* (*mono-tags*, *hide-lams*) *marked-lit.sel*(*1*) *get-level-skip-beginning image-eqI*

list.set-intros(*1*))
moreover
have *length* (*get-all-levels-of-marked* (*trail S*)) = *?k*
using *level-inv* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have *Max* (*set* (*0#get-all-levels-of-marked* (*tl* (*trail S*)))) = *?k* − *1*
unfolding *g-r* **by** (*auto simp add*: *Max-n-upt*)
then have *get-level* (*tl ?M*) *L* < *?k*
using *get-maximum-possible-level-ge-get-level*[*of tl ?M L*]
by (*metis One-nat-def add.right-neutral add-Suc-right diff-add-inverse2*
get-maximum-possible-level-max-get-all-levels-of-marked k′ le-imp-less-Suc
list.simps(*15*))
finally show *False* **using** *lev-L M* **by** *auto*
qed
have *L*: *hd ?M* = *Marked* (−*L*) *?k* **using** ‹*l′* = *?k*› *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′′* ∈# *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* ≠ *L′′* **using** *no-dup* ‹*L′′* ∈# *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* ≠ *L′′*› *get-level-skip-beginning*[*of L′′ hd ?M tl ?M*] **unfolding** *L*
by (*auto simp*: *atm-of-eq-atm-of*)
then show *False*
by (*metis L-k Max-n-upt One-nat-def Suc-n-not-le-n* ‹*l′* = *backtrack-lvl S*›
add-Suc-right add-implies-diff g-r
get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked list.set(*2*)
get-rev-level-less-max-get-all-levels-of-marked k′ l′-cons list.sel(*1*)
rev-rev-ident semiring-normalization-rules(*6*) *set-upt*)
qed
then have *taut*: *tautology* (*D* + {#*L*#})
using ‹*L′′* ∈# *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 raw-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-marked* = *this*

moreover {
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*
assume *nm*: $\neg$*is-marked* (*hd ?M*)
then obtain L' C **where** $L'C$: *hd-raw-trail S* = *Propagated* L' C
by (*metis* ‹*trail S* $\neq$ []› *hd-raw-trail is-marked-def mmset-of-mlit.elims*)
then have *hd ?M* = *Propagated* L' (*mset-cls C*)
using ‹*trail S* $\neq$ []› *hd-raw-trail mmset-of-mlit.simps*(*1*) **by** *fastforce*
then have *M*: *?M* = *Propagated* L' (*mset-cls C*) # *tl ?M*
using ‹*?M* $\neq$ []› *list.collapse* **by** *fastforce*
then obtain C' **where** C': *mset-cls C* = $C' + \{\#L'\#\}$
using *confl* **unfolding** *$cdcl_W$-conflicting-def* **by** (*metis append-Nil diff-single-eq-union*)
{ assume $-L' \notin\#\ ?D$
then have *Ex* (*skip S*)
using *skip-rule*[*OF M raw-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 $L'D$: $-L' \in\#\ ?D$
then obtain D' **where** D': $?D = D' + \{\#-L'\#\}$ **by** (*metis insert-DiffM2*)
have *g-r*: *get-all-levels-of-marked* (*Propagated* L' (*mset-cls C*) # *tl* (*trail S*))
= *rev* [*Suc 0*..<*Suc* (*length* (*get-all-levels-of-marked* (*trail S*)))]
using *level-inv M* **unfolding** *$cdcl_W$-M-level-inv-def* **by** *auto*
have *Max* (*insert 0*
(*set* (*get-all-levels-of-marked* (*Propagated* L' (*mset-cls C*) # *tl* (*trail S*))))) = *?k*
using *level-inv M* **unfolding** *g-r $cdcl_W$-M-level-inv-def set-rev*
by (*auto simp add*:*Max-n-upt*)
then have *get-maximum-level* (*trail S*) $D' \leq ?k$
using *get-maximum-possible-level-ge-get-maximum-level*[*of*
Propagated L' (*mset-cls C*) # *tl ?M*] *M*
unfolding *get-maximum-possible-level-max-get-all-levels-of-marked* **by** *auto*
then have *get-maximum-level* (*trail S*) $D' = ?k$
$\vee$ *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* $\neq$ []› - - *raw-conf*] *raw-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)
unfolding *rev.simps*
apply (*subst get-rev-level-can-skip-correctly-ordered*)
using *level-inv* **unfolding** $cdcl_W$*-M-level-inv-def*
apply (*subst* (*asm*) (*2*) M) **apply** (*simp add*: $cdcl_W$*-M-level-inv-decomp*)
using *level-inv* **unfolding** $cdcl_W$*-M-level-inv-def*
apply (*subst* (*asm*) (*2*) M) **apply** (*auto simp*: $cdcl_W$*-M-level-inv-decomp lits-of-def*)[]
using *level-inv* **unfolding** $cdcl_W$*-M-level-inv-def*
apply (*subst* (*asm*) (*4*) M) **apply** (*auto simp add*: $cdcl_W$*-M-level-inv-decomp*)[]
using *level-inv* **unfolding** $cdcl_W$*-M-level-inv-def*
apply (*subst* (*asm*) (*4*) M) **by** (*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 raw-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* $S\ S' \Longrightarrow cdcl_W{}^{++}\ S\ S'$
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*$^{++}$ $S\ S' \Longrightarrow cdcl_W{}^{++}\ S\ S'$
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* $S\ S' \Longrightarrow cdcl_W{}^{++}\ S\ S'$
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 $S' = S'' \lor cdcl_W\text{-}cp^{++}\ S'\ S''$
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-lev2*)
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*) (*mset-ccls 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 option.simps*(*9*))
moreover have *L-not-D*: *L* $\notin$# *remove1-mset* (−*L*) (*mset-ccls D*)
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *L* $\in$# *mset-ccls D*
by (*auto simp*: *in-remove1-mset-neq*)
moreover have *Propagated L* (*mset-cls C*) # *M* $\models$*as CNot* (*mset-ccls D*)
using *conflicting confl tr-S* **unfolding** *cdcl$_W$-conflicting-def* **by** *auto*
ultimately have −*L* $\in$ *lits-of-l* (*Propagated L* (*mset-cls C*) # *M*)
using *in-CNot-implies-uminus*(*2*) **by** *blast*
moreover have *no-dup* (*Propagated L* (*mset-cls 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 marked-lit.sel*(*2*) *distinct-consistent-interp*)
qed

ultimately
 have *g-D*: *get-maximum-level* (*Propagated L* (*mset-cls C*) # *M*) (*remove1-mset* (−*L*) (*mset-ccls D*))
  = *get-maximum-level M* (*remove1-mset* (−*L*) (*mset-ccls D*))
 proof −
  have ∀ *a f L*. ((*a*::′*v*) ∈ *f* ‘ *L*) = (∃ *l*. (*l*::′*v literal*) ∈ *L* ∧ *a* = *f l*)
   by *blast*
  then show *?thesis*
   using *get-maximum-level-skip-first*[*of L remove1-mset* (−*L*) (*mset-ccls D*) *mset-cls C M*]
   unfolding *atms-of-def*
   by (*metis* (*no-types*) *uL-not-D L-not-D atm-of-eq-atm-of*)
 qed
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*) (*mset-ccls 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-all-levels-of-marked M* = *rev* [*Suc 0*..<*Suc* (*backtrack-lvl S*)]
 using *lev* **unfolding** *tr-S* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *get-maximum-level M* (*remove1-mset L* (*mset-cls C*)) ≤ *backtrack-lvl S*
 using *get-maximum-possible-level-ge-get-maximum-level*[*of M*]
 get-maximum-possible-level-max-get-all-levels-of-marked[*of M*] **by** (*auto simp*: *Max-n-upt*)
then have
 get-maximum-level M (*remove1-mset* (− *L*) (*mset-ccls D*) #∪ *remove1-mset L* (*mset-cls 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*) (*mset-ccls D*) #∪ *remove1-mset L* (*mset-cls 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 ∈# *mset-ccls 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* ∈# *mset-ccls D*› ‹− *L* ∉# *mset-ccls D*›
    by (*auto simp add*: *atm-of-eq-atm-of*)
   have *Propagated L C′* # *M* ⊨*as CNot* (*mset-ccls D*)
    using *conflicting tr-S D* **unfolding** $cdcl_W$*-conflicting-def* **by** *auto*
   then have −*L* ∈ *lits-of-l M*
    using ‹*La* ∈# *mset-ccls D*› *in-CNot-implies-uminus*(*2*)[*of L mset-ccls 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*

19.5.5 Strong completeness

lemma $cdcl_W$*-cp-propagate-confl*:
assumes $cdcl_W$*-cp S T*
shows $propagate^{**}\ S\ T \lor (\exists S'.\ propagate^{**}\ S\ S' \land conflict\ S'\ T)$
using *assms* **by** *induction blast+*

lemma *rtranclp-*$cdcl_W$*-cp-propagate-confl*:
assumes $cdcl_W\text{-}cp^{**}\ S\ T$
shows $propagate^{**}\ S\ T \lor (\exists S'.\ propagate^{**}\ S\ S' \land conflict\ S'\ T)$
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**
$C + \{\#L\#\} \in\#\ local.clauses\ S$ **and**
$M' \models as\ CNot\ C$ **and**
undefined-lit (*trail S*) *L*
proof −
obtain *E L* **where**
conf: *conflicting S* = *None* **and**
E: *E* !∈! *raw-clauses S* **and**
LE: $L \in\#$ *mset-cls E* **and**
tr: *trail S* $\models$*as CNot* (*mset-cls* (*remove-lit L E*)) **and**
undef: *undefined-lit* (*trail S*) *L* **and**
T: $T \sim$ *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* (*mset-cls E*)] *S T LE E tr undef*
by *auto*
qed

lemma $cdcl_W$*-cp-propagate-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**
lits-of-l (*trail S*) $\subseteq$ *set M* **and**
init-clss S = *N* **and**
$propagate^{**}\ S\ S'$ **and**
learned-clss S = {#}
shows *length* (*trail S*) $\leq$ *length* (*trail S'*) $\land$ *lits-of-l* (*trail S'*) $\subseteq$ *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* ⊨*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* ⊨ *C* + {#*L*#}
using *MN C learned Y NS* ‹*init-clss S* = *init-clss Y*› ‹*learned-clss Y* = {#}›
unfolding *true-clss-def raw-clauses-def* **by** *fastforce*
ultimately have *L* ∈ *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* ⊨*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*) ⊆ *set M*
shows ∃ *S'*. *propagate*** *S S'* ∧ *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*) ∃ *X*. *propagate*** *S X* ∧ *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 ∈# *init-clss* X + *learned-clss* X **and**
Not-E: *trail* X ⊨*as CNot* E
using *Xconf* **by** (*auto simp add*: *raw-clauses-def elim*!: *conflictE*)
have *lits-of-l* (*trail* X) ⊆ *set* M
using *cdcl*$_W$*-cp-propagate-completeness*[*OF assms*(*1−3*) *lits - X learned*] *learned* **by** *auto*
then have *MNE*: *set* M ⊨*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 ⊨*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 *cdcl*$_W$*-cp*** *?S ?S'* ⟹ ∃ *M*. *trail ?S'* = *M* @ *trail ?S* ∧ (∀ *l*∈*set M*. ¬ *is-marked l*)

lemma *rtranclp-propagate-is-trail-append*:
propagate** S T ⟹ ∃ *c*. *trail* T = *c* @ *trail* S
by (*induction rule*: *rtranclp-induct*) (*auto elim*: *propagateE*)

lemma *rtranclp-propagate-is-update-trail*:
propagate** S T ⟹ *cdcl*$_W$*-M-level-inv* S ⟹
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 ⊨*s set-mset* (*mset-clss* N) **and**
cons: *consistent-interp* (*set* M) **and**
tot: *total-over-m* (*set* M) (*set-mset* (*mset-clss* N)) **and**
atm-incl: *atm-of* ‘ (*set* M) ⊆ *atms-of-mm* (*mset-clss* N) **and**
distM: *distinct* M **and**

length: $n \le$ *length M*
shows
$\exists M'$ *k S. length* $M' \ge n \wedge$
lits-of-l $M' \subseteq$ *set M* $\wedge$
no-dup $M' \wedge$
state S = $(M'$, *mset-clss N*, {#}, *k*, *None*) $\wedge$
cdcl$_W$*-stgy*** (*init-state N*) *S*
using *length*
proof (*induction n*)
case *0*
have *state* (*init-state N*) = ([], *mset-clss N*, {#}, *0*, *None*)
by (*auto simp*: *state-eq-def simp del*: *state-simp*)
moreover have
0 $\le$ *length* [] **and**
lits-of-l [] $\subseteq$ *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* $M' \ge n$ **and**
M': *lits-of-l* $M' \subseteq$ *set M* **and**
n-d[*simp*]: *no-dup* M' **and**
S: *state S* = $(M'$, *mset-clss 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*: $\neg$*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*) $\le$ *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' = mset-clss 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', mset-clss N, {#}, backtrack-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 ∈# mset-clss 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
              raw-clauses-def dest!: in-clss-mset-clss 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 *m* **where** *m*: $m \in$ *set M* **and** *undef-m*: $m \notin$ *lits-of-l M'* **by** *auto*
moreover have *undef*: *undefined-lit M' m*
using *M' Marked-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 m* $\in$ *atms-of-mm* (*init-clss S*)
using *atm-incl calculation S* **by** *auto*
ultimately
have *dec*: *decide S* (*cons-trail* (*Marked m* (*k+1*)) (*incr-lvl S*))
using *decide-rule*[*of S* -
cons-trail (*Marked m* (*k* + *1*)) (*incr-lvl S*)] *S*
by *auto*
let *?S'* = *cons-trail* (*Marked m* (*k+1*)) (*incr-lvl S*)
have *lits-of-l* (*trail ?S'*) $\subseteq$ *set M* **using** *m 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''*] *m 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* (*Marked m* (*Suc* (*backtrack-lvl S*)))
(*update-backtrack-lvl* (*Suc* (*backtrack-lvl S*)) *S*))
using *S* ‹$cdcl_W$*-M-level-inv* (*cons-trail* (*Marked m* (*k* + *1*)) (*incr-lvl S*))› **by** *auto*
then have *S''*:
state S'' = (*trail S''*, *mset-clss 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

lemma $cdcl_W$*-stgy-strong-completeness*:
assumes
MN: *set M* $\models$*s set-mset* (*mset-clss N*) **and**
cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset* (*mset-clss N*)) **and**
atm-incl: *atm-of* ‘ (*set M*) $\subseteq$ *atms-of-mm* (*mset-clss N*) **and**
distM: *distinct M*
shows
$\exists$ *M' k S*.

lits-of-l M' = *set* M ∧
state S = (M', *mset-clss* N, {#}, k, *None*) ∧
$cdcl_W$*-stgy*** (*init-state* N) S ∧
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 ≤ *length* M' **and**
M'*-M*: *lits-of-l* M' ⊆ *set* M **and**
no-dup: *no-dup* M' **and**
T: *state* T = (M', *mset-clss* 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) ≤ *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' ⊨*asm mset-clss* 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

19.5.6 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*) ≡
(∀ M K i M' D. M' @ *Marked* K i # M = *trail* S ⟶ D ∈# *clauses* S
⟶ ¬M ⊨*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-lev2*)
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 i M' Da
      assume M'' @ Marked K i # M' = trail T
      and D: Da ∈# local.clauses T
      then have tl M'' @ Marked K i # M' = trail S
        ∨ (M'' = [] ∧ Marked K i # M' = Marked L (backtrack-lvl S + 1) # trail S)
        using T undef by (cases M'') auto
      moreover {
        assume tl M'' @ Marked K i # M' = trail S
        then have ¬M' ⊨as CNot Da
          using D T undef no-f confl smaller unfolding no-smaller-confl-def smaller by fastforce
      }
      moreover {
        assume Marked K i # M' = Marked L (backtrack-lvl S + 1) # trail S
        then have ¬M' ⊨as CNot Da using no-f D confl T by auto
      }
      ultimately show ¬M' ⊨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 K i M1 M2 L D T) note confl = this(1) and LD = this(2) and decomp = this(3)
and
    undef = this(7) and T = this(8)
  obtain c where M: trail S = c @ M2 @ Marked K (i+1) # M1
    using decomp by auto

  show ?case
    proof (intro allI impI)
      fix M ia K' M' Da

      assume M' @ Marked K' ia # M = trail T
      then have tl M' @ Marked K' ia # M = M1
        using T decomp undef lev by (cases M') (auto simp: cdcl_W-M-level-inv-decomp)
      let ?S' = (cons-trail (Propagated L (cls-of-ccls D))
                (reduce-trail-to M1 (add-learned-cls (cls-of-ccls 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' @ Marked K' ia # M = M1› M confl undef smaller
          unfolding no-smaller-confl-def by auto
      }
      moreover {
        assume Da: Da = mset-ccls D
        have ¬M ⊨as CNot Da
          proof (rule ccontr)
            assume ¬ ?thesis
```

then have $-L \in$ *lits-of-l M*
using *LD* **unfolding** *Da* **by** (*simp add*: *in-CNot-implies-uminus*(*2*))
then have $-L \in$ *lits-of-l* (*Propagated L* (*mset-ccls D*) # *M1*)
using *UnI2* ‹*tl M′* @ *Marked K′ ia* # *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* (*mset-ccls D*) # *M1*)
using *decomp undef lev* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
ultimately show *False*
using *undef* **by** (*auto simp*: *Marked-Propagated-in-iff-in-lits-of-l*)
qed
}
ultimately show ¬*M* ⊨*as CNot Da*
using *T undef 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 i M″ D*
assume *M′*: *M″* @ *Marked K i* # *M′* = *trail S′*
and *D* ∈# *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*#} ∈# *clauses S* **and**
M ⊨*as CNot C* **and**
undefined-lit M L
using *propagate* **by** (*auto elim*: *propagate-high-levelE*)
have *tl M″* @ *Marked K i* # *M′* = *trail S* **using** *M′ S S′*
by (*metis Pair-inject list.inject list.sel*(*3*) *marked-lit.distinct*(*1*) *self-append-conv2*
tl-append2)
then have ¬*M′* ⊨*as CNot D*
using ‹*D* ∈# *clauses S′*› *n-l S S′ raw-clauses-def* **unfolding** *no-smaller-confl-def* **by** *auto*
then show ¬*M′* ⊨*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 ¬(∀ D∈#*init-clss* S' + *learned-clss* S'. ¬ *trail* S' ⊨*as CNot* D)
and *conflicting* S' = *None*
shows ∃ S''. *conflict* S' S''
using *assms raw-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-lev2*)
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** *lits-of-l* (*Marked L* (*?k* + *1*) # *?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 $\exists L \in\# x$. *lits-of-l ?M* $\models$*l L* **unfolding** *Bex-def*
using $L'(1)$ $L'(2)$ ‹$-$ $L \notin\# D$› ‹$L'' \in\# x$›
‹*lits-of-l* (*Marked L* (*backtrack-lvl S* + *1*) # *trail S*) $\models$*l* L''› **by** *auto*
qed
then show *False* **using** ‹$\neg$ *?M* $\models$*as CNot D*› **by** *auto*
qed
have *atm-of L* $\notin$ *atm-of* ‘ (*lits-of-l ?M*)
using *undef defined-lit-map* **unfolding** *lits-of-def* **by** *fastforce*
then have *get-level* (*Marked L* (*?k* + *1*) # *?M*) ($-L$) = *?k* + *1* **by** *simp*
then show $\exists$ *La*. *La* $\in\#$ *D* $\wedge$ *get-level* $?M'$ *La* = *backtrack-lvl T*
using ‹$-L \in\# D$› *T undef* **by** *auto*
qed
next
case *resolve*
then show *?case* **by** *auto*
next
case *skip*
then show *?case* **by** *auto*
next
case (*backtrack K i M1 M2 L D T*) **note** *decomp* = *this*(*3*) **and** *undef* = *this*(*7*) **and** *T* = *this*(*8*)
show *?case*
proof (*rule HOL.disjI2*, *clarify*)
fix *Da*
assume *Da*: *Da* $\in\#$ *clauses T*
and *M-D*: *trail T* $\models$*as CNot Da*
obtain *c* **where** *M*: *trail S* = *c* @ *M2* @ *Marked K* (*i* + *1*) # *M1*
using *decomp* **by** *auto*
have *tr-T*: *trail T* = *Propagated L* (*mset-ccls D*) # *M1*
using *T decomp undef 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* $\notin$ *lits-of-l M1*
using *lev* $cdcl_W$*-M-level-inv-def Marked-Propagated-in-iff-in-lits-of-l undef* **by** *blast*
{ assume *Da* $\in\#$ *clauses S*
then have $\neg$*M1* $\models$*as CNot Da* **using** *no-l M* **unfolding** *no-smaller-confl-def* **by** *auto*
}
moreover {
assume *Da*: *Da* = *mset-ccls D*
have $\neg$*M1* $\models$*as CNot Da* **using** ‹$-$ *L* $\notin$ *lits-of-l M1*› **unfolding** *Da*
using *backtrack.hyps*(*2*) *in-CNot-implies-uminus*(*2*) **by** *auto*
}
ultimately have $\neg$*M1* $\models$*as CNot Da*
using *Da T undef decomp lev* **by** (*fastforce simp*: $cdcl_W$*-M-level-inv-decomp*)
then have $-L \in\#$ *Da*
using *M-D* ‹$-$ *L* $\notin$ *lits-of-l M1*› *T* **unfolding** *tr-T true-annots-true-cls true-clss-def*
by (*auto simp*: *uminus-lit-swap*)
have *g-M1*: *get-all-levels-of-marked M1* = *rev* [*1..*<*i*+*1*]
using *lev* lev' *T decomp undef* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
have *no-dup* (*Propagated L* (*mset-ccls D*) # *M1*)
using *lev* lev' *T decomp undef* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *L*: *atm-of L* $\notin$ *atm-of* ‘ *lits-of-l M1* **unfolding** *lits-of-def* **by** *auto*

have *get-level* (*Propagated L* (*mset-ccls D*) # *M1*) ($-L$) = *i*
using *get-level-get-rev-level-get-all-levels-of-marked*[*OF L*,
of [*Propagated L* (*mset-ccls D*)]]
by (*simp add*: *g-M1 split*: *if-splits*)
then show ∃ *La*. *La* ∈# *Da* ∧ *get-level* (*trail T*) *La* = *backtrack-lvl T*
using ‹$-L$ ∈# *Da*› *T decomp undef lev* **by** (*auto simp*: $cdcl_W$*-M-level-inv-def*)
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*∈#*clauses U* **and**
trS: *trail S* ⊨*as CNot D*
using *confl raw-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* ⊨*as CNot D*
apply (*rule true-annots-mono*)
using *trS* **unfolding** *M* **by** *simp-all*
have ∃ *V*. *conflict U V*
using ‹*conflicting U* = *None*› *D raw-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: ∃ *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 D*. *propagate*** *S T* ∧ *conflict T U*
∧ *trail T* ⊨*as CNot D* ∧ *conflicting U* = *Some D* ∧ *D* ∈# *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*: *raw-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-marked 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*, *nat*, *'v clause*) *marked-lit* **and**
xb :: (*'v*, *nat*, *'v clause*) *marked-lit*
assume *a1*: *x* ∈ *set* (*trail S'*) **and**
a2: *xb* ∈ *set M* **and**
a3: (λ*l*. *atm-of* (*lit-of l*)) ‘ *set M* ∩ (λ*l*. *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*, *nat*, *'v clause*) *marked-lit* **and**
xb :: (*'v*, *nat*, *'v clause*) *marked-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: (λ*l*. *atm-of* (*lit-of l*)) ‘ *set M* ∩ (λ*l*. *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* $\notin$ *atm-of ‘ lits-of-l* (*trail S′*)
using *nd* ⟨*L* $\in\#$ *D*⟩ *LM* **unfolding** *M* **by** (*auto simp add*: *lits-of-def*)
show *?thesis*
proof *cases*
assume *ne*: *get-all-levels-of-marked* (*trail S′*) = []
have *backtrack-lvl S″* = *0*
using *inv ne nm* **unfolding** $cdcl_W$*-M-level-inv-def M*
by (*simp add*: *get-all-levels-of-marked-nil-iff-not-is-marked*)
moreover
have *a1*: *get-level M L* = *0*
using *nm* **by** *auto*
then have *get-level* (*M* @ *trail S′*) *L* = *0*
by (*metis LS′ get-all-levels-of-marked-nil-iff-not-is-marked*
get-level-skip-beginning-not-marked lits-of-def ne)
ultimately show *?thesis* **using** ⟨*conflicting S″* = *Some D*⟩ ⟨*L* $\in\#$ *D*⟩ **unfolding** *M*
by *auto*
next
assume *ne*: *get-all-levels-of-marked* (*trail S′*) $\neq$ []
have *hd* (*get-all-levels-of-marked* (*trail S′*)) = *backtrack-lvl S′*
using *ne lev′ M nm* **unfolding** $cdcl_W$*-M-level-inv-def*
by (*cases get-all-levels-of-marked* (*trail S′*))
(*simp-all add*: *get-all-levels-of-marked-nil-iff-not-is-marked*[*symmetric*])
moreover have *atm-of L* $\in$ *atm-of ‘ lits-of-l M*
using ⟨$-L$ $\in$ *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* ⟨*L*$\in\#$*D*⟩ ⟨*conflicting S″* = *Some D*⟩
get-level-skip-beginning-hd-get-all-levels-of-marked[*OF LS′*, *of M*]
get-level-skip-in-all-not-marked[*of rev M L backtrack-lvl S′*]
unfolding *lits-of-def btS M*
by *auto*
qed
qed
}
ultimately have *conflict-is-false-with-level S″* **by** *blast*
}
moreover
{
assume *conflicting S′* $\neq$ *None*
have *no-clause-is-false S′* **using** ⟨*conflicting S′* $\neq$ *None*⟩ **by** *auto*
then have *conflict-is-false-with-level S″* **using** *calculation*(*3*) **by** *presburger*
}
ultimately show *?case* **by** *fast*
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-marked-decomposition* (*trail S*)) **and**
learned: *cdcl$_W$-learned-clause S* **and**
alien: *no-strange-atm S*
shows *no-smaller-confl S'* ∧ *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

19.5.7 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* (*mset-clss N*)
and *no-empty*: ∀ *D*∈#*mset-clss N*. *D* ≠ {#}
shows (*conflicting S'* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S'*)))
∨ (*conflicting S'* = *None* ∧ *trail S'* ⊨*asm init-clss S'*)
proof −
let *?S* = *init-state N*
have
termi: ∀ *S''*. ¬*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-marked-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*∈#*mset-clss 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 $\neq$ *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* $\neq$ *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* $\neq$ *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' $\Longrightarrow$ *conflicting S* = *Some* {#} $\Longrightarrow$ *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-marked 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-marked-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-raw-trail S* = *Propagated K p'* **and**
pp': *mset-cls p'* = *p*
using *hd-raw-trail*[*of S*] *S K* **by** (*cases hd-raw-trail S*) *auto*
obtain *raw-D* **where**
raw-D: *raw-conflicting S* = *Some raw-D*
using *S E* **by** (*cases raw-conflicting S*) *auto*
then have *raw-DD*: *mset-ccls raw-D* = *D*
using *S E* **by** *auto*
consider (*D*) *D* = {#} | (*D'*) *D* $\neq$ {#} **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 hd-raw-trail*[*of S*] **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 $\bigwedge$*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-marked*
get-maximum-level-exists-lit nm not-gr0)
qed
then obtain *T* **where** *sk*: *resolve S T*
using *resolve-rule*[*of S K p′ raw-D*] *S p′* ‹*K* ∈# *p*› *raw-D LD*
unfolding *K′ D E pp′ raw-DD* **by** *auto*
then have *res*: *no-step skip S*
using *LD S D′ K hd-raw-trail*[*of S*] **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*: ∃ *T*. *cdcl*$_W$*-stgy S T*
using ‹*no-step cdcl*$_W$*-cp S*› *other′* **by** (*meson bj resolve skip*)
have *get-all-marked-decomposition* (*L* # *M*) = [([], *L*#*M*)]
using *nm* **unfolding** *K* **apply** (*induction M rule*: *marked-lit-list-induct*, *simp*)
by (*rename-tac L l xs*, *case-tac hd* (*get-all-marked-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* ∨ *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* (*mset-clss N*)
and *empty*: {#} ∈# (*mset-clss N*)
shows *conflicting S'* = *Some* {#} ∧ *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'* ∨ *S'* = *?S* **unfolding** *rtranclp-unfold* **by** *auto*
have ∃*S''*. *conflict ?S S''*
using *empty not-conflict-not-any-negated-init-clss*[*of init-state N*] **by** *auto*

then have $cdcl_W$*-stgy*: ∃*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'* ≠ *?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 ∃ *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* = *mset-clss N*
using *fullSt* $cdcl_W$*-stgy-no-more-init-clss*[*OF St*] **by** *auto*
have *conflicting St* ≠ *None*
proof (*rule ccontr*)
assume *conf*: ¬ *?thesis*
obtain *E* **where**
ES: *E* !∈! *raw-init-clss St* **and**
E: *mset-cls E* = {#}
using *empty cls-St* **by** (*metis in-mset-clss-exists-preimage*)
then have ∃ *T*. *conflict St T*
using *empty cls-St conflict-rule*[*of St E*] *ES conf* **unfolding** *E*
by (*auto simp*: *raw-clauses-def dest*: *in-mset-clss-exists-preimage*)
then show *False* **using** *fullSt* **unfolding** *full1-def* **by** *blast*
qed

have *1*: ∀ *m*∈*set* (*trail St*). ¬ *is-marked 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-marked-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* ≠ *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* ≠ *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'* = *mset-clss N*

using ‹$cdcl_W$*-stgy*** (*init-state* N) S'› *rtranclp-cdcl*$_W$*-stgy-no-more-init-clss* **by** *fastforce*
moreover have *unsatisfiable* (*set-mset* (*mset-clss* N))
by (*meson empty satisfiable-def true-cls-empty true-clss-def*)
ultimately show *?thesis* **by** *auto*
qed

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* (*mset-clss* 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*

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* (*mset-clss* N)
shows (*conflicting* S' = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*mset-clss* N)))
∨ (*conflicting* S' = *None* ∧ *trail* S' ⊨*asm* (*mset-clss* N) ∧ *satisfiable* (*set-mset* (*mset-clss* N)))
proof –
have N: *init-clss* S' = (*mset-clss* 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

19.6 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* ⟷
    no-strange-atm S ∧
    $cdcl_W$*-M-level-inv S* ∧
    (∀ *s* ∈# *learned-clss S*. ¬*tautology s*) ∧
    distinct-$cdcl_W$*-state S* ∧
    $cdcl_W$*-conflicting S* ∧
    all-decomposition-implies-m (*init-clss S*) (*get-all-marked-decomposition* (*trail S*)) ∧
    $cdcl_W$*-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-marked-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 ∀ *s*∈#*learned-clss S'*. ¬ *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* ⟹ $cdcl_W$*-all-struct-inv S* ⟹ $cdcl_W$*-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* ⟹ $cdcl_W$*-all-struct-inv S* ⟹ $cdcl_W$*-all-struct-inv T*
  by (*induction rule*: *rtranclp-induct*) (*auto intro*: $cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*)

19.7 No Relearning of a clause

lemma $cdcl_W$*-o-new-clause-learned-is-backtrack-step*:
  assumes *learned*: *D* ∈# *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-lev2*)
case (*backtrack K i M1 M2 L C T*) **note** *decomp = this*(*3*) **and** *undef = this*(*6*) **and** *andef = this*(*7*) **and**
T = this(*8*) **and** *D-T = this*(*9*) **and** *D-S = this*(*10*)
then have *D = mset-ccls 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*(*3−6*)
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-Marked-lit*:
assumes *propagate* S S'
shows *Marked* K $i \in$ *set* (*trail* S) $\longleftrightarrow$ *Marked* K $i \in$ *set* (*trail* S')
using *assms* **by** (*auto elim*: *propagateE*)

lemma *conflict-no-more-Marked-lit*:
assumes *conflict* S S'
shows *Marked* K $i \in$ *set* (*trail* S) $\longleftrightarrow$ *Marked* K $i \in$ *set* (*trail* S')
using *assms* **by** (*auto elim*: *conflictE*)

lemma *cdcl*$_W$*-cp-no-more-Marked-lit*:
assumes *cdcl*$_W$*-cp* S S'
shows *Marked* K $i \in$ *set* (*trail* S) $\longleftrightarrow$ *Marked* K $i \in$ *set* (*trail* S')
using *assms* **apply** (*induct rule*: *cdcl*$_W$*-cp.induct*)
using *conflict-no-more-Marked-lit propagate-no-more-Marked-lit* **by** *auto*

lemma *rtranclp-cdcl*$_W$*-cp-no-more-Marked-lit*:
assumes *cdcl*$_W$*-cp*** S S'
shows *Marked* K $i \in$ *set* (*trail* S) $\longleftrightarrow$ *Marked* K $i \in$ *set* (*trail* S')
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
using *cdcl*$_W$*-cp-no-more-Marked-lit* **by** *blast+*

lemma *cdcl*$_W$*-o-no-more-Marked-lit*:
assumes *cdcl*$_W$*-o* S S' **and** *lev*: *cdcl*$_W$*-M-level-inv* S **and** $\neg$*decide* S S'
shows *Marked* K $i \in$ *set* (*trail* S') $\longrightarrow$ *Marked* K $i \in$ *set* (*trail* S)
using *assms*

proof (*induct rule*: *cdcl$_W$-o-induct-lev2*)
  case *backtrack* **note** *decomp* = *this*(*3*) **and** *undef* = *this*(*7*) **and** *T* = *this*(*8*)
  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-marked-at-beginning-is-decide*:
  assumes *cdcl$_W$-stgy S S′* **and**
  lev: *cdcl$_W$-M-level-inv S* **and**
  trail S′ = *M′* @ *Marked L i* # *M* **and**
  trail S = *M*
  shows ∃ *T*. *decide S T* ∧ *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 *Marked L i* ∈ *set* (*trail S′*) **and** *Marked L i* ∉ *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-Marked-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 *Marked L i* ∈ *set* (*trail U*) **and** *Marked L i* ∉ *set* (*trail S*)
    using *no-dup* **unfolding** *S S′ cdcl$_W$-M-level-inv-def* **by** (*auto simp add*: *rev-image-eqI*)
  then have *Marked L i* ∈ *set* (*trail T*)
    using *st rtranclp-cdcl$_W$-cp-no-more-Marked-lit* **unfolding** *full-def* **by** *blast*
  then show *?case*
    using *cdcl$_W$-o-no-more-Marked-lit*[*OF o*] ‹*Marked L i* ∉ *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′* @ *Marked L i* # *H* @ *M***and**
  ¬ (∃ *M′*. *trail S* = *M′* @ *Marked L i* # *H* @ *M*)
  shows *decide S T*
  using *assms*
proof (*induction rule*:*cdcl$_W$-o-induct-lev2*)
  case (*backtrack K i M1 M2 L D T*)
  then obtain *c* **where** *trail S* = *c* @ *M2* @ *Marked K* (*Suc i*) # *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* @ *Marked K* (*Suc i*) # *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-marked-at-beginning-is-decide*:
  assumes *cdcl*$_W$*-stgy*** *R U* **and**
  trail U = *M′* @ *Marked L i* # *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* = *Marked L i* # *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′* @ *Marked L i* # *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** *nmarked*: ∀ *l*∈*set* M_0. ¬ *is-marked l*
       using *rtranclp-cdcl*$_W$*-cp-dropWhile-trail* **unfolding** *full1-def rtranclp-unfold* **by** *meson*
      then have *MV*: *M′* @ *Marked L i* # *H* @ *M* = M_0 @ *trail T* **unfolding** *W* **by** *simp*
      then have *V*: *trail T* = *drop* (*length* M_0) (*M′* @ *Marked L i* # *H* @ *M*)
       by *auto*
      have *takeWhile* (*Not o is-marked*) *M′* = M_0 @ *takeWhile* (*Not* ∘ *is-marked*) (*trail T*)
       using *arg-cong*[*OF MV*, *of takeWhile* (*Not o is-marked*)] *nmarked*
       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** *nmarked*: ∀ *l*∈*set* M_0. ¬ *is-marked l*
       using *rtranclp-cdcl*$_W$*-cp-dropWhile-trail cp* **unfolding** *full-def* **by** *meson*
      then have *MV*: *M′* @ *Marked L i* # *H* @ *M* = M_0 @ *trail T′* **unfolding** *U* **by** *simp*
      then have *V*: *trail T′* = *drop* (*length* M_0) (*M′* @ *Marked L i* # *H* @ *M*)
       by *auto*
      have *takeWhile* (*Not o is-marked*) *M′* = M_0 @ *takeWhile* (*Not* ∘ *is-marked*) (*trail T′*)
       using *arg-cong*[*OF MV*, *of takeWhile* (*Not o is-marked*)] *nmarked*
       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′* @ *Marked L i* # *H* @ *M* **using** *V* **by** *auto*

then have LT': *Marked L i* $\in$ *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-Marked-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'*› ‹*Marked L i* $\in$ *set* (*trail T'*)› *nd tr-T'*
by (*auto simp add*: *Cons-eq-append-conv elim*: *decideE*)
have T': *drop* (*length* M_0) M' @ *Marked L i* # *H* @ *M* = *Marked L i* # *trail T*
using ‹*decide T T'*› ‹*Marked L i* $\in$ *set* (*trail T'*)› *nd tr-T'*
by (*auto elim*: *decideE*)
have *trail T'* = *Marked L i* # *trail T*
using ‹*decide T T'*› ‹*Marked L i* $\in$ *set* (*trail T'*)› *tr-T'*
by (*auto elim*: *decideE*)
then have *5*: *trail T'* = *Marked L i* # *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'* = *Marked L i* # *trail T*›
‹*trail T'* = *drop* (*length* M_0) M' @ *Marked L i* # *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' @ *Marked L i* # *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''* = *Marked L i* # *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-marked-at-beginning-is-decide'*:
assumes *cdcl*$_W$*-stgy*** *R U* **and**
trail U = M' @ *Marked L i* # *H* @ *M* **and**

trail $R = M$ **and**
cdcl$_W$*-M-level-inv* R
shows $\exists y\ y'.$ *cdcl*$_W$*-stgy*** $R\ y \land$ *cdcl*$_W$*-stgy* $y\ y' \land \neg$ ($\exists c.$ *trail* $y = c$ @ *Marked* $L\ i$ # H @ M)
$\land$ ($\lambda a\ b.$ *cdcl*$_W$*-stgy* $a\ b \land$ ($\exists c.$ *trail* $a = c$ @ *Marked* $L\ i$ # 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* $T =$ *Marked* $L\ i$ # 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-marked-at-beginning-is-decide*[*OF assms*] **by** *blast*
have *n*: $\neg$ ($\exists c.$ *trail* $S' = c$ @ *Marked* $L\ i$ # 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* $a = c$ @ *Marked* $L\ i$ # H @ M), *OF* $S'T'\ T'U$ *n*]
by *meson*
qed

lemma *beginning-not-marked-invert*:
assumes *A*: M @ $A = M'$ @ *Marked* $K\ i$ # H **and**
nm: $\forall m\in$*set* M. $\neg$*is-marked* m
shows $\exists M.\ A = M$ @ *Marked* $K\ i$ # H
proof −
have $A =$ *drop* (*length* M) (M' @ *Marked* $K\ i$ # H)
using *arg-cong*[*OF A*, *of drop* (*length* M)] **by** *auto*
moreover have *drop* (*length* M) (M' @ *Marked* $K\ i$ # H) = *drop* (*length* M) M' @ *Marked* $K\ i$ # H
using *nm* **by** (*metis* (*no-types*, *lifting*) *A drop-Cons' drop-append marked-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-marked-is-decide-step*:
assumes *cdcl*$_W$*-stgy* $S\ T$
$\neg$ ($\exists c.$ *trail* $S = c$ @ *Marked* $L\ i$ # H @ M) **and**
($\lambda a\ b.$ *cdcl*$_W$*-stgy* $a\ b \land$ ($\exists c.$ *trail* $a = c$ @ *Marked* $L\ i$ # H @ M))** $T\ U$ **and**
$\exists M'.$ *trail* $U = M'$ @ *Marked* $L\ i$ # H @ M **and**
lev: *cdcl*$_W$*-M-level-inv* S
shows $\exists S'.$ *decide* $S\ S' \land$ *full cdcl*$_W$*-cp* $S'\ T \land$ *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* $T = M'$ @ *Marked* $L\ i$ # H @ M **by** *metis*
obtain M'' **where** M'': *trail* $T = M''$ @ *trail* S **and** *nm*: $\forall m\in$ *set* M''. $\neg$*is-marked* m

using *cp* **unfolding** *full1-def*
by (*metis rtranclp-cdcl$_W$-cp-dropWhile-trail' tranclp-into-rtranclp*)
have *False*
using *beginning-not-marked-invert*[*of M'' trail S M' L i 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' @ Marked L i # H @ M*
using *trU' beginning-not-marked-invert*[*of - trail T - L i H @ M*] **by** *metis*
then obtain *M'* **where** *M'*: *trail T = M' @ Marked L i # H @ M*
by *auto*
with *o lev nd cp ns*
show *?case*
proof (*induction rule*: *cdcl$_W$-o-induct-lev2*)
case (*decide L*) **note** *dec = this*(*1*) **and** *cp = this*(*5*) **and** *ns = this*(*4*)
then have *decide S* (*cons-trail* (*Marked L* (*backtrack-lvl S +1*)) (*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 K j M1 M2 L' D T*) **note** *decomp = this*(*3*) **and** *undef = this*(*7*) **and** *T = this*(*8*) **and** *trT = this*(*12*)
obtain *MS3* **where** *MS3*: *trail S = MS3 @ M2 @ Marked K* (*Suc j*) *# M1*
using *get-all-marked-decomposition-exists-prepend*[*OF decomp*] **by** *metis*
have *tl* (*M' @ Marked L i # H @ M*) *= tl M' @ Marked L i # 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' @ Marked L i # 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 (λ*a b. cdcl$_W$-stgy a b* $\wedge$ ($\exists$*c. trail a = c @ Marked L i # H @ M*))** *T U* **and**
$\exists$*M'. trail U = M' @ Marked L i # H @ M*
shows $\exists$*M'. trail T = M' @ Marked L i # 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**
trM: *trail y = c @ Marked Kh i # H* **and**

DL: *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**
learned: ∀ *T*. *conflicting y* = *Some T* ⟶ *trail y* ⊨*as CNot T* **and**
z: *trail z* = *c′* @ *Marked Kh i* # *H*
shows *D* ∉# *learned-clss z*
using *assms*(*1*−*2*) *trM DL DH LH learned z*
proof (*induction rule*: *cdcl*$_W$*-o-induct-lev2*)
case (*backtrack K j M1 M2 L′ D′ 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** *undef* = *this*(*7*) **and** *T* = *this*(*8*) **and** *z* = *this*(*14*)

obtain *M3* **where** *M3*: *trail y* = *M3* @ *M2* @ *Marked K* (*Suc j*) # *M1*
using *decomp get-all-marked-decomposition-exists-prepend* **by** *metis*
have *M*: *trail y* = *c* @ *Marked Kh i* # *H* **using** *trM* **by** *simp*
have *H*: *get-all-levels-of-marked* (*trail y*) = *rev* [*1*..<*1* + *backtrack-lvl y*]
using *lev* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
have *c′* @ *Marked Kh i* # *H* = *Propagated L′* (*mset-ccls D′*) # *trail* (*reduce-trail-to M1 y*)
using *z decomp undef T lev* **by** (*force simp*: *cdcl*$_W$*-M-level-inv-def*)
then obtain *d* **where** *d*: *M1* = *d* @ *Marked Kh i* # *H*
by (*metis* (*no-types*) *decomp in-get-all-marked-decomposition-trail-update-trail list.inject list.sel*(*3*) *marked-lit.distinct*(*1*) *self-append-conv2 tl-append2*)
have *i* ∈ *set* (*get-all-levels-of-marked* (*M3* @ *M2* @ *Marked K* (*Suc j*) # *d* @ *Marked Kh i* # *H*))
by *auto*
then have *i* > *0* **unfolding** *H*[*unfolded M3 d*] **by** *auto*
show *?case*
proof
assume *D* ∈# *learned-clss T*
then have *DLD′*: *D* = *mset-ccls D′*
using *DL T neq0-conv undef decomp lev* **by** (*fastforce simp*: *cdcl*$_W$*-M-level-inv-def*)
have *L-cKh*: *atm-of L* ∈ *atm-of ‘ lits-of-l* (*c* @ [*Marked Kh i*])
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**
have *get-all-levels-of-marked* (*M3* @ *M2* @ *Marked K* (*j* + *1*) # *M1*)
= *rev* [*1*..<*1* + *backtrack-lvl y*]
using *lev* **unfolding** *cdcl*$_W$*-M-level-inv-def M3* **by** *auto*
from *arg-cong*[*OF this*, *of* λ*a*. (*Suc j*) ∈ *set a*] **have** *backtrack-lvl y* ≥ *j* **by** *auto*

have *DD′*[*simp*]: *remove1-mset L D* = *mset-ccls 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′* ≤ *get-maximum-level* (*trail y*) (*remove1-mset L D*)
using *get-maximum-level-ge-get-level* **by** *blast*
moreover {
have *get-maximum-level* (*trail y*) (*remove1-mset L D*) =
get-maximum-level H (*remove1-mset L D*)
using *DH* **unfolding** *M* **by** (*simp add*: *get-maximum-level-skip-beginning*)
moreover
have *get-all-levels-of-marked* (*trail y*) = *rev* [*1*..<*1* + *backtrack-lvl y*]
using *lev* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have *get-all-levels-of-marked H* = *rev* [*1*..< *i*]

unfolding *M* **by** (*auto dest*: *append-cons-eq-upt-length-i*
simp add: *rev-swap*[*symmetric*])
then have *get-maximum-possible-level H* < *i*
using *get-maximum-possible-level-max-get-all-levels-of-marked*[*of H*] ‹*i* > *0*› **by** *auto*
ultimately have *get-maximum-level* (*trail y*) (*remove1-mset L D*) < *i*
by (*metis* (*full-types*) *dual-order.strict-trans nat-neq-iff not-le*
get-maximum-possible-level-ge-get-maximum-level) **}**
moreover
have *L* ∈# *remove1-mset L′* (*mset-ccls D′*)
using *DLD′*[*symmetric*] *DD′ LD* **by** (*metis in-remove1-mset-neq*)
then have *get-maximum-level* (*trail y*) (*remove1-mset L′* (*mset-ccls D′*)) ≥
get-level (*trail y*) *L*
using *get-maximum-level-ge-get-level* **by** *blast*
moreover {
have *get-all-levels-of-marked* (*c* @ [*Marked Kh i*]) = *rev* [*i*..< *backtrack-lvl y*+*1*]
using *append-cons-eq-upt-length-i-end*[*of rev* (*get-all-levels-of-marked H*) *i*
rev (*get-all-levels-of-marked c*) *Suc 0 Suc* (*backtrack-lvl y*)] *H*
unfolding *M* **apply** (*auto simp add*: *rev-swap*[*symmetric*])
by (*metis* (*no-types*, *hide-lams*) *Nil-is-append-conv Suc-le-eq less-Suc-eq list.sel*(*1*)
rev.simps(*2*) *rev-rev-ident upt-Suc upt-rec*)
have *get-level* (*trail y*) *L* = *get-level* (*c* @ [*Marked Kh i*]) *L*
using *L-cKh LH* **unfolding** *M* **by** *simp*
have *get-level* (*c* @ [*Marked Kh i*]) *L* ≥ *i*
using *L-cKh levL*
‹*get-all-levels-of-marked* (*c* @ [*Marked Kh i*]) = *rev* [*i*..<*backtrack-lvl y* + *1*]›
calculation(*1*,*2*) **by** *auto*
then have *get-level* (*trail y*) *L* ≥ *i*
using *M* ‹*get-level* (*trail y*) *L* = *get-level* (*c* @ [*Marked Kh i*]) *L*› **by** *auto* **}**
moreover
have *get-maximum-level* (*trail y*) (*remove1-mset L′* (*mset-ccls D′*))
< *get-level* (*trail y*) *L*
using ‹*j* ≤ *backtrack-lvl y*› *levL j calculation*(*1*−*4*) **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 *nd*: *no-dup* (*trail y*) **using** *lev* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*

{ assume *D*: *remove1-mset L* (*mset-ccls D′*) = {#}
then have *j*: *j* = *0* **using** *levD j* **by** (*simp add*: *LL′*)
have ∀ *m* ∈ *set M1*. ¬*is-marked m*
using *H* **unfolding** *M3 j*
by (*auto simp add*: *rev-swap*[*symmetric*] *get-all-levels-of-marked-no-marked*
dest!: *append-cons-eq-upt-length-i*)
then have *False* **using** *d* **by** *auto*
}
moreover {
assume *D*[*simp*]: *remove1-mset L* (*mset-ccls D′*) ≠ {#}
have *i* ≤ *j*
using *H* **unfolding** *M3 d* **by** (*auto simp add*: *rev-swap*[*symmetric*]
dest: *upt-decomp-lt*)
have *j* > *0* **apply** (*rule ccontr*)
using *H* ‹ *i* > *0*› **unfolding** *M3 d*
by (*auto simp add*: *rev-swap*[*symmetric*] *dest*!: *upt-decomp-lt*)
obtain *L″* **where**

L''∈# *remove1-mset* L (*mset-ccls* D') **and**
$L''D'$: *get-level* (*trail y*) L'' = *get-maximum-level* (*trail y*)
(*remove1-mset* L (*mset-ccls* 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-rev-level-ge-0-atm-of-in*[*of 0 rev* (*trail y*) L'] ‹*j*>*0*› *levD* $L''D'$
‹*j* ≤ *backtrack-lvl y*› *levL* **by** (*simp add*: *LL' j*)
then have L'' ∈ *lits-of-l* (*Marked Kh i* # *d*)
proof −
{
assume $L''H$: *atm-of* L'' ∈ *atm-of* ' *lits-of-l H*
have *get-all-levels-of-marked H* = *rev* [*1*..<*i*]
using *H* **unfolding** *M*
by (*auto simp add*: *rev-swap*[*symmetric*] *dest*!: *append-cons-eq-upt-length-i*)
moreover have *get-level* (*trail y*) L'' = *get-level H* L''
using $L''H$ **unfolding** *M* **by** *simp*
ultimately have *False*
using *levD* ‹*j*>*0*› *get-rev-level-in-levels-of-marked*[*of rev H 0* L''] ‹*i* ≤ *j*›
unfolding $L''D'$[*symmetric*] *nd*
by (*metis* $L''D'$ *LL' Max-n-upt Nat.le-trans One-nat-def Suc-pred* ‹*0* < *i*›
get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked
get-rev-level-less-max-get-all-levels-of-marked j lessI list.simps(*15*)
not-less rev-rev-ident set-upt)
}
moreover
have *atm-of* L'' ∈ *atm-of* ' *lits-of-l H*
using *DD' DH* ‹L'' ∈# *remove1-mset* L (*mset-ccls* D')› *atm-of-lit-in-atms-of LL' LD*
LD' **by** *fastforce*
ultimately show *?thesis*
using *DD' DH* ‹L'' ∈# *remove1-mset* L (*mset-ccls* D')› *atm-of-lit-in-atms-of*
by *auto*
qed
moreover
have *atm-of* L'' ∈ *atms-of* (*remove1-mset* L (*mset-ccls* D'))
using ‹L''∈# *remove1-mset* L (*mset-ccls* 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 *nd* **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* @ *Marked Kh i* # *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'* @ *Marked Kh i* # *H*
shows $D \notin\#$ *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'* @ *Marked Kh i* # *H*
using *cp beginning-not-marked-invert*[*of - trail T c' Kh i 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* @ *Marked K i* # *H* @ []))** *S z* **and**
cdcl$_W$*-all-struct-inv S* **and**
trail S = *c* @ *Marked K i* # *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'* @ *Marked K i* # *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-S* = *this*(*6*) **and** *trU* = *this*(*7*)
obtain *c* **where** *c*: *trail T* = *c* @ *Marked K i* # *H* **using** *s* **by** *auto*
obtain *c'* **where** *c'*: *trail U* = *c'* @ *Marked K i* # *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-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

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: *raw-conflicting S* = *Some E* **and**
already-learned: *mset-ccls 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 M1 M2-loc K i* **where**
T: *T* $\sim$ *cons-trail* (*Propagated L* (*cls-of-ccls E*))
(*reduce-trail-to M1* (*add-learned-cls* (*cls-of-ccls E*)
(*update-backtrack-lvl i* (*update-conflicting None S*))))
and
decomp: (*Marked K* (*Suc i*) # *M1*, *M2-loc*) $\in$
set (*get-all-marked-decomposition* (*trail S*)) **and**
LD: *L* $\in$# *mset-ccls E* **and**
k: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
level: *get-level* (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls E*) **and**
confl-S: *raw-conflicting S* = *Some E* **and**
i: *i* = *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls E*)) **and**
undef: *undefined-lit M1 L*
using *confl* **by** (*induction rule*: *backtrack-induction-lev2*) *fastforce*
obtain *M2* **where**
M: *trail S* = *M2* @ *Marked K* (*Suc i*) # *M1*
using *get-all-marked-decomposition-exists-prepend*[*OF decomp*] **unfolding** *i* **by** (*metis append-assoc*)
let *?E* = *mset-ccls E*
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 *get-lvls-M*: *get-all-levels-of-marked* (*trail S*) = *rev* [*1*..<*Suc* (*backtrack-lvl S*)]
  using *lev'* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*

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'* ⊆ *atm-of ‘ lits-of-l M1*
  using *backtrack-atms-of-D-in-M1*[*OF lev' undef - decomp - - - T*] *confl-S conf T decomp k level*
  lev' i undef **unfolding** $cdcl_W$*-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*:
  ∀ *x* ∈ *atms-of ?E'*. *x* ∉ *atm-of ‘ lits-of-l* (*M2* @ [*Marked K* (*i* + *1*)])
  unfolding *Set.Ball-def* **apply** (*intro impI allI*)
    apply (*rule vars-of-D distinct-atms-of-incl-not-in-other*[*of*
    M2 @ *Marked K* (*i* + *1*) # [] *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* @ *Marked K* (*i* + *1*) # [] *M1 CNot ?E'*]
  MD' M **by** *simp*

have *get-lvls-M*: *get-all-levels-of-marked* (*trail S*) = *rev* [*1*..<*Suc* (*backtrack-lvl S*)]
  using *lev'* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *backtrack-lvl S* > *0* **unfolding** *M* **by** (*auto split*: *if-split-asm simp add*: *upt.simps*(*2*))

obtain *M1' K' Ls* **where**
  M': *trail S* = *Ls* @ *Marked K'* (*backtrack-lvl S*) # *M1'* **and**
  Ls: ∀ *l* ∈ *set Ls*. ¬ *is-marked l* **and**
  set M1 ⊆ *set M1'*
  proof −
    let *?Ls* = *takeWhile* (*Not o is-marked*) (*trail S*)
    have *MLs*: *trail S* = *?Ls* @ *dropWhile* (*Not o is-marked*) (*trail S*)
      by *auto*
    have *dropWhile* (*Not o is-marked*) (*trail S*) ≠ [] **unfolding** *M* **by** *auto*
    moreover
      from *hd-dropWhile*[*OF this*] **have** *is-marked*(*hd* (*dropWhile* (*Not o is-marked*) (*trail S*)))
        by *simp*
    ultimately
      obtain *K' K'k* **where**
        K'k: *dropWhile* (*Not o is-marked*) (*trail S*)
          = *Marked K' K'k* # *tl* (*dropWhile* (*Not o is-marked*) (*trail S*))
        by (*cases dropWhile* (*Not* ∘ *is-marked*) (*trail S*);
            cases hd (*dropWhile* (*Not* ∘ *is-marked*) (*trail S*)))
          simp-all
    moreover have ∀ *l* ∈ *set ?Ls*. ¬*is-marked l* **using** *set-takeWhileD* **by** *force*
    moreover
      have *get-all-levels-of-marked* (*trail S*)
            = *K'k* # *get-all-levels-of-marked*(*tl* (*dropWhile* (*Not* ∘ *is-marked*) (*trail S*)))
        apply (*subst MLs*, *subst K'k*)
        using *calculation*(*2*) **by** (*auto simp add*: *get-all-levels-of-marked-no-marked*)

then have $K'k =$ *backtrack-lvl S*
using *calculation(2)* **by** (*auto split*: *if-split-asm simp add*: *get-lvls-M upt.simps(2)*)
moreover have *set M1* $\subseteq$ *set* (*tl* (*dropWhile* (*Not o is-marked*) (*trail S*)))
unfolding *M* **by** (*induction M2*) *auto*
ultimately show *?thesis* **using** *that MLs* **by** *metis*
qed

have *get-lvls-M*: *get-all-levels-of-marked* (*trail S*) = *rev* [*1..<Suc* (*backtrack-lvl S*)]
using *lev'* **unfolding** *cdcl*$_W$*-M-level-inv-def* **by** *auto*
then have *backtrack-lvl S* > *0* **unfolding** *M* **by** (*auto split*: *if-split-asm simp add*: *upt.simps(2) i*)

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 *lvls-M1'*: *get-all-levels-of-marked M1'* = *rev* [*1..<backtrack-lvl S*]
using *get-lvls-M Ls* **by** (*auto simp add*: *get-all-levels-of-marked-no-marked M' upt.simps(2)*
split: *if-split-asm*)
have *L-notin*: *atm-of L* $\in$ *atm-of* ‘ *lits-of-l Ls* $\vee$ *atm-of L* = *atm-of K'*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then have *atm-of L* $\notin$ *atm-of* ‘ *lits-of-l* (*Marked K'* (*backtrack-lvl S*) # *rev Ls*) **by** *simp*
then have *get-level* (*trail S*) *L* = *get-level M1' L*
unfolding *M'* **by** *auto*
then show *False* **using** *get-level-in-levels-of-marked*[*of M1' L*] ‹*backtrack-lvl S* > *0*›
unfolding *k lvls-M1'* **by** *auto*
qed
obtain *Y Z* **where**
RY: *cdcl*$_W$*-stgy*** *R Y* **and**
YZ: *cdcl*$_W$*-stgy Y Z* **and**
nt: $\neg$ ($\exists$ *c. trail Y* = *c* @ *Marked K'* (*backtrack-lvl S*) # *M1'* @ []) **and**
Z: (λ*a b. cdcl*$_W$*-stgy a b* $\wedge$ ($\exists$ *c. trail a* = *c* @ *Marked K'* (*backtrack-lvl S*) # *M1'* @ []))** *Z S*
using *rtranclp-cdcl*$_W$*-new-marked-at-beginning-is-decide'*[*OF st' - - lev*, *of Ls K'*
backtrack-lvl S 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'* @ *Marked K'* (*backtrack-lvl S*) # *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-marked-is-decide-step*[*OF YZ nt Z*] *M'* **by** *auto*
have *trY*: *trail Y* = *M1'*
proof –
obtain *M'* **where** *M*: *trail Z* = *M'* @ *Marked K'* (*backtrack-lvl S*) # *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** $\forall$ *m*$\in$*set M''*. $\neg$*is-marked m*
using *Y'Z rtranclp-cdcl*$_W$*-cp-dropWhile-trail'* **unfolding** *full-def* **by** *blast*
obtain *M'''* **where** *trail Y'* = *M'''* @ *Marked K'* (*backtrack-lvl S*) # *M1'*
using *M''* **unfolding** *M*
by (*metis* (*no-types*, *lifting*) ‹$\forall$ *m*$\in$*set M''*. $\neg$ *is-marked m*› *beginning-not-marked-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*: *mset-ccls 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'* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *LM1*: *undefined-lit M1 L*
by (*metis Marked-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*)
obtain *E'* **where**
E': *E'* !∈! *raw-clauses Y* **and**
EE': *mset-cls E' = mset-ccls E*
using *DL in-mset-clss-exists-preimage* **by** *blast*
have *Ex* (*propagate Y*)
using *propagate-rule*[*of Y E' L*] *DL M1'-D L-trY Y-CT trY LD E'*
by (*auto simp*: *EE'*)
then have *False* **using** ‹*no-step* $cdcl_W$*-cp Y*› *propagate'* **by** *blast*
}
moreover {
assume *DL*: *mset-ccls 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*: *mset-ccls E* ∉# *learned-clss Z*
using *DL lY-lZ YZ* **unfolding** *raw-clauses-def* **by** *auto*
have *?E* ∉#*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*› *raw-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-induction-lev2*) (*auto simp*: $cdcl_W$*-M-level-inv-decomp*)
          then have *E* $\notin$# *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* (*mset-clss N*) **and**
    no-duplicate-in-clause: *distinct-mset-mset* (*mset-clss 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*)

19.8 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) $\leq$ *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**
$\forall s \in\#$ *learned-clss* S. $\neg$*tautology* s
shows *card*(*set-mset* (*learned-clss* S)) $\leq$ *3* ^ *card* (*atms-of-mm* (*learned-clss* S))
proof −
have *set-mset* (*learned-clss* S) $\subseteq$ *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))
$\leq$ *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 *lexn3*[*intro*!, *simp*]:
$a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$
$\Longrightarrow$ ([a::*nat*, b, c], [a', b', c']) $\in$ *lexn* $\{(x, y).\ x < y\}$ *3*
apply *auto*
unfolding *lexn-conv* **apply** *fastforce*
unfolding *lexn-conv* **apply** *auto*
apply (*metis append.simps*(*1*) *append.simps*(*2*))+
done

lemma *cdcl*$_W$*-measure-decreasing*:
fixes S :: *'st*
assumes
cdcl$_W$ S S' **and**
no-restart:
$\neg$(*learned-clss* S $\subseteq\#$ *learned-clss* S' $\wedge$ [] = *trail* S' $\wedge$ *conflicting* S' = *None*)
and
no-forget: *learned-clss* S $\subseteq\#$ *learned-clss* S' **and**
no-relearn: $\bigwedge S'$. *backtrack* S S' $\Longrightarrow \forall T$. *conflicting* S = *Some* T $\longrightarrow$ $T \notin\#$ *learned-clss* S
and
alien: *no-strange-atm* S **and**
M-level: *cdcl*$_W$*-M-level-inv* S **and**
no-taut: $\forall s \in\#$ *learned-clss* S. $\neg$*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) $\in$ *lexn* $\{(a, b).\ a < b\}$ *3*
using *assms*(*1*) *M-level assms*(*2*,*3*)
proof (*induct rule*: *cdcl*$_W$*-all-induct-lev2*)
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* (*mset-cls 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* (*mset-cls 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* (*mset-cls 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* (*mset-cls 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*)
next
case (*decide L*) **note** *conf* = *this*(*1*) **and** *undef* = *this*(*2*) **and** *T* = *this*(*4*)
moreover
  have *dec*: *decide S* (*cons-trail* (*Marked L* (*backtrack-lvl S* + *1*)) (*incr-lvl S*))
    using *decide-rule decide.hyps* **by** *force*
  then have $cdcl_W$:$cdcl_W$ *S* (*cons-trail* (*Marked L* (*backtrack-lvl S* + *1*)) (*incr-lvl S*))
    using $cdcl_W$*.simps* $cdcl_W$*-o.intros* **by** *blast*
moreover
  have *lev*: $cdcl_W$*-M-level-inv* (*cons-trail* (*Marked L* (*backtrack-lvl S* + *1*)) (*incr-lvl S*))
    using $cdcl_W$ *M-level* $cdcl_W$*-consistent-inv*[*OF* $cdcl_W$] **by** *auto*
  then have *no-dup*: *no-dup* (*Marked L* (*backtrack-lvl S* + *1*) # *trail S*)
    using *undef* **unfolding** $cdcl_W$*-M-level-inv-def* **by** *auto*
  have *no-strange-atm* (*cons-trail* (*Marked L* (*backtrack-lvl S* + *1*)) (*incr-lvl S*))
    using *M-level alien calculation*(*4*) $cdcl_W$*-no-strange-atm-inv* **by** *blast*
  then have *length* (*Marked L* ((*backtrack-lvl S*) + *1*) # (*trail S*))
    ≤ *card* (*atms-of-mm* (*init-clss S*))
    using *no-dup undef*
    length-model-le-vars[*of cons-trail* (*Marked L* (*backtrack-lvl S* + *1*)) (*incr-lvl S*)]
    by *fastforce*
ultimately show *?case* **using** *conf* **by** *auto*
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*)
next
case *conflict*
then show *?case* **by** *simp*
next
case *resolve*
then show *?case* **using** *finite* **by** *simp*
next
case (*backtrack K i M1 M2 L D T*) **note** *conf* = *this*(*1*) **and** *decomp* = *this*(*3*) **and** *undef* = *this*(*7*)
and
  T = *this*(*8*) **and** *lev* = *this*(*9*)
let *?S'* = *T*
have *bt*: *backtrack S ?S'*
  using *backtrack.hyps backtrack.intros*[*of S D L K i*] **by** *auto*
have *mset-ccls D* ∉# *learned-clss S*

    using *no-relearn conf bt* **by** *auto*
  then have *card-T*:
    card (*set-mset* ({#*mset-ccls D*#} + *learned-clss S*)) = *Suc* (*card* (*set-mset* (*learned-clss S*)))
    by *simp*
  have *distinct-cdcl*$_W$*-state ?S'*
    using *bt M-level distinct-cdcl*$_W$*-state-inv no-dup other cdcl*$_W$*-o.intros cdcl*$_W$*-bj.intros* **by** *blast*
  moreover have ∀ *s*∈#*learned-clss ?S'*. ¬ *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*)) ≤ *3* ^ *card* (*atms-of-mm* (*learned-clss T*))
      by (*auto simp*: *learned-clss-less-upper-bound*)
    then have *H*: *card* (*set-mset* ({#*mset-ccls D*#} + *learned-clss S*))
      ≤ *3* ^ *card* (*atms-of-mm* ({#*mset-ccls D*#} + *learned-clss S*))
      using *T undef decomp M-level* **by** (*simp add*: *cdcl*$_W$*-M-level-inv-decomp*)
  moreover
    have *atms-of-mm* ({#*mset-ccls D*#} + *learned-clss S*) ⊆ *atms-of-mm* (*init-clss S*)
      using *alien conf* **unfolding** *no-strange-atm-def* **by** *auto*
    then have *card-f*: *card* (*atms-of-mm* ({#*mset-ccls D*#} + *learned-clss S*))
      ≤ *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* ({#*mset-ccls D*#} + *learned-clss S*))
      ≤ *3* ^ *card* (*atms-of-mm* (*init-clss S*)) **by** *simp*
  ultimately have (*3*::*nat*) ^ *card* (*atms-of-mm* (*init-clss S*))
    ≥ *card* (*set-mset* ({#*mset-ccls D*#} + *learned-clss S*))
    using *le-trans* **by** *blast*
  then show *?case* **using** *decomp undef diff-less-mono2 card-T T M-level*
    by (*auto simp*: *cdcl*$_W$*-M-level-inv-decomp*)
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*(*8*)
  then have *mset-cls C* ∈# *learned-clss S* **and** *mset-cls C* ∉# *learned-clss T*
    using *forget.hyps* **by** *auto*
  then show *?case* **using** *no-forget* **by** (*auto simp add*: *mset-leD*)
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*) ∈ *lexn* {(*a*, *b*). *a* < *b*} *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*) ∈ *lexn* {(*a*, *b*). *a* < *b*} *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* $\{(a, b).\ a < b\}$ *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 *trans-le*:
trans $\{(a, (b::nat)).\ a < b\}$
unfolding *trans-def* **by** *auto*

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* $\{(a, b).\ a < b\}$ *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* $\{(a, b).\ a < b\}$ *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* $\{a.\ case\ a\ of\ (a, b) \Rightarrow a < b\}$ *3* **by** *blast*

moreover have ($cdcl_W$*-measure* U, $cdcl_W$*-measure* T) $\in$ *lexn* $\{a.\ case\ a\ of\ (a, b) \Rightarrow a < b\}$ *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-le*] **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 $(cdcl_W\text{-}measure\ T,\ cdcl_W\text{-}measure\ S) \in lexn\ \{(a,\ b).\ a < b\}\ 3$
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*: $(cdcl_W\text{-}measure\ U,\ cdcl_W\text{-}measure\ T) \in lexn\ \{a.\ case\ a\ of\ (a,\ b) \Rightarrow a < b\}\ 3\ \vee$
$cdcl_W\text{-}measure\ U = cdcl_W\text{-}measure\ T$
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** $(cdcl_W\text{-}measure\ T,\ cdcl_W\text{-}measure\ S) \in lexn\ \{a.\ case\ a\ of\ (a,\ b) \Rightarrow a < b\}\ 3$
proof (*induction rule*:*cdcl*$_W$*-o-induct-lev2*)
case (*decide T*)
then show *?case* **using** *decide-measure-decreasing H decide.intros*[*OF decide.hyps*] **by** *blast*
next
case (*backtrack K i M1 M2 L D T*) **note** *conf = this*(*1*) **and** *decomp = this*(*3*) **and**
undef = this(*7*) **and** *T = this*(*8*)
have *bt*: *backtrack S T*
apply (*rule backtrack-rule*)
using *backtrack.hyps* **by** *auto*
then have *no-relearn*: $\forall T.\ conflicting\ S = Some\ T \longrightarrow T \notin\#\ learned\text{-}clss\ S$
using *cdcl*$_W$*-stgy-no-relearned-clause*[*of R S T*] *H conf*
unfolding *cdcl*$_W$*-all-struct-inv-def raw-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** *force*
next
case *resolve*

then show *?case* **by** *force*
qed
ultimately show *?case*
by (*metis lexn-transI transD trans-le*)
qed
qed

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* {(*a*, *b*). *a* < *b*} *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-le*, *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* (*mset-clss N*)
shows (*cdcl*$_W$*-measure S*, *cdcl*$_W$*-measure* (*init-state N*)) ∈ *lexn* {(*a*, *b*). *a* < *b*} *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* (*mset-clss N*) ∧ *cdcl*$_W$*-stgy*$^{++}$ (*init-state N*) *S*}
apply (*rule wf-wf-if-measure'-notation2*[*of lexn* {(*a*, *b*). *a* < *b*} *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 -*
λ*S*. *card* (*atms-of-mm* (*init-clss S*))+*1*
λ*S*. *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
theory *DPLL-CDCL-W-Implementation*
imports *Partial-Annotated-Clausal-Logic*
begin

20 Simple Implementation of the DPLL and CDCL

20.1 Common Rules

20.1.1 Propagation

The following theorem holds:

lemma *lits-of-l-unfold*[*iff*]:
$(\forall c \in set\ C.\ -c \in lits\text{-}of\text{-}l\ Ms) \longleftrightarrow Ms \models as\ CNot\ (mset\ C)$
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* $\Rightarrow$ $('a, 'b, 'c)$ *marked-lit list* $\Rightarrow$ $'a$ *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* $\Rightarrow$ $('a, 'b, 'c)$ *marked-lit list*
$\Rightarrow$ $'a$ *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*: $\bigwedge a.$ ($\forall c \in set$ (*remove1 a l*). $-$ *c* $\in$ *lits-of-l M*) $\longleftrightarrow$ *M* $\models$*as CNot* (*mset l* $-$ {#*a*#})
using *lits-of-l-unfold*[*of remove1 - l*, *of - M*] **by** *simp*
thus *?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* $\notin$ *atm-of* ‘ *lits-of-l M*] *of* [] $\Rightarrow$ *None*
| [*a*] $\Rightarrow$ *if M* $\models$*as CNot* (*mset l* $-$ {#*a*#}) *then Some a else None*
| *a* # *ab* # *xa* $\Rightarrow$ *Map.empty xa*) = *Some a*
using *assms* **unfolding** *is-unit-clause-def* **.**
hence *a* $\in$ *set* [*a*←*l* . *atm-of a* $\notin$ *atm-of* ‘ *lits-of-l M*]
apply (*cases* [*a*←*l* . *atm-of a* $\notin$ *atm-of* ‘ *lits-of-l M*])
apply *simp*
apply (*rename-tac aa list*; *case-tac list*) **by** (*auto split*: *if-split-asm*)
hence *atm-of a* $\notin$ *atm-of* ‘ *lits-of-l M* **by** *auto*
thus *?thesis*

by (*simp add*: *Marked-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* $\Longrightarrow$ *M* $\models$*as CNot* (*mset l* − {#*a*#})
unfolding *is-unit-clause-def*
proof −
assume (*case* [*a*←*l* . *atm-of a* $\notin$ *atm-of* ' *lits-of-l M*] *of* [] ⇒ *None*
| [*a*] ⇒ *if M* $\models$*as CNot* (*mset l* − {#*a*#}) *then Some a else None*
| *a* # *ab* # *xa* ⇒ *Map.empty xa*) = *Some a*
thus *?thesis*
apply (*cases* [*a*←*l* . *atm-of a* $\notin$ *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* $\Longrightarrow$ *a* ∈ *set l*
unfolding *is-unit-clause-def*
proof −
assume (*case* [*a*←*l* . *atm-of a* $\notin$ *atm-of* ' *lits-of-l M*] *of* [] ⇒ *None*
| [*a*] ⇒ *if M* $\models$*as CNot* (*mset l* − {#*a*#}) *then Some a else None*
| *a* # *ab* # *xa* ⇒ *Map.empty xa*) = *Some a*
thus *a* ∈ *set l*
by (*cases* [*a*←*l* . *atm-of a* $\notin$ *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*

20.1.2 Unit propagation for all clauses

Finding the first clause to propagate

fun *find-first-unit-clause* :: *'a literal list list* ⇒ (*'a*, *'b*, *'c*) *marked-lit list*
⇒ (*'a literal* × *'a literal list*) *option* **where**
find-first-unit-clause (*a* # *l*) *M* =
(*case is-unit-clause a M of*
None ⇒ *find-first-unit-clause l M*
| *Some L* ⇒ *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* ∈ *set l* ∧ *M* $\models$*as CNot* (*mset c* − {#*a*#}) ∧ *undefined-lit M a* ∧ *a* ∈ *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* ∈ *set c*
shows *is-unit-clause c M* ≠ *None*

proof −
  have $[a{\leftarrow}c\ .\ \textit{atm-of}\ a \notin \textit{atm-of}\ \text{`}\ \textit{lits-of-l}\ M] = [a]$
    using *assms*
    proof (*induction c*)
      case *Nil* **thus** *?case* **by** *simp*
    next
      case (*Cons ac c*)
      show *?case*
        proof (*cases a = ac*)
          case *True*
          thus *?thesis* **using** *Cons*
            by (*auto simp del*: *lits-of-l-unfold*
                simp add: *lits-of-l-unfold*[*symmetric*] *Marked-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*
          hence *T*: $\textit{mset}\ c + \{\#ac\#\} - \{\#a\#\} = \textit{mset}\ c - \{\#a\#\} + \{\#ac\#\}$
            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
  thus *?thesis*
    using *M* **unfolding** *is-unit-clause-def* **by** *auto*
qed

lemma *find-first-unit-clause-none*:
  $\textit{distinct}\ c \Longrightarrow c \in \textit{set}\ l \Longrightarrow\ M \models\!\textit{as}\ \textit{CNot}\ (\textit{mset}\ c - \{\#a\#\}) \Longrightarrow \textit{undefined-lit}\ M\ a \Longrightarrow a \in \textit{set}\ c$
  $\Longrightarrow \textit{find-first-unit-clause}\ l\ M \neq \textit{None}$
  by (*induction l*)
    (*auto split*: *option.split simp add*: *propagate-is-unit-clause-not-None*)

20.1.3 Decide

fun *find-first-unused-var* :: *'a literal list list* ⇒ *'a literal set* ⇒ *'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 ⇒ *find-first-unused-var l M*
  | *Some a* ⇒ *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 $\neg(\forall a \in set\ l.\ atm\text{-}of\ `\ set\ a \subseteq atm\text{-}of\ `\ M)$
proof −
  have *find-first-unused-var l M* $\neq$ *None*
    using *assms* **by** (*cases find-first-unused-var l M*) *auto*
  thus $\neg(\forall a \in set\ l.\ atm\text{-}of\ `\ set\ a \subseteq atm\text{-}of\ `\ M)$ **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*] *Marked-Propagated-in-iff-in-lits-of-l*
  by *blast*

end
theory *DPLL-W-Implementation*
imports *DPLL-CDCL-W-Implementation DPLL-W ~~/src/HOL/Library/Code-Target-Numeral*
begin

20.2 Simple Implementation of DPLL

20.2.1 Combining the propagate and decide: a DPLL step

definition *DPLL-step* :: *int* $dpll_W$*-marked-lits* × *int literal list list*
  ⇒ *int* $dpll_W$*-marked-lits* × *int literal list list* **where**
DPLL-step = (λ(*Ms*, *N*).
  (*case find-first-unit-clause N Ms of*
    Some (*L*, -) ⇒ (*Propagated L* () # *Ms*, *N*)
  | - ⇒
    if ∃ *C* ∈ *set N*. (∀ *c* ∈ *set C*. −*c* ∈ *lits-of-l Ms*)
    then
      (*case backtrack-split Ms of*
        (-, *L* # *M*) ⇒ (*Propagated* (− (*lit-of L*)) () # *M*, *N*)
      | (-, -) ⇒ (*Ms*, *N*)
)
    else
    (*case find-first-unused-var N* (*lits-of-l Ms*) *of*
        Some a ⇒ (*Marked a* () # *Ms*, *N*)
      | *None* ⇒ (*Ms*, *N*))))

Example of propagation:

value *DPLL-step* ([*Marked* (*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* ≡ λ(*Ms*::(*int*, *unit*, *unit*) *marked-lit list*)
                    (*N*:: *int literal list list*). (*Ms*, *mset* (*map mset N*))
abbreviation *toS′* ≡ λ(*Ms*::(*int*, *unit*, *unit*) *marked-lit list*,
                    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*: $(Ms', N') = DPLL\text{-}step\ (Ms, N)$
  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*: $find\text{-}first\text{-}unit\text{-}clause\ N\ Ms = Some\ (L, E)$
    hence $Ms'N$: $(Ms', N') = (Propagated\ L\ ()\ \#\ Ms, N)$
      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*)
    hence *?thesis* **using** $Ms'N$ **by** *auto*
  }
  moreover
  { assume *unit*: $find\text{-}first\text{-}unit\text{-}clause\ N\ Ms = None$
    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*: $backtrack\text{-}split\ Ms\ = (M', L\ \#\ M)$
      using *step exC neq* **unfolding** *DPLL-step-def prod.case unit*
      by (*cases backtrack-split Ms*, *rename-tac b*, *case-tac b*) *auto*
    hence *is-marked L* **using** *backtrack-split-snd-hd-marked*[*of Ms*] **by** *auto*
    have *1*: $dpll_W\ (Ms, mset\ (map\ mset\ N))$
          $(Propagated\ (-\ lit\text{-}of\ L)\ ()\ \#\ M, snd\ (Ms, mset\ (map\ mset\ N)))$
      apply (*rule* $dpll_W$*.backtrack*[*OF* - ‹*is-marked L*›, *of*])
      using *C Ms bt* **by** *auto*
    moreover have $(Ms', N') = (Propagated\ (-\ (lit\text{-}of\ L))\ ()\ \#\ M, N)$
      using *step exC* **unfolding** *DPLL-step-def bt prod.case unit* **by** *auto*
    ultimately have *?thesis* **by** *auto*
  }
  moreover
  { assume *unit*: $find\text{-}first\text{-}unit\text{-}clause\ N\ Ms = None$
    assume *exC*: $\neg\ (\exists\, C \in set\ N.\ Ms \models as\ CNot\ (mset\ C))$
    obtain *L* **where** *unused*: $find\text{-}first\text{-}unused\text{-}var\ N\ (lits\text{-}of\text{-}l\ Ms) = Some\ L$
      using *step exC neq* **unfolding** *DPLL-step-def prod.case unit*
      by (*cases find-first-unused-var N* (*lits-of-l Ms*)) *auto*
    have $dpll_W\ (Ms, mset\ (map\ mset\ N))$
        $(Marked\ L\ ()\ \#\ fst\ (Ms, mset\ (map\ mset\ N)), snd\ (Ms, mset\ (map\ mset\ N)))$
      apply (*rule* $dpll_W$*.decided*[*of ?S L*])
      using *find-first-unused-var-Some*[*OF unused*]
      by (*auto simp add*: *Marked-Propagated-in-iff-in-lits-of-l atms-of-ms-def*)
    moreover have $(Ms', N') = (Marked\ L\ ()\ \#\ Ms, N)$
      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*)
    hence *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*) = []
      thus *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*
      hence *Ms*: *Ms* = *Propagated* (− *lit-of aa*) () # *list* **using** *Ms* **by** *auto*
      have *is-marked aa* **using** *backtrack-split-snd-hd-marked*[*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*
      thus *snd* (*backtrack-split Ms*) = [] **by** *blast*
    qed

    hence *?thesis*
      using *n backtrack-snd-empty-not-marked*[*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*))
    hence *find-first-unused-var N* (*lits-of-l Ms*) = *None*
      using *step* **unfolding** *DPLL-step-def* **by** (*simp add*: *unit split*: *option.splits*)
    hence *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*))
        hence ¬*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
    hence *?thesis* **unfolding** *conclusive-dpll*$_W$*-state-def* **by** *blast*
  }
  ultimately show *?thesis* **by** *blast*
qed

20.2.2 Adding invariants

Invariant tested in the function **function** *DPLL-ci* :: *int* $dpll_W$*-marked-lits* ⇒ *int literal list list*
⇒ *int* $dpll_W$*-marked-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* $dpll_W$*-marked-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*)
thus ((*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* $dpll_W$*-marked-lits* ⇒ *int literal list list* ⇒
int $dpll_W$*-marked-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*)
hence *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*
hence *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 $(Ms', N) = (Ms, N)$
hence *?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 $\neg dpll_W$*-all-inv* (*toS Ms N*)
hence (*Ms*, *N*) = (*Ms'*, *N*) **using** *step* **by** *auto*
hence *?case* **by** *auto*
}
moreover
{ assume $dpll_W$*-all-inv* (*toS Ms N*)
and $(S_1, S_2) = (Ms, N)$
hence *?case* **using** *S step* **by** *auto*
}
moreover
{ assume $dpll_W$*-all-inv* (*toS Ms N*)
and $(S_1, S_2) \neq (Ms, N)$
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*) ⇒
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*
hence (*if* (S_1, S_2) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci* S_1 *N*) = *DPLL-ci Ms N*
by *fastforce*
thus *?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*
  thus *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*
      hence *DPLL-ci Ms′ N* = (*Ms*, *N*) **using** *step inv st Ms′N*[*symmetric*] **by** *fastforce*
      hence *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*) ≠ (*Ms*,
N)›
          prod.sel(*2*) *rtranclp-into-tranclp2 snd-DPLL-step*)
      thus *False* **using** *dpll*$_W$*-all-inv-dpll*$_W$*-tranclp-irrefl inv* **by** *auto*
    qed
  thus *?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*)
    hence *?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*›)
    hence *?thesis*
      using *IH that* **by** *fastforce*
  }
  moreover {
    assume *n*: (*S*, *N*) = (*Ms*, *N*)
    hence *?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*)
    hence *?case* **using** *step* **by** *auto*
  }
  moreover {
    assume $dpll_W$*-all-inv* (*toS Ms N*)
    and (S_1, *N*) = (*Ms*, *N*)
    hence *?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*
        hence (*if* (S_1, *N*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-ci* S_1 *N*) = *DPLL-ci Ms N*
          by *fastforce*
        thus *?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*, *unit*) *marked-lit list* **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*
  hence *star*: $dpll_W{}^{**}$ (*toS Ms N*) (*toS Ms' N*) **unfolding** *MsN* **using** *DPLL-ci-*$dpll_W$*-rtranclp* **by** *blast*
  hence *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* =

$\{(M::(int,\ unit,\ unit)\ marked\text{-}lit\ list,\ N::int\ literal\ list\ list).$
$dpll_W\text{-}all\text{-}inv\ (toS\ M\ N)\}$
morphisms *rough-state-of state-of*
proof
show $([],[]) \in \{(M,\ N).\ dpll_W\text{-}all\text{-}inv\ (toS\ M\ N)\}$ **by** $(auto\ simp\ add{:}\ dpll_W\text{-}all\text{-}inv\text{-}def)$
qed

lemma
DPLL-part-dom $([],\ N)$
using *assms* $dpll_W\text{-}all\text{-}inv\text{-}implieS\text{-}2\text{-}eq3\text{-}and\text{-}dom[of\ []\ N]$ **by** $(simp\ add{:}\ dpll_W\text{-}all\text{-}inv\text{-}def)$

Some type classes **instantiation** $dpll_W\text{-}state :: equal$
begin
definition $equal\text{-}dpll_W\text{-}state :: dpll_W\text{-}state \Rightarrow dpll_W\text{-}state \Rightarrow bool$ **where**
$equal\text{-}dpll_W\text{-}state\ S\ S' = (rough\text{-}state\text{-}of\ S = rough\text{-}state\text{-}of\ S')$
instance
by $standard\ (simp\ add{:}\ rough\text{-}state\text{-}of\text{-}inject\ equal\text{-}dpll_W\text{-}state\text{-}def)$
end

DPLL **definition** $DPLL\text{-}step' :: dpll_W\text{-}state \Rightarrow dpll_W\text{-}state$ **where**
$DPLL\text{-}step'\ S = state\text{-}of\ (DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S))$

declare *rough-state-of-inverse*[*simp*]

lemma $DPLL\text{-}step\text{-}dpll_W\text{-}conc\text{-}inv$:
$DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S) \in \{(M,\ N).\ dpll_W\text{-}all\text{-}inv\ (toS\ M\ N)\}$
by $(smt\ DPLL\text{-}ci.simps\ DPLL\text{-}ci\text{-}dpll_W\text{-}rtranclp\ case\text{-}prodE\ case\text{-}prodI2\ rough\text{-}state\text{-}of$
$mem\text{-}Collect\text{-}eq\ old.prod.case\ prod.sel(2)\ rtranclp\text{-}dpll_W\text{-}all\text{-}inv\ snd\text{-}DPLL\text{-}step)$

lemma *rough-state-of-DPLL-step′-DPLL-step*[*simp*]:
$rough\text{-}state\text{-}of\ (DPLL\text{-}step'\ S) = DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S)$
using $DPLL\text{-}step\text{-}dpll_W\text{-}conc\text{-}inv\ DPLL\text{-}step'\text{-}def\ state\text{-}of\text{-}inverse$ **by** *auto*

function $DPLL\text{-}tot:: dpll_W\text{-}state \Rightarrow dpll_W\text{-}state$ **where**
$DPLL\text{-}tot\ S =$
$(let\ S' = DPLL\text{-}step'\ S\ in$
$if\ S' = S\ then\ S\ else\ DPLL\text{-}tot\ S')$
by *fast+*
termination
proof $(relation\ \{(T',\ T).$
$(rough\text{-}state\text{-}of\ T',\ rough\text{-}state\text{-}of\ T)$
$\in \{(S',\ S).\ (toS'\ S',\ toS'\ S)$
$\in \{(S',\ S).\ dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}\}\})$
show $wf\ \{(b,\ a).$
$(rough\text{-}state\text{-}of\ b,\ rough\text{-}state\text{-}of\ a)$
$\in \{(b,\ a).\ (toS'\ b,\ toS'\ a)$
$\in \{(b,\ a).\ dpll_W\text{-}all\text{-}inv\ a \wedge dpll_W\ a\ b\}\}\}$
using $wf\text{-}if\text{-}measure\text{-}f[OF\ wf\text{-}if\text{-}measure\text{-}f[OF\ dpll_W\text{-}wf,\ of\ toS'],\ of\ rough\text{-}state\text{-}of]$ **.**
next
fix $S\ x$
assume $x{:}\ x = DPLL\text{-}step'\ S$
and $x \neq S$
have $dpll_W\text{-}all\text{-}inv\ (case\ rough\text{-}state\text{-}of\ S\ of\ (Ms,\ N) \Rightarrow (Ms,\ mset\ (map\ mset\ N)))$
by $(metis\ (no\text{-}types,\ lifting)\ case\text{-}prodE\ mem\text{-}Collect\text{-}eq\ old.prod.case\ rough\text{-}state\text{-}of)$
moreover have $dpll_W\ (case\ rough\text{-}state\text{-}of\ S\ of\ (Ms,\ N) \Rightarrow (Ms,\ mset\ (map\ mset\ N)))$

(*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+*
thus *?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* λ*S*. *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*
hence *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*)
thus *?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

20.2.3 Code export

A conversion to *DPLL-W-Implementation.dpll*$_W$*-state* **definition** *Con* :: (*int*, *unit*, *unit*) *marked-lit list* × *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* $(\forall A \in$ *set* N. $(\exists a{\in}$*set* A. $a \in$ *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

notation *image-mset* (**infixr** *'# 90*)

type-synonym *'a* $cdcl_W$*-mark* = *'a literal list*
type-synonym $cdcl_W$*-marked-level* = *nat*

type-synonym *'v* $cdcl_W$*-marked-lit* = (*'v*, $cdcl_W$*-marked-level*, *'v* $cdcl_W$*-mark*) *marked-lit*
type-synonym *'v* $cdcl_W$*-marked-lits* = (*'v*, $cdcl_W$*-marked-level*, *'v* $cdcl_W$*-mark*) *marked-lits*
type-synonym *'v* $cdcl_W$*-state* =
'v $cdcl_W$*-marked-lits* × *'v literal list list* × *'v literal list list* × *nat* ×
'v literal list option

abbreviation *raw-trail* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* **where**
raw-trail ≡ (λ(*M*, -). *M*)

abbreviation *raw-cons-trail* :: *'a* ⇒ *'a list* × *'b* × *'c* × *'d* × *'e* ⇒ *'a list* × *'b* × *'c* × *'d* × *'e* **where**
raw-cons-trail ≡ (λ*L* (*M*, *S*). (*L*#*M*, *S*))

abbreviation *raw-tl-trail* :: *'a list* × *'b* × *'c* × *'d* × *'e* ⇒ *'a list* × *'b* × *'c* × *'d* × *'e* **where**
raw-tl-trail ≡ (λ(*M*, *S*). (*tl M*, *S*))

abbreviation *raw-init-clss* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'b* **where**
raw-init-clss ≡ λ(*M*, *N*, -). *N*

abbreviation *raw-learned-clss* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'c* **where**
raw-learned-clss ≡ λ(*M*, *N*, *U*, -). *U*

abbreviation *raw-backtrack-lvl* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'d* **where**
raw-backtrack-lvl ≡ λ(*M*, *N*, *U*, *k*, -). *k*

abbreviation *raw-update-backtrack-lvl* :: *'d* ⇒ *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e* **where**
raw-update-backtrack-lvl ≡ λ*k* (*M*, *N*, *U*, -, *S*). (*M*, *N*, *U*, *k*, *S*)

abbreviation *raw-conflicting* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'e* **where**
raw-conflicting ≡ λ(*M*, *N*, *U*, *k*, *D*). *D*

abbreviation *raw-update-conflicting* :: *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e*

where
raw-update-conflicting $\equiv \lambda S\ (M,\ N,\ U,\ k,\ \text{-}).\ \ (M,\ N,\ U,\ k,\ S)$

abbreviation *raw-add-learned-cls* **where**
raw-add-learned-cls $\equiv \lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ \{\#C\#\} + U,\ S)$

abbreviation *raw-remove-cls* **where**
raw-remove-cls $\equiv \lambda C\ (M,\ N,\ U,\ S).\ (M,\ \textit{removeAll-mset}\ C\ N,\ \textit{removeAll-mset}\ C\ U,\ S)$

type-synonym $'v\ \textit{cdcl}_W\textit{-state-inv-st} = ('v,\ \textit{nat},\ 'v\ \textit{literal list})\ \textit{marked-lit list} \times$
$'v\ \textit{literal list list} \times\ 'v\ \textit{literal list list} \times \textit{nat} \times\ 'v\ \textit{literal list option}$

abbreviation $\textit{raw-S0-cdcl}_W\ N \equiv (([],\ N,\ [],\ 0,\ \textit{None}) {::}\ 'v\ \textit{cdcl}_W\textit{-state-inv-st})$

fun *mmset-of-mlit'* :: $('v,\ \textit{nat},\ 'v\ \textit{literal list})\ \textit{marked-lit} \Rightarrow ('v,\ \textit{nat},\ 'v\ \textit{clause})\ \textit{marked-lit}$
where
mmset-of-mlit' $(\textit{Propagated}\ L\ C) = \textit{Propagated}\ L\ (\textit{mset}\ C)$ |
mmset-of-mlit' $(\textit{Marked}\ L\ i) = \textit{Marked}\ L\ i$

lemma *lit-of-mmset-of-mlit'*[*simp*]:
lit-of (*mmset-of-mlit' xa*) = *lit-of xa*
by (*induction xa*) *auto*

abbreviation *trail* **where**
trail S $\equiv$ *map mmset-of-mlit'* (*raw-trail S*)

abbreviation *clauses-of-l* **where**
clauses-of-l $\equiv \lambda L.\ \textit{mset}\ (\textit{map mset}\ L)$

global-interpretation $\textit{state}_W\textit{-ops}$
mset:: $'v\ \textit{literal list} \Rightarrow\ 'v\ \textit{clause}$
op # remove1

clauses-of-l op @ $\lambda L\ C.\ L \in \textit{set}\ C$ *op #* $\lambda C.\ \textit{remove1-cond}\ (\lambda L.\ \textit{mset}\ L = \textit{mset}\ C)$

mset $\lambda xs\ ys.\ \textit{case-prod append}\ (\textit{fold}\ (\lambda x\ (ys,\ zs).\ (\textit{remove1}\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ []))$
op # remove1

id id

$\lambda(M,\ \text{-}).\ \textit{map mmset-of-mlit'}\ M\ \lambda(M,\ \text{-}).\ \textit{hd}\ M$
$\lambda(\text{-},\ N,\ \text{-}).\ N$
$\lambda(\text{-},\ \text{-},\ U,\ \text{-}).\ U$
$\lambda(\text{-},\ \text{-},\ \text{-},\ k,\ \text{-}).\ k$
$\lambda(\text{-},\ \text{-},\ \text{-},\ \text{-},\ C).\ C$

$\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
$\lambda(M,\ S).\ (\textit{tl}\ M,\ S)$
$\lambda C\ (M,\ N,\ S).\ (M,\ C\ \#\ N,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ C\ \#\ U,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ \textit{filter}\ (\lambda L.\ \textit{mset}\ L \neq \textit{mset}\ C)\ N,\ \textit{filter}\ (\lambda L.\ \textit{mset}\ L \neq \textit{mset}\ C)\ U,\ S)$
$\lambda(k{::}\textit{nat})\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
$\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
$\lambda N.\ ([],\ N,\ [],\ 0,\ \textit{None})$
$\lambda(\text{-},\ N,\ U,\ \text{-}).\ ([],\ N,\ U,\ 0,\ \textit{None})$

apply *unfold-locales* **by** (*auto simp*: *hd-map comp-def map-tl ac-simps union-mset-list mset-map-mset-remove1-cond ex-mset*)

lemma *mmset-of-mlit′-mmset-of-mlit*: *mmset-of-mlit′ l* = *mmset-of-mlit l*
apply (*induct l*)
apply *auto*
done

lemma *clauses-of-l-filter-removeAll*:
clauses-of-l [*L*←*a* . *mset L* ≠ *mset C*] = *mset* (*removeAll* (*mset C*) (*map mset a*))
by (*induct a*) *auto*

interpretation $state_W$
mset::′*v literal list* ⇒ ′*v clause*
op # *remove1*

clauses-of-l op @ λ*L C*. *L* ∈ *set C op* # λ*C*. *remove1-cond* (λ*L*. *mset L* = *mset C*)

mset λ*xs ys*. *case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))
op # *remove1*

id id

λ(*M*, -). *map mmset-of-mlit′ M* λ(*M*, -). *hd M*
λ(-, *N*, -). *N*
λ(-, -, *U*, -). *U*
λ(-, -, -, *k*, -). *k*
λ(-, -, -, -, *C*). *C*

λ*L* (*M*, *S*). (*L* # *M*, *S*)
λ(*M*, *S*). (*tl M*, *S*)
λ*C* (*M*, *N*, *S*). (*M*, *C* # *N*, *S*)
λ*C* (*M*, *N*, *U*, *S*). (*M*, *N*, *C* # *U*, *S*)
λ*C* (*M*, *N*, *U*, *S*). (*M*, *filter* (λ*L*. *mset L* ≠ *mset C*) *N*, *filter* (λ*L*. *mset L* ≠ *mset C*) *U*, *S*)
λ(*k*::*nat*) (*M*, *N*, *U*, -, *D*). (*M*, *N*, *U*, *k*, *D*)
λ*D* (*M*, *N*, *U*, *k*, -). (*M*, *N*, *U*, *k*, *D*)
λ*N*. ([], *N*, [], *0*, *None*)
λ(-, *N*, *U*, -). ([], *N*, *U*, *0*, *None*)
apply *unfold-locales*
apply (*rename-tac S*, *case-tac S*)
by (*auto simp*: *hd-map comp-def map-tl ac-simps clauses-of-l-filter-removeAll mmset-of-mlit′-mmset-of-mlit*)

global-interpretation $conflict\text{-}driven\text{-}clause\text{-}learning_W$
mset::′*v literal list* ⇒ ′*v clause*
op # *remove1*

clauses-of-l op @ λ*L C*. *L* ∈ *set C op* # λ*C*. *remove1-cond* (λ*L*. *mset L* = *mset C*)

mset λ*xs ys*. *case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))
op # *remove1*

id id

λ(*M*, -). *map mmset-of-mlit′ M* λ(*M*, -). *hd M*

λ(-, N, -). N
λ(-, -, U, -). U
λ(-, -, -, k, -). k
λ(-, -, -, -, C). C

λL (M, S). (L # M, S)
λ(M, S). (*tl* M, S)
λC (M, N, S). (M, C # N, S)
λC (M, N, U, S). (M, N, C # U, S)
λC (M, N, U, S). (M, *filter* (λL. *mset* $L \neq$ *mset* C) N, *filter* (λL. *mset* $L \neq$ *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*)
λ(-, N, U, -). ([], N, U, *0*, *None*)
by *intro-locales*

declare *state-simp*[*simp del*] *raw-clauses-def*[*simp*] *state-eq-def*[*simp*]
notation *state-eq* (**infix** $\sim$ *50*)
term *reduce-trail-to*

lemma *reduce-trail-to-map*[*simp*]:
reduce-trail-to (*map f M1*) = *reduce-trail-to M1*
by (*rule ext*) (*auto intro*: *reduce-trail-to-length*)

20.3 CDCL Implementation

20.3.1 Definition of the rules

Types **lemma** *true-clss-remdups*[*simp*]:
$I \models s$ (*mset* $\circ$ *remdups*) ' $N \longleftrightarrow I \models s$ *mset* ' N
by (*simp add*: *true-clss-def*)

lemma *satisfiable-mset-remdups*[*simp*]:
satisfiable ((*mset* $\circ$ *remdups*) ' N) $\longleftrightarrow$ *satisfiable* (*mset* ' N)
unfolding *satisfiable-carac*[*symmetric*] **by** *simp*

We need some functions to convert between our abstract state *nat* $cdcl_W$*-state* and the concrete state *'v* $cdcl_W$*-state-inv-st*.

abbreviation *convertC* :: *'a list option* $\Rightarrow$ *'a multiset option* **where**
convertC $\equiv$ *map-option mset*

lemma *convert-Propagated*[*elim*!]:
mmset-of-mlit' z = *Propagated* L C $\Longrightarrow$ ($\exists$ C'. z = *Propagated* L C' $\wedge$ C = *mset* C')
by (*cases z*) *auto*

lemma *get-rev-level-map-convert*:
get-rev-level (*map mmset-of-mlit'* M) n x = *get-rev-level* M n x
by (*induction M arbitrary*: n *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-map-convert*[*simp*]:
get-level (*map mmset-of-mlit'* M) = *get-level* M
using *get-rev-level-map-convert*[*of rev M*] **by** (*simp add*: *rev-map*)

lemma *get-rev-level-map-mmsetof-mlit*[*simp*]:
get-rev-level (*map mmset-of-mlit* M) = *get-rev-level* M
by (*induction M rule*: *marked-lit-list-induct*) (*auto intro*!: *ext*)

lemma *get-level-map-mmsetof-mlit*[*simp*]:
get-level (*map mmset-of-mlit M*) = *get-level M*
using *get-rev-level-map-mmsetof-mlit*[*of rev M*] **unfolding** *rev-map* **by** *simp*

lemma *get-maximum-level-map-convert*[*simp*]:
get-maximum-level (*map mmset-of-mlit′ M*) *D* = *get-maximum-level M D*
by (*induction D*) (*auto simp add*: *get-maximum-level-plus*)

lemma *get-all-levels-of-marked-map-convert*[*simp*]:
get-all-levels-of-marked (*map mmset-of-mlit′ M*) = (*get-all-levels-of-marked M*)
by (*induction M rule*: *marked-lit-list-induct*) *auto*

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 simp add*:)
qed
qed

Definition an abstract type

typedef *′v* $cdcl_W$*-state-inv* = {*S*::*′v* $cdcl_W$*-state-inv-st*. $cdcl_W$*-all-struct-inv S*}
morphisms *rough-state-of state-of*
proof
show ([],[], [], *0*, *None*) ∈ {*S*. $cdcl_W$*-all-struct-inv 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 mmset-of-mlit' M*) = *lits-of-l M*
  by (*induction M rule*: *marked-lit-list-induct*) *simp-all*

lemma *undefined-lit-map-convert*[*iff*]:
  undefined-lit (*map mmset-of-mlit' M*) *L* ⟷ *undefined-lit M L*
  by (*auto simp add*: *defined-lit-map image-image mmset-of-mlit'-mmset-of-mlit*)

lemma *true-annot-map-convert*[*simp*]: *map mmset-of-mlit' M* ⊨*a N* ⟷ *M* ⊨*a N*
  by (*induction M rule*: *marked-lit-list-induct*) (*simp-all add*: *true-annot-def*
    mmset-of-mlit'-mmset-of-mlit lits-of-def)

lemma *true-annots-map-convert*[*simp*]: *map mmset-of-mlit' 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* (*M*, *N*, *U*, *k*, *None*) (*Propagated L C* # *M*, *N*, *U*, *k*, *None*)
  using *assms*
  by (*auto dest*!: *find-first-unit-clause-some intro*!: *propagate-rule*)

20.3.2 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 S* (*do-propagate-step S*)
  apply (*cases S*, *cases conflicting S*)
  using *find-first-unit-clause-some-is-propagate*[*of raw-init-clss S raw-learned-clss S*]
  by (*auto simp add*: *do-propagate-step-def split*: *option.splits*)

lemma *do-propagate-step-option*[*simp*]:
  conflicting S ≠ *None* ⟹ *do-propagate-step S* = *S*
  unfolding *do-propagate-step-def* **by** (*cases S*, *cases conflicting S*) *auto*
thm *prod-cases*

lemma *do-propagate-step-no-step*:
  assumes *dist*: ∀ *c*∈*set* (*raw-clauses S*). *distinct c* **and**
  prop-step: *do-propagate-step S* = *S*
  shows *no-step propagate S*
proof (*standard*, *standard*)
  fix *T*
  assume *propagate S T*

then obtain *C L* **where**
toSS: *conflicting S = None* **and**
C: *C ∈ set (raw-clauses S)* **and**
L: *L ∈ set C* **and**
MC: *raw-trail S ⊨as CNot (mset (remove1 L C))* **and**
T: *T ∼ raw-cons-trail (Propagated L C) S* **and**
undef: *undefined-lit (raw-trail S) L*
apply (*cases S rule*: *prod-cases5*)
by (*elim propagateE*) *simp*
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 conflicting S*) *simp-all*

have *find-first-unit-clause (?N @ ?U) ?M ≠ None*
apply (*rule dist find-first-unit-clause-none*[*of C ?N @ ?U ?M L, OF -*])
using *C dist* **apply** *auto*[]
using *C* **apply** *auto*[*1*]
using *MC* **apply** *auto*[*1*]
using *undef* **apply** *auto*[*1*]
using *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 (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 S (do-conflict-step S)

apply (*cases S*, *cases conflicting S*)
unfolding *conflict.simps do-conflict-step-def*
by (*auto dest*!:*find-conflict-Some split*: *option.splits simp*: *state-eq-def*)

lemma *do-conflict-step-no-step*:
do-conflict-step S = *S* $\Longrightarrow$ *no-step conflict S*
apply (*cases S*, *cases 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.split elim*: *conflictE*)

lemma *do-conflict-step-option*[*simp*]:
conflicting S $\neq$ *None* $\Longrightarrow$ *do-conflict-step S* = *S*
unfolding *do-conflict-step-def* **by** (*cases S*, *cases conflicting S*) *auto*

lemma *do-conflict-step-conflicting*[*dest*]:
do-conflict-step S $\neq$ *S* $\Longrightarrow$ *conflicting* (*do-conflict-step S*) $\neq$ *None*
unfolding *do-conflict-step-def* **by** (*cases S*, *cases 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* $\neq$ *S*
shows *cdcl*$_W$*-cp S* (*do-cp-step S*)
proof −
show *?thesis*
proof (*cases do-conflict-step S* $\neq$ *S*)
case *True*
then have *do-propagate-step* (*do-conflict-step S*) = *do-conflict-step S*
by *auto*
then show *?thesis*
by (*auto simp add*: *do-conflict-step do-conflict-step-conflicting do-cp-step-def True*)
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* = *S*
let *?T* = (*do-propagate-step S*)
let *?U* = (*do-conflict-step* (*do-propagate-step S*))
have *propa*: *propagate S ?T* **using** *False do-propgate-step* **by** *blast*
moreover have *ns*: *no-step conflict 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 $\Longrightarrow$ *do-conflict-step S = S* $\wedge$ *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* $\longleftrightarrow$ *no-step propagate S* $\wedge$ *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**
$\forall$ *c* $\in$ *set* (*raw-init-clss S* @ *raw-learned-clss S*). *distinct c*
shows *no-step cdcl*$_W$*-cp S*
unfolding *no-cdcl*$_W$*-cp-iff-no-propagate-no-conflict*
using *assms* **apply** (*cases S*, *cases 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'* $\Longrightarrow$ *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* (*rough-state-of S*)
using *rough-state-of* **by** *auto*

lemma [*simp*]: *cdcl*$_W$*-all-struct-inv S* $\Longrightarrow$ *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* (*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* $\Rightarrow$ *'v cdcl*$_W$*-state-inv-st* **where**
do-skip-step (*Propagated L C # Ls*,*N*,*U*,*k*, *Some D*) =
(*if* −*L* $\notin$ *set D* $\wedge$ *D* $\neq$ []
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 S* (*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 S*

by (*induction S rule*: *do-skip-step.induct*)
(*auto simp add*: *other split*: *if-split-asm elim*!: *skipE*)

lemma *do-skip-step-trail-is-None*[*iff*]:
do-skip-step S = (*a*, *b*, *c*, *d*, *None*) ⟷ *S* = (*a*, *b*, *c*, *d*, *None*)
by (*cases S rule*: *do-skip-step.cases*) *auto*

Resolve **fun** *maximum-level-code*:: ′*a literal list* ⇒ (′*a*, *nat*, ′*a literal list*) *marked-lit list* ⇒ *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* ⇒ ′*v* $cdcl_W$*-state-inv-st* **where**
do-resolve-step (*Propagated L C* # *Ls*, *N*, *U*, *k*, *Some D*) =
(*if* −*L* ∈ *set D* ∧ *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 S* ⟹ *do-resolve-step S* ≠ *S*
⟹ *resolve S* (*do-resolve-step S*)
proof (*induction S rule*: *do-resolve-step.induct*)
case (*1 L C M N U k D*)
then have
LD: − *L* ∈ *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* (*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 *LC*: *L* ∈ *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* ∈ *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* ∈ *set C*› **by** (*auto simp add*: *multiset-eq-iff*)
have *max*: *get-maximum-level* (*Propagated L* (*C*′ + {#*L*#}) # *map mmset-of-mlit*′ *M*) *D*′ = *k*
using *M*[*simplified*] **unfolding** *maximum-level-code-eq-get-maximum-level C*[*symmetric*] *CL*
by (*metis D D*′*L get-maximum-level-map-convert list.simps*(*9*) *mmset-of-mlit*′*.simps*(*1*))
have *distinct-mset* (*mset C*) **and** *distinct-mset* (*mset D*)
using ‹$cdcl_W$*-all-struct-inv* (*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 *conf*: (*mset C* − {#*L*#}) #∪ (*mset D* − {#− *L*#}) =
remdups-mset (*mset C* − {#*L*#} + (*mset D* − {#− *L*#}))
by (*auto simp*: *distinct-mset-rempdups-union-mset*)
show *?case*

apply (*rule resolve-rule*)
using *LC LD max M conf C D* **by** (*auto simp*: *subset-mset.sup.commute*)
qed *auto*

lemma *do-resolve-step-no*:
do-resolve-step S = S $\Longrightarrow$ *no-step resolve S*
apply (*cases 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*] *do-resolve-step*)

lemma *rough-state-of-state-of-resolve*[*simp*]:
$cdcl_W$*-all-struct-inv S* $\Longrightarrow$ *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-trail-is-None*[*iff*]:
do-resolve-step S = (*a*, *b*, *c*, *d*, *None*) $\longleftrightarrow$ *S* = (*a*, *b*, *c*, *d*, *None*)
by (*cases S rule*: *do-resolve-step.cases*) *auto*

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*) $\Longrightarrow$
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 *f4 a2 a1*[*of L*′ # *D*] **unfolding** *find-def* **by** (*metis* (*no-types*) *add-diff-cancel-left*′ *mset.simps*(*2*) *option.inject prod.inject union-commute*)+
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*′#}) ⟹ *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 (*Marked K k* # *Ls*) = (*if k* = *Suc i then Some* (*Marked K k* # *Ls*) *else bt-cut i Ls*) |
bt-cut i [] = *None*

lemma *bt-cut-some-decomp*:
bt-cut i M = *Some M*′ ⟹ ∃ *K M2 M1*. *M* = *M2* @ *M*′ ∧ *M*′ = *Marked K* (*i*+*1*) # *M1*
by (*induction i M rule*: *bt-cut.induct*) (*auto split*: *if-split-asm*)

lemma *bt-cut-not-none*: *M* = *M2* @ *Marked K* (*Suc i*) # *M*′ ⟹ *bt-cut i M* ≠ *None*
by (*induction M2 arbitrary*: *M rule*: *marked-lit-list-induct*) *auto*

lemma *get-all-marked-decomposition-ex*:
∃ *N*. (*Marked K* (*Suc i*) # *M*′, *N*) ∈ *set* (*get-all-marked-decomposition* (*M2*@*Marked K* (*Suc i*) # *M*′))
apply (*induction M2 rule*: *marked-lit-list-induct*)
apply *auto*[*2*]
by (*rename-tac L m xs*, *case-tac get-all-marked-decomposition* (*xs* @ *Marked K* (*Suc i*) # *M*′)) *auto*

lemma *bt-cut-in-get-all-marked-decomposition*:
bt-cut i M = *Some M*′ ⟹ ∃ *M2*. (*M*′, *M2*) ∈ *set* (*get-all-marked-decomposition M*)
by (*auto dest*!: *bt-cut-some-decomp simp add*: *get-all-marked-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 (*Marked* - - # *Ls*) ⇒ (*Propagated L D* # *Ls*, *N*, *D* # *U*, *j*, *None*)
| - ⇒ (*M*, *N*, *U*, *k*, *Some D*))
) |
do-backtrack-step S = *S*

lemma *get-all-marked-decomposition-map-convert*:
(*get-all-marked-decomposition* (*map mmset-of-mlit′ M*)) =
map (λ(*a*, *b*). (*map mmset-of-mlit′ a*, *map mmset-of-mlit′ b*)) (*get-all-marked-decomposition M*)
apply (*induction M rule*: *marked-lit-list-induct*)
apply *simp*
by (*rename-tac L l xs*, *case-tac get-all-marked-decomposition xs*; *auto*)+

lemma *do-backtrack-step*:
assumes
db: *do-backtrack-step S* ≠ *S* **and**
inv: $cdcl_W$*-all-struct-inv S*
shows *backtrack S* (*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*)
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 M_2 **where** M_2: *bt-cut j M* = *Some* M_2
using *db fd* **unfolding** *S E* **by** (*auto split*: *option.splits*)
obtain *M1 K* **where** *M1*: M_2 = *Marked K* (*Suc j*) # *M1*
using *bt-cut-some-decomp*[*OF* M_2] **by** (*cases* M_2) *auto*
obtain *c* **where** *c*: *M* = *c* @ *Marked K* (*Suc j*) # *M1*
using *bt-cut-in-get-all-marked-decomposition*[*OF* M_2]
unfolding *M1* **by** *fastforce*
have *get-all-levels-of-marked* (*map mmset-of-mlit′ M*) = *rev* [*1*..<*Suc k*]
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def S* **by** *auto*
from *arg-cong*[*OF this*, *of* λ*a*. *Suc j* ∈ *set a*] **have** *j* ≤ *k* **unfolding** *c* **by** *auto*
have *max-l-j*: *maximum-level-code C′ M* = *j*
using *db fd* M_2 *C* **unfolding** *S E* **by** (*auto*
split: *option.splits list.splits marked-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 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*: (M_2, *M2*) ∈ *set* (*get-all-marked-decomposition M*)
using *bt-cut-in-get-all-marked-decomposition*[*OF* M_2] **by** *metis*
have *decomp*:
(*Marked K* (*Suc* (*get-maximum-level M* (*remove1-mset L* (*mset C*)))) # (*map mmset-of-mlit′ M1*),
(*map mmset-of-mlit′ M2*)) ∈
set (*get-all-marked-decomposition* (*map mmset-of-mlit′ M*))
using *imageI*[*of - - λ*(*a*, *b*). (*map mmset-of-mlit′ a*, *map mmset-of-mlit′ b*), *OF M2*] *j*
unfolding *S E M1* **by** (*auto simp add*: *get-all-marked-decomposition-map-convert*)
have *red*: (*reduce-trail-to* (*map mmset-of-mlit′ M1*)
(*M*, *N*, *C* # *U*, *get-maximum-level M* (*remove1-mset L* (*mset C*)), *None*))
= (*M1*, *N*, *C* # *U*, *get-maximum-level M* (*remove1-mset L* (*mset C*)), *None*)
using *M2 M1* **by** (*auto simp*: *reduce-trail-to*)
show *?case*
apply (*rule backtrack-rule*)
using M_2 *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*]
unfolding *CDCL-W-Implementation.state-eq-def*
using M_2 *fd confl* ‹*L* ∈ *set C*› *j decomp levL* ‹*get-maximum-level M* (*mset C*) = *k*› *red*
unfolding *S E M1*
by *auto*
qed

lemma *map-eq-list-length*:
map f L = *L′* ⟹ *length L* = *length L′*
by *auto*

lemma *map-mmset-of-mlit-eq-cons*:
assumes *map mmset-of-mlit′ M* = *a* @ *c*
obtains *a′ c′* **where**
M = *a′* @ *c′* **and**
a = *map mmset-of-mlit′ a′* **and**
c = *map mmset-of-mlit′ 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 *do-backtrack-step-no*:
assumes
db: *do-backtrack-step S* = *S* **and**
inv: *cdcl*$_W$-*all-struct-inv S*
shows *no-step backtrack S*
proof (*rule ccontr*, *cases S*, *cases 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: (*Marked K* (*Suc j*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*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**
undef: *undefined-lit M1 L*
using *bt* **apply** *clarsimp*
apply (*elim backtrack-levE*)
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **apply** *fast*
apply (*cases S*)
by (*auto simp add*: *get-all-marked-decomposition-map-convert*)

obtain *c* **where** *c*: *trail S* = *c* @ *M2* @ *Marked K* (*Suc j*) # *M1*
using *decomp* **by** *blast*
have *get-all-levels-of-marked* (*trail S*) = *rev* [*1..<Suc k*]
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def S* **by** *auto*
from *arg-cong*[*OF this*, *of* λ*a*. *Suc j* ∈ *set a*] **have** *k* > *j*
unfolding *c* **by** (*auto simp*: *get-all-marked-decomposition-map-convert*)
have [*simp*]: *L* ∈ *set D*
using *LD* **by** *auto*
have *CD*: *C* = *mset 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′* @ *Marked K* (*Suc j*) # *M1′*
apply (*rule map-mmset-of-mlit-eq-cons*[*of M c* @ *M2 Marked K* (*Suc j*) # *M1*])
using *c S* **apply** *simp*
apply (*rule map-mmset-of-mlit-eq-cons*[*of -* [*Marked K* (*Suc j*)] *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 *cM* **by** *simp*
show *?case* **using** *db* **unfolding** *S E*
by (*auto split*: *option.splits list.splits marked-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 S*
shows *rough-state-of* (*state-of* (*do-backtrack-step S*))= *do-backtrack-step S*
proof (*rule state-of-inverse*)
have *f2*: *backtrack S* (*do-backtrack-step S*) ∨ *do-backtrack-step S* = *S*
using *do-backtrack-step inv* **by** *blast*
have ⋀*p*. ¬ *cdcl*$_W$*-o S p* ∨ *cdcl*$_W$*-all-struct-inv p*
using *inv cdcl*$_W$*-all-struct-inv-inv other* **by** *blast*
then have *do-backtrack-step S* = *S* ∨ *cdcl*$_W$*-all-struct-inv* (*do-backtrack-step S*)
using *f2 inv cdcl*$_W$*-o.intros cdcl*$_W$*-bj.intros* **by** *blast*
then show *do-backtrack-step S* ∈ {*S*. *cdcl*$_W$*-all-struct-inv S*}
using *inv* **by** *fastforce*
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* ⇒ (*Marked L* (*Suc k*) # *M*, *N*, *U*, *k+1*, *None*)) |
do-decide-step S = *S*

lemma *do-decide-step*:
fixes *S* :: *'v cdcl*$_W$*-state-inv-st*
assumes *do-decide-step S* ≠ *S*
shows *decide S* (*do-decide-step S*)
using *assms*
apply (*cases S*, *cases conflicting S*)
defer
apply (*auto split*: *option.splits simp add*: *decide.simps Marked-Propagated-in-iff-in-lits-of-l*
dest: *find-first-unused-var-undefined find-first-unused-var-Some*
intro:)[*1*]
proof −
fix *a* :: (*'v*, *nat*, *'v literal list*) *marked-lit list* **and**
b :: *'v literal list list* **and** *c* :: *'v literal list list* **and**
d :: *nat* **and** *e* :: *'v literal list option*
{
fix *a* :: (*'v*, *nat*, *'v literal list*) *marked-lit list* **and**
b :: *'v literal list list* **and** *c* :: *'v literal list list* **and**
d :: *nat* **and** *x2* :: *'v literal* **and** *m* :: *'v 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*::*'v clause*) ∈ *f* ‘ *set b*
using *a1* **by** *blast*
then have ⋀*f*. (*atms-of* (*f m*)::*'v set*) ⊆ *atms-of-ms* (*f* ‘ *set b*)
by *simp*
then have ⋀*n f*. (*n*::*'v*) ∈ *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 :: $'v$ *literal list* **and** $x2$
 have $m \in set\ b \Longrightarrow x2 \in set\ m \Longrightarrow x2 \notin lits\text{-}of\text{-}l\ a \Longrightarrow -\ x2 \notin lits\text{-}of\text{-}l\ a \Longrightarrow$
 $\exists\ aa{\in}set\ b.\ \neg\ atm\text{-}of\ `\ set\ aa \subseteq atm\text{-}of\ `\ lits\text{-}of\text{-}l\ a$
 by (*meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI*)
} **note** $H' = this$

assume $do\text{-}decide\text{-}step\ S \neq S$ **and**
 $S = (a, b, c, d, e)$ **and**
 $conflicting\ S = None$
then show $decide\ S\ (do\text{-}decide\text{-}step\ S)$
 using $H\ H'$ **by** (*auto split*: *option.splits simp*: *lits-of-def decide.simps*
 Marked-Propagated-in-iff-in-lits-of-l
 dest!: *find-first-unused-var-Some*)
qed

lemma *mmset-of-mlit'-eq-Marked*[*iff*]: $mmset\text{-}of\text{-}mlit'\ z = Marked\ x\ k \longleftrightarrow z = Marked\ x\ k$
 by (*cases z*) *auto*

lemma *do-decide-step-no*:
 $do\text{-}decide\text{-}step\ S = S \Longrightarrow no\text{-}step\ decide\ S$
 apply (*cases S*, *cases conflicting S*)

 apply (*auto simp*: *atms-of-ms-mset-unfold Marked-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*]:
 $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \Longrightarrow rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}decide\text{-}step\ S)) = do\text{-}decide\text{-}step\ S$
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*]:
 $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \Longrightarrow rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}skip\text{-}step\ S)) = do\text{-}skip\text{-}step\ S$
 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*)+

20.3.3 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**
 $Con\ xs = state\text{-}of\ (if\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ xs\ then\ xs\ else\ ([], [], [], 0, None))$

lemma [*code abstype*]:
 $Con\ (rough\text{-}state\text{-}of\ S) = S$

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 S*
∧ $cdcl_W$*-stgy*** (*raw-S0-*$cdcl_W$ (*raw-init-clss S*)) *S*}
morphisms *rough-state-from-init-state-of state-from-init-state-of*
proof
show ([],[], [], *0*, *None*) ∈ {*S.* $cdcl_W$*-all-struct-inv S*
∧ $cdcl_W$*-stgy*** (*raw-S0-*$cdcl_W$ (*raw-init-clss S*)) *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 S*
∧ $cdcl_W$*-stgy*** (*raw-S0-*$cdcl_W$ (*raw-init-clss S*)) *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*).
(*S′*, *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*], *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* (*rough-state-of S*)
(*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*** ((*do-cp-step* (*rough-state-of S*))) (
(*rough-state-of* (*do-full1-cp-step* (*state-of* (*do-cp-step* (*rough-state-of S*))))))
∨ *state-of* (*do-cp-step* (*rough-state-of S*)) = *S*
using *rough-state-of-state-of-do-cp-step*[*of S*] **unfolding** *do-cp-step′-def* **by** *fastforce*
have *f2*: ⋀*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*: ¬ *cdcl*$_W$*-cp* (*rough-state-of S*) (*do-cp-step* (*rough-state-of S*))
∨ *state-of* (*do-cp-step* (*rough-state-of S*)) = *S*
∨ *cdcl*$_W$*-cp*$^{++}$ (*rough-state-of S*)
(*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* ≠ *S*
then have *do-cp-step* (*rough-state-of S*) = *rough-state-of S*
⟶ *cdcl*$_W$*-cp*** (*rough-state-of S*) (*rough-state-of* (*do-full1-cp-step S*))
∨ *do-cp-step* (*rough-state-of S*) ≠ *rough-state-of S*
∧ *state-of* (*do-cp-step* (*rough-state-of S*)) ≠ *S*
using *f2 f1* **by** (*metis* (*no-types*))
then have *do-cp-step* (*rough-state-of S*) ≠ *rough-state-of S*
∧ *state-of* (*do-cp-step* (*rough-state-of S*)) ≠ *S*
∨ *cdcl*$_W$*-cp*** (*rough-state-of S*) (*rough-state-of* (*do-full1-cp-step S*))
by (*metis rough-state-of-state-of-do-cp-step*)
then have *cdcl*$_W$*-cp*** (*rough-state-of S*) (*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* (*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 ≠ *S*
then T
else
(*let U* = *do-resolve-step T in*
if U ≠ *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 S* **and**
st: *do-other-step S* ≠ *S*
shows $cdcl_W$*-o S* (*do-other-step S*)
using *st inv* **by** (*auto split*: *if-split-asm*
simp add: *Let-def*
intro: *do-skip-step do-resolve-step do-backtrack-step do-decide-step*
$cdcl_W$*-o.intros* $cdcl_W$*-bj.intros*)

lemma *do-other-step-no*:
assumes *inv*: $cdcl_W$*-all-struct-inv S* **and**
st: *do-other-step S* = *S*
shows *no-step* $cdcl_W$*-o 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* (*rough-state-of S*) (*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* (*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* ≠ *S* ⟹
rough-state-of S ≠ *rough-state-of* (*do-full1-cp-step S*)
proof −
assume *a1*: *do-full1-cp-step S* ≠ *S*
then have *S* ≠ *do-cp-step' S*
by *fastforce*
then show *?thesis*
by (*metis* (*no-types*) *do-cp-step'-def do-full1-cp-step-fix-point-of-do-full1-cp-step*
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* ≠ *S*
shows $cdcl_W$*-stgy* (*rough-state-of S*) (*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* (*rough-state-of S*) (*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* (*rough-state-of S*) **and**
nc: *no-step conflict* (*rough-state-of S*)
apply (*metis True* $cdcl_W$*-cp.simps do-cp-step-eq-no-step*
do-full1-cp-step-fix-point-of-do-full1-cp-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* (*rough-state-of S*)

by (*simp add*: $cdcl_W$*-cp.simps*)
moreover have *full* $cdcl_W$*-cp* (*rough-state-of* (*do-other-step′ S*))
(*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 *do-skip-step-trail-changed-or-conflict*:
assumes *d*: *do-other-step* $S \neq S$
and *inv*: $cdcl_W$*-all-struct-inv S*
shows *trail* $S \neq$ *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 S*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* **by** *auto*
have $cdcl_W$*-o S* (*do-other-step S*) **using** *do-other-step*[*OF inv d*] **.**
then show *?thesis*
using ‹$cdcl_W$*-M-level-inv S*›
proof (*induction do-other-step S rule*: $cdcl_W$*-o-induct-lev2*)
case *decide*
then show *?thesis*
apply (*cases S*)
apply (*auto dest*!: *find-first-unused-var-Some*
simp: *split*: *option.splits*)
by (*meson atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set contra-subsetD*)
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 K i M1 M2 L D*) **note** *decomp* = *this*(*1*) **and** *confl-S* = *this*(*3*) **and** *undef* = *this*(*6*)
and *U* = *this*(*7*)
then show *?case*
apply (*cases do-other-step S*)
apply (*auto split*: *if-split-asm simp*: *Let-def*)
apply (*cases S rule*: *do-skip-step.cases*; *auto split*: *if-split-asm*)
apply (*cases S rule*: *do-skip-step.cases*; *auto split*: *if-split-asm*)

apply (*cases S rule*: *do-backtrack-step.cases*;
auto split: *if-split-asm option.splits list.splits marked-lit.splits*
dest!: *bt-cut-some-decomp simp*: *Let-def*)
using *d* **apply** (*cases S rule*: *do-decide-step.cases*; *auto split*: *option.splits*)[]
done
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-trail-increase*:
$\exists\, c.\ raw\text{-}trail\ (do\text{-}cp\text{-}step\ S) = c\ @\ raw\text{-}trail\ S\ \wedge(\forall\, m \in set\ c.\ \neg\ is\text{-}marked\ m)$
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-trail-increase*:
$\exists\, c.\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}full1\text{-}cp\text{-}step\ S)) = c\ @\ raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)$
$\wedge\ (\forall\, m \in set\ c.\ \neg\ is\text{-}marked\ m)$
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-trail-increase do-full1-cp-step.simps rough-state-of-state-of-do-cp-step set-append*)

lemma *do-cp-step-conflicting*:
$conflicting\ (rough\text{-}state\text{-}of\ S) \neq None \Longrightarrow do\text{-}cp\text{-}step'\ S = S$
unfolding *do-cp-step'-def do-cp-step-def* **by** *simp*

lemma *do-full1-cp-step-conflicting*:
$conflicting\ (rough\text{-}state\text{-}of\ S) \neq None \Longrightarrow do\text{-}full1\text{-}cp\text{-}step\ S = S$
unfolding *do-cp-step'-def do-cp-step-def*
apply (*induction rule*: *do-full1-cp-step-induct*)
by (*rename-tac S*, *case-tac S ≠ do-cp-step' S*)
(*auto simp add*: *do-full1-cp-step.simps do-cp-step-conflicting*)

lemma *do-decide-step-not-conflicting-one-more-decide*:
assumes
$conflicting\ S = None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $Suc\ (length\ (filter\ is\text{-}marked\ (raw\text{-}trail\ S)))$
$= length\ (filter\ is\text{-}marked\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
using *assms* **unfolding** *do-other-step'-def*
by (*cases S*) (*force simp*: *Let-def split*: *if-split-asm option.splits*
dest!: *find-first-unused-var-Some-not-all-incl*)

lemma *do-decide-step-not-conflicting-one-more-decide-bt*:
assumes $conflicting\ S \neq None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $length\ (filter\ is\text{-}marked\ (raw\text{-}trail\ S)) <$
$length\ (filter\ is\text{-}marked\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
using *assms* **unfolding** *do-other-step'-def* **by** (*cases S*, *cases conflicting S*)
(*auto simp add*: *Let-def split*: *if-split-asm option.splits*)

lemma *do-other-step-not-conflicting-one-more-decide-bt*:
assumes
$conflicting\ (rough\text{-}state\text{-}of\ S) \neq None$ **and**
$conflicting\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S)) = None$ **and**
$do\text{-}other\text{-}step'\ S \neq S$
shows $length\ (filter\ is\text{-}marked\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)))$
$> length\ (filter\ is\text{-}marked\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S))))$
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 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: ⋀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 (cases do-other-step' S) auto
      then show ?thesis
        using f3 2 by (metis (no-types) do-decide-step.simps(2) do-resolve-step-trail-is-None
          do-skip-step-trail-is-None rough-state-of-inverse)
    qed
  show ?case
    using assms(2) S unfolding bt y inv
    apply simp
    by (auto simp add: M bt-cut-not-none
          split: option.splits
          dest!: bt-cut-some-decomp)
qed

lemma do-other-step-not-conflicting-one-more-decide:
  assumes conflicting (rough-state-of S) = None and
  do-other-step' S ≠ S
  shows 1 + length (filter is-marked (raw-trail (rough-state-of S)))
    = length (filter is-marked (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*) $\neq$ *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-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 *conflicting-do-resolve-step-iff*[*iff*]:
conflicting (*do-resolve-step S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
by (*cases S rule*: *do-resolve-step.cases*)
(*auto simp add*: *Let-def split*: *option.splits*)

lemma *conflicting-do-skip-step-iff*[*iff*]:
conflicting (*do-skip-step S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
by (*cases S rule*: *do-skip-step.cases*)
(*auto simp add*: *Let-def split*: *option.splits*)

lemma *conflicting-do-decide-step-iff*[*iff*]:
conflicting (*do-decide-step S*) = *None* $\longleftrightarrow$ *conflicting S* = *None*
by (*cases S rule*: *do-decide-step.cases*)
(*auto simp add*: *Let-def split*: *option.splits*)

lemma *conflicting-do-backtrack-step-imp*[*simp*]:
do-backtrack-step S $\neq$ *S* $\Longrightarrow$ *conflicting* (*do-backtrack-step S*) = *None*
by (*cases S rule*: *do-backtrack-step.cases*)
(*auto simp add*: *Let-def split*: *list.splits option.splits marked-lit.splits*)

lemma *do-skip-step-eq-iff-trail-eq*:
do-skip-step S = *S* $\longleftrightarrow$ *trail* (*do-skip-step S*) = *trail S*
by (*cases S rule*: *do-skip-step.cases*) *auto*

lemma *do-decide-step-eq-iff-trail-eq*:
do-decide-step S = *S* $\longleftrightarrow$ *trail* (*do-decide-step S*) = *trail S*
by (*cases S rule*: *do-decide-step.cases*) (*auto split*: *option.split*)

lemma *do-backtrack-step-eq-iff-trail-eq*:
do-backtrack-step S = *S* $\longleftrightarrow$ *raw-trail* (*do-backtrack-step S*) = *raw-trail S*
by (*cases S rule*: *do-backtrack-step.cases*)
(*auto split*: *option.split list.splits marked-lit.splits*
dest!: *bt-cut-in-get-all-marked-decomposition*)

lemma *do-resolve-step-eq-iff-trail-eq*:
do-resolve-step S = *S* $\longleftrightarrow$ *trail* (*do-resolve-step S*) = *trail S*
by (*cases S rule*: *do-resolve-step.cases*) *auto*

lemma *do-other-step-eq-iff-trail-eq*:
do-other-step S = *S* $\longleftrightarrow$ *raw-trail* (*do-other-step S*) = *raw-trail S*

apply
(*auto simp add*: *Let-def do-skip-step-eq-iff-trail-eq*
do-decide-step-eq-iff-trail-eq do-backtrack-step-eq-iff-trail-eq
do-resolve-step-eq-iff-trail-eq
)
apply (*simp add*: *do-resolve-step-eq-iff-trail-eq*[*symmetric*]
do-skip-step-eq-iff-trail-eq[*symmetric*])
apply (*simp add*: *do-skip-step-eq-iff-trail-eq*[*symmetric*]
do-decide-step-eq-iff-trail-eq do-backtrack-step-eq-iff-trail-eq[*symmetric*]
do-resolve-step-eq-iff-trail-eq[*symmetric*]
)
done

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*: *conflicting* (*rough-state-of ?T*) ≠ *None*
then have *tr*: *trail* (*rough-state-of* (*do-full1-cp-step ?T*)) = *trail* (*rough-state-of ?T*)
using *do-full1-cp-step-conflicting* **by** *fastforce*
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*)
using *confl* **by** (*auto simp add*: *do-full1-cp-step-conflicting*)
then have *do-other-step′ S* = *S*
by (*simp add*: *do-other-step-eq-iff-trail-eq*[*symmetric*] *do-other-step′-def*
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-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*: *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-marked m*
using *do-full1-cp-step-neq-trail-increase* **by** *auto*
have *length* (*filter is-marked* (*raw-trail* (*rough-state-of* (*do-full1-cp-step ?T*))))
= *length* (*filter is-marked* (*raw-trail* (*rough-state-of ?T*)))
using *nm* **unfolding** *c* **by** *force*
moreover have *length* (*filter is-marked* (*raw-trail* (*rough-state-of S*)))
≠ *length* (*filter is-marked* (*raw-trail* (*rough-state-of ?T*)))
using *do-other-step-not-conflicting-one-more-decide*[*OF* - *neq*]
do-other-step-not-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* (*rough-state-of S*)
proof −
 {
  fix S'
  assume *full1 cdcl*$_W$*-cp* (*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* (*rough-state-of S*) S' **and**
   no-step propagate (*rough-state-of S*) **and**
   no-step conflict (*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*]:
 rough-state-of (*state-of* (*rough-state-from-init-state-of S*))
  = *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-clss*: *cdcl*$_W$*-stgy S T* $\Longrightarrow$ *cdcl*$_W$*-M-level-inv S* $\Longrightarrow$ *init-clss S = init-clss T*
 using *rtranclp-cdcl*$_W$*-init-clss cdcl*$_W$*-stgy-is-rtranclp-cdcl*$_W$ **by** *fast*

lemma *clauses-toS-rough-state-of-do-cdcl*$_W$*-stgy-step*[*simp*]:
init-clss (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*state-of* (*rough-state-from-init-state-of S*))))
= *init-clss* (*rough-state-from-init-state-of S*) (**is** - = *init-clss ?S*)
proof (*cases do-cdcl*$_W$*-stgy-step* (*state-of ?S*) = *state-of ?S*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have $\bigwedge$*c. cdcl*$_W$*-M-level-inv* (*rough-state-of c*)
using *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-all-struct-inv-rough-state* **by** *blast*
then have $\bigwedge$*c. init-clss* (*rough-state-of c*) = *init-clss* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step c*))
∨ *do-cdcl*$_W$*-stgy-step c* = *c*
using *cdcl*$_W$*-stgy-no-more-init-clss do-cdcl*$_W$*-stgy-step* **by** *blast*
then show *?thesis*
using *False* **by** *force*
qed

lemma *raw-init-clss-do-cp-step*[*simp*]:
raw-init-clss (*do-cp-step S*) = *raw-init-clss S*
by (*cases S*) (*auto simp*: *do-cp-step-def do-propagate-step-def do-conflict-step-def*
split: *option.splits*)
lemma *raw-init-clss-do-cp-step'*[*simp*]:
raw-init-clss (*rough-state-of* (*do-cp-step' S*)) = *raw-init-clss* (*rough-state-of S*)
by (*simp add*: *do-cp-step'-def*)

lemma *raw-init-clss-rough-state-of-do-full1-cp-step*[*simp*]:
raw-init-clss (*rough-state-of* (*do-full1-cp-step S*))
= *raw-init-clss* (*rough-state-of S*)
apply (*rule do-full1-cp-step.induct*[*of* λ*S*.
raw-init-clss (*rough-state-of* (*do-full1-cp-step S*))
= *raw-init-clss* (*rough-state-of S*)])
by (*metis* (*mono-tags*, *lifting*) *do-full1-cp-step.simps raw-init-clss-do-cp-step'*)

lemma *raw-init-clss-do-skip-def*[*simp*]:
raw-init-clss (*do-skip-step S*) = *raw-init-clss S*
by (*cases S rule*: *do-skip-step.cases*) (*auto simp*: *do-other-step'-def Let-def*
split: *option.splits*)

lemma *raw-init-clss-do-resolve-def*[*simp*]:
raw-init-clss (*do-resolve-step S*) = *raw-init-clss S*
by (*cases S rule*: *do-resolve-step.cases*) (*auto simp*: *do-other-step'-def Let-def*
split: *option.splits*)

lemma *raw-init-clss-do-backtrack-def*[*simp*]:
raw-init-clss (*do-backtrack-step S*) = *raw-init-clss S*
by (*cases S rule*: *do-backtrack-step.cases*) (*auto simp*: *do-other-step'-def Let-def*
split: *option.splits list.splits marked-lit.splits*)

lemma *raw-init-clss-do-decide-def*[*simp*]:
raw-init-clss (*do-decide-step S*) = *raw-init-clss S*
by (*cases S rule*: *do-decide-step.cases*) (*auto simp*: *do-other-step'-def Let-def*
split: *option.splits*)

lemma *raw-init-clss-rough-state-of-do-other-step'*[*simp*]:

raw-init-clss (*rough-state-of* (*do-other-step'* *S*))
= *raw-init-clss* (*rough-state-of* *S*)
by (*cases S*) (*auto simp*: *do-other-step'-def Let-def do-skip-step.cases*
split: *option.splits*)

lemma [*simp*]:
raw-init-clss (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*state-of* (*rough-state-from-init-state-of* *S*))))
=
raw-init-clss (*rough-state-from-init-state-of* *S*)
unfolding *do-cdcl*$_W$*-stgy-step-def* **by** (*auto simp*: *Let-def*)

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*** (*raw-S0-cdcl*$_W$ (*raw-init-clss* (*rough-state-from-init-state-of* *S*)))
(*rough-state-from-init-state-of* *S*)
using *rough-state-from-init-state-of*[*of S*] **by** *auto*
moreover have *cdcl*$_W$*-stgy***
(*rough-state-from-init-state-of* *S*)
(*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* (*rough-state-from-init-state-of* *T*),
cdcl$_W$*-measure* (*rough-state-from-init-state-of* *S*))
∈ *lexn* {(*a*, *b*). *a* < *b*} *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*** (*raw-S0-cdcl*$_W$ (*raw-init-clss ?S*)) *?S*
using *rough-state-from-init-state-of*[*of S*] **by** *auto*
moreover have *cdcl*$_W$*-stgy* (*rough-state-from-init-state-of* *S*)
(*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* **by** *force*
then have *do-cdcl*$_W$*-stgy-step* (*state-of* (*rough-state-from-init-state-of* *S*))

$\neq$ *state-of* (*rough-state-from-init-state-of S*)
using *ST T rough-state-from-init-state-of-inverse*
unfolding *id-of-I-to-def do-cdcl$_W$-stgy-step'-def*
by *fastforce*
from *do-cdcl$_W$-stgy-step*[*OF this*] **show** *?thesis*
by (*simp add*: *T id-of-I-to-def rough-state-from-init-state-of-do-cdcl$_W$-stgy-step'*)
qed
moreover
have *cdcl$_W$-all-struct-inv* (*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* (*raw-S0-cdcl$_W$* (*raw-init-clss* (*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*
by (*auto intro*!: *cdcl$_W$-stgy-step-decreasing*[*of - - raw-S0-cdcl$_W$* (*raw-init-clss ?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* $\neq$ *S* $\Longrightarrow$ *P* (*do-cdcl$_W$-stgy-step' S*)) $\Longrightarrow$ *P S*) $\Longrightarrow$ *P a0*
using *do-all-cdcl$_W$-stgy.induct* **by** *metis*

lemma [*simp*]: *raw-init-clss* (*rough-state-from-init-state-of* (*do-all-cdcl$_W$-stgy S*)) =
raw-init-clss (*rough-state-from-init-state-of S*)
apply (*induction rule*: *do-all-cdcl$_W$-stgy-induct*)
by (*smt do-all-cdcl$_W$-stgy.simps do-cdcl$_W$-stgy-step-def id-of-I-to-def*
raw-init-clss-rough-state-of-do-full1-cp-step raw-init-clss-rough-state-of-do-other-step'
rough-state-from-init-state-of-do-cdcl$_W$-stgy-step'
toS-rough-state-of-state-of-rough-state-from-init-state-of)

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* (*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*: ¬ *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 ¬ *cdcl$_W$-stgy* (*rough-state-from-init-state-of* (*do-all-cdcl$_W$-stgy Sa*)) *pp*
using *a1* **by** (*smt 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* (*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* (*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* (*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*** (*rough-state-from-init-state-of S*)
   (*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* ⟶
    rough-state-from-init-state-of (*do-cdcl*$_W$*-stgy-step′ S*)
    = *rough-state-of* (*state-of* (*rough-state-from-init-state-of S*))
    unfolding *rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step′*
    id-of-I-to-def **by** *presburger*
   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* (*rough-state-from-init-state-of S*)
     (*rough-state-from-init-state-of* (*do-cdcl*$_W$*-stgy-step′ S*))
    = *cdcl*$_W$*-stgy* (*rough-state-of* (*id-of-I-to S*))
     (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*id-of-I-to S*)))
    unfolding *id-of-I-to-def rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step′*
    toS-rough-state-of-state-of-rough-state-from-init-state-of **by** *presburger*
   then show *?thesis*
    using *f3 f2 IH do-cdcl*$_W$*-stgy-step*
    by (*smt False toS-rough-state-of-state-of-rough-state-from-init-state-of tranclp.intros*(*1*)
     tranclp-into-rtranclp transitive-closurep-trans′(*2*))
  qed
qed

Final theorem:

lemma *consistent-interp-mmset-of-mlit*[*simp*]:
 consistent-interp (*lit-of* ‘ *mmset-of-mlit′* ‘ *set M′*) ⟷
  consistent-interp (*lit-of* ‘ *set M′*)
 by (*auto simp*: *image-image*)

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*) = *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* ([], *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* ([], *?N*, [], *0*, *None*) *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*
by (*auto simp del*: *do-all-cdcl$_W$-stgy.simps simp*: *state-from-init-state-of-inverse*
r[*symmetric*] *no-step-cdcl$_W$-stgy-cdcl$_W$-all*)+
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 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 simp*: *lits-of-def*)
moreover
have [*simp*]:
rough-state-from-init-state-of (*state-from-init-state-of* (*raw-S0-cdcl$_W$* (*map remdups N*)))
= *raw-S0-cdcl$_W$* (*map remdups N*)
apply (*rule cdcl$_W$-state-inv-from-init-state.state-from-init-state-of-inverse*)
using *3* **by** (*auto simp*: *cdcl$_W$-all-struct-inv-def distinct-cdcl$_W$-state-def*
image-image comp-def)
have *raw-init-clss* ([], *?N*, [], *0*, *None*) = *raw-init-clss S*
using *arg-cong*[*OF r*, *of raw-init-clss*] **unfolding** *S*[*symmetric*]
by (*simp del*: *do-all-cdcl$_W$-stgy.simps*)
then have *N'*: *N' =map remdups N*
using *S*[*symmetric*] **by** *auto*

have *conflicting S = Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss S*)) $\vee$
conflicting S = None $\wedge$ (*case S of* (*M*, *uu-*) $\Rightarrow$ *map mmset-of-mlit' M*) $\models$*asm init-clss S*
apply (*rule full-cdcl$_W$-stgy-final-state-conclusive*)
using *1* **apply** *simp*
using *2* **apply** *simp*
using *3* **by** *simp*
then have (*E* $\neq$ *Some* [] $\wedge$ *satisfiable* (*set* (*map mset ?N*)))
$\vee$ (*E = Some* [] $\wedge$ *unsatisfiable* (*set* (*map mset ?N*)))
using *cons* **unfolding** *S*[*symmetric*] *N'* **apply** (*auto simp*: *comp-def*)
by (*simp add*: *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-W-Merge*
imports *CDCL-W-Termination*
begin

21 Link between Weidenbach's and NOT's CDCL

21.1 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!: backtrack-levE[OF - inv] simp: cdcl_W-M-level-inv-def)
   apply (elim resolveE, force elim!: backtrack-levE[OF - inv] simp: cdcl_W-M-level-inv-def)
  apply standard
  apply (elim backtrack-levE[OF - inv], elim backtrackE)
  apply (force simp del: state-simp simp add: state-eq-def cdcl_W-M-level-inv-decomp)
  done


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* $\Rightarrow$ $'v$ *clause* $\Rightarrow$ $'v$ *literal* $\Rightarrow$ $'st$ $\Rightarrow$ $'st$ $\Rightarrow$ *bool* **where**
backjump-l-cond $\equiv \lambda C\ C'\ L'\ S\ T.$ *True*

definition inv_{NOT} :: $'st \Rightarrow$ *bool* **where**
$inv_{NOT} \equiv \lambda S.$ *no-dup* (*trail* S)

declare inv_{NOT}*-def*[*simp*]
end

context *conflict-driven-clause-learning*$_W$
begin

21.2 More lemmas conflict–propagate and backjumping

21.2.1 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-marked-decomposition-exists-prepend*
    elim!: *backtrack-levE skipE resolveE*
    simp: $cdcl_W$*-M-level-inv-def*)+

lemma *wf-*$cdcl_W$*-bj*:
  wf {(*b*,*a*). $cdcl_W$*-bj a b* $\wedge$ $cdcl_W$*-M-level-inv a*}
  apply (*rule wfP-if-measure*[*of* λ-. *True*
      - λT. *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 $\exists$ T. *full* $cdcl_W$*-bj* S T
proof –
  obtain T **where** T: *full* ($\lambda a\ b.$ $cdcl_W$*-bj a b* $\wedge$ $cdcl_W$*-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

$\exists M.$ *trail* $S = M$ @ *trail* $T \wedge (\forall m \in$ *set* $M. \neg$*is-marked* $m)$
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*)

21.2.2 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: *raw-conflicting* V = *Some* D **and**
LD: $L \in\#$ *mset-ccls* D **and**
decomp: (*Marked* K (*Suc* i) # *M1*, *M2*) $\in$ *set* (*get-all-marked-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) (*mset-ccls* D) **and**
max-D: *get-maximum-level* (*trail* V) (*remove1-mset* L (*mset-ccls* D)) $\equiv i$ **and**
undef: *undefined-lit* *M1* L **and**
W: $W \sim$ *cons-trail* (*Propagated* L (*cls-of-ccls* D))
(*reduce-trail-to* *M1*
(*add-learned-cls* (*cls-of-ccls* D)
(*update-backtrack-lvl* i
(*update-conflicting* *None* V))))
using *bt inv* **by** (*elim backtrack-levE*) *metis+*
obtain L' C' M E **where**
tr: *trail* T = *Propagated* L' C' # M **and**
raw: *raw-conflicting* T = *Some* E **and**
LE: $-L' \notin\#$ *mset-ccls* E **and**
E: *mset-ccls* $E \neq \{\#\}$ **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 $DE[simp]$: *mset-ccls D = mset-ccls 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-rev-level-ge-0-atm-of-in*[*of 0 rev* (*trail V*) *L*] **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* (*mset-ccls D*)
  using *inv' raw* **unfolding** $cdcl_W$*-conflicting-def* $cdcl_W$*-all-struct-inv-def tr-M* **by** *auto*
then have *L'* ∉# *mset-ccls* (*remove-clit L D*)
  using *L-L' L'-M* ‹*Propagated L' C'* # *trail V* ⊨*as CNot* (*mset-ccls 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 undef 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* (*cls-of-ccls E*)) (*reduce-trail-to M1*
  (*add-learned-cls* (*cls-of-ccls E*) (*update-backtrack-lvl i* (*update-conflicting None T*))))
  using *W V undef M-lev decomp tr*
  by (*auto simp del*: *state-simp simp*: *state-eq-def* $cdcl_W$*-M-level-inv-def*)

obtain *M2'* **where**
  decomp': (*Marked K* (*i+1*) # *M1*, *M2'*) ∈ *set* (*get-all-marked-decomposition* (*trail T*))
  using *decomp V* **unfolding** *tr-M* **by** (*cases hd* (*get-all-marked-decomposition* (*trail V*)),
    cases get-all-marked-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* (*mset-ccls D*)
    by (*metis DE LE L-L'* ‹*L'* ∉# *mset-ccls* (*remove-clit L D*)› *in-remove1-mset-neq remove-clit*
      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* (*mset-ccls D*)
    using *calculation*(*2*) *lev-L max* **by** *auto*
moreover

    have *atm-of L′ ∉ atms-of (mset-ccls (remove-clit L D))*
      by (*metis DE LE ‹L′ ∉# mset-ccls (remove-clit L D)›*
        atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def in-remove1-mset-neq remove-clit
        in-atms-of-remove1-mset-in-atms-of)
    have *i = get-maximum-level ?M (mset-ccls (remove-clit L D))*
      using *max-D ‹atm-of L′ ∉ atms-of (mset-ccls (remove-clit L D))›* **by** *auto*

  ultimately have *backtrack T W*
    apply −
    apply (*rule backtrack-rule*[*of T - L K i M1 M2′ W, OF raw*])
    unfolding *tr-M*[*symmetric*]
          using *LD* **apply** *simp*
        apply *simp*
      apply *simp*
    apply *simp*
   apply *auto*[]
    using *W-S* **by** *auto*
  then show *?thesis* **using** *IH inv* **by** *blast*
qed

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

See also ⟦$skip^{**}$ *?S ?T*; *backtrack ?T ?W*; $cdcl_W$*-all-struct-inv ?S*⟧ ⟹ *backtrack ?S ?W*

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*!: *backtrack-levE*)
  then obtain *K i M1 M2 L D* **where**
    raw-S: *raw-conflicting S = Some D* **and**
    LD: *L* ∈# *mset-ccls D* **and**
    decomp: (*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-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*) (*mset-ccls D*) **and**
    i: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* **and**
    undef: *undefined-lit M1 L* **and**
    W: *W* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
              (*reduce-trail-to M1*
                (*add-learned-cls* (*cls-of-ccls D*)
                  (*update-backtrack-lvl i*
                    (*update-conflicting None S*))))

    using *bt* **by** (*elim backtrack-levE*)
    (*simp-all add*: $cdcl_W$*-M-level-inv-decomp state-eq-def del*: *state-simp*)

let *?D* = *remove1-mset L* (*mset-ccls 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*: $\forall m \in set\ MS.\ \neg is\text{-}marked\ m$
  using *rtranclp-skip-state-decomp*(*1*)[*OF skip*] *raw-S M-lev* **by** *auto*
have *T*: *state T* = (M_T, *init-clss S*, *learned-clss S*, *backtrack-lvl S*, *Some* (*mset-ccls D*))
  using M_T *rtranclp-skip-state-decomp*[*of S T*] *skip raw-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 $M_T \models as$ *CNot* (*mset-ccls D*)
  unfolding $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def* **using** *T* **by** *blast*
then have $\forall L \in \# mset\text{-}ccls\ D.\ atm\text{-}of\ L \in atm\text{-}of$ ‘ *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 $\forall L \in \# mset\text{-}ccls\ D.\ atm\text{-}of\ L \notin atm\text{-}of$ ‘ *lits-of-l MS*
  unfolding *M* **unfolding** *lits-of-def* **by** *auto*
then have *H*: $\bigwedge L.\ L \in \# mset\text{-}ccls\ D \Longrightarrow get\text{-}level\ (trail\ S)\ L = get\text{-}level\ M_T\ L$
  unfolding *M* **by** (*fastforce simp*: *lits-of-def*)
have [*simp*]: *get-maximum-level* (*trail S*) (*mset-ccls D*) = *get-maximum-level* M_T (*mset-ccls D*)
  using ‹$M_T \models as$ *CNot* (*mset-ccls D*)› *M nm* **by** (*metis true-annots-CNot-all-atms-defined get-maximum-level-skip-un-marked-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-marked-invert get-all-marked-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp*)
have *W*: *W* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*)) (*reduce-trail-to M1* (*add-learned-cls* (*cls-of-ccls D*) (*update-backtrack-lvl i* (*update-conflicting None T*))))
  using *W T i decomp undef* **by** (*auto simp del*: *state-simp simp*: *state-eq-def*)
have *lev-l-D'*: *get-level* M_T *L* = *get-maximum-level* M_T (*mset-ccls 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 *Marked K* (*i* + *1*) # *M1* ∈ *set* (*map fst* (*get-all-marked-decomposition* (*trail S*)))
  using *Set.imageI*[*OF decomp*, *of fst*] **by** *auto*
then have *Marked K* (*i* + *1*) # *M1* ∈ *set* (*map fst* (*get-all-marked-decomposition* M_T))
  using *fst-get-all-marked-decomposition-prepend-not-marked*[*OF nm*] **unfolding** *M* **by** *auto*
then obtain *M2'* **where** *decomp'*:(*Marked K* (*i*+*1*) # *M1*, *M2'*) ∈ *set* (*get-all-marked-decomposition*

M_T)
  by *auto*
 then show *backtrack T W*
  using *T decomp′ lev-l′ lev-l-D′ i′ W LD undef*
  by (*force intro*!: *backtrack.intros 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 dest*: *hd-raw-trail*)

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* ∼ *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**
  raw-S: *raw-conflicting S* = *Some D* **and**

LD: $L \in\#$ *mset-ccls D* **and**
decomp: (*Marked K* (*Suc i*) # *M1*, *M2*) $\in$ *set* (*get-all-marked-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*) (*mset-ccls D*) **and**
i: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls D*)) $\equiv$ *i* **and**
undef: *undefined-lit M1 L* **and**
T: *T* $\sim$ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls D*)
(*update-backtrack-lvl i*
(*update-conflicting None S*))))
using *bt-T* **by** (*elim backtrack-levE*) (*force simp*: $cdcl_W$*-M-level-inv-def*)+

obtain *K' i' M1' M2' L' D'* **where**
raw-S': *raw-conflicting S* = *Some D'* **and**
LD': $L' \in\#$ *mset-ccls D'* **and**
decomp': (*Marked K'* (*Suc i'*) # *M1'*, *M2'*) $\in$ *set* (*get-all-marked-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*) (*mset-ccls D'*) **and**
i': *get-maximum-level* (*trail S*) (*remove1-mset L'* (*mset-ccls D'*)) $\equiv$ *i'* **and**
undef': *undefined-lit M1' L'* **and**
U: *U* $\sim$ *cons-trail* (*Propagated L'* (*cls-of-ccls D'*))
(*reduce-trail-to M1'*
(*add-learned-cls* (*cls-of-ccls D'*)
(*update-backtrack-lvl i'*
(*update-conflicting None S*))))
using *bt-U lev* **by** (*elim backtrack-levE*) (*force simp*: $cdcl_W$*-M-level-inv-def*)+
obtain *c* **where** *M*: *trail S* = *c* @ *M2* @ *Marked K* (*i* + *1*) # *M1*
using *decomp* **by** *auto*
obtain *c'* **where** *M'*: *trail S* = *c'* @ *M2'* @ *Marked K'* (*i'* + *1*) # *M1'*
using *decomp'* **by** *auto*
have *marked*: *get-all-levels-of-marked* (*trail S*) = *rev* [*1*..<*1*+*backtrack-lvl S*]
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *i* < *backtrack-lvl S*
unfolding *M* **by** (*force simp add*: *rev-swap*[*symmetric*] *dest*!: *arg-cong*[*of - - set*])

have [*simp*]: *L'* = *L*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have $L' \in\#$ *remove1-mset L* (*mset-ccls D*)
using *raw-S raw-S' LD LD'* **by** (*simp add*: *in-remove1-mset-neq*)
then have *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls 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*]: *mset-ccls D'* = *mset-ccls D*
using *raw-S raw-S'* **by** *auto*
have [*simp*]: *i'* = *i*
using *i i'* **by** *auto*

Automation in a step later...

have *H*: $\bigwedge$*a A B*. *insert a A* = *B* $\Longrightarrow$ *a* : *B*
by *blast*

have *get-all-levels-of-marked* (*c*@*M2*) = *rev* [*i*+*2*..<*1*+*backtrack-lvl S*] **and**
get-all-levels-of-marked (*c'*@*M2'*) = *rev* [*i*+*2*..<*1*+*backtrack-lvl S*]
using *marked* **unfolding** *M*
using *marked* **unfolding** *M'*
unfolding *rev-swap*[*symmetric*] **by** (*auto dest*: *append-cons-eq-upt-length-i-end*)
from *arg-cong*[*OF this*(*1*), *of set*] *arg-cong*[*OF this*(*2*), *of set*]
have
dropWhile (λ*L*. ¬*is-marked L* ∨ *level-of L* ≠ *Suc i*) (*c* @ *M2*) = [] **and**
dropWhile (λ*L*. ¬*is-marked L* ∨ *level-of L* ≠ *Suc i*) (*c'* @ *M2'*) = []
unfolding *dropWhile-eq-Nil-conv Ball-def*
by (*intro allI*; *rename-tac x*; *case-tac x*; *auto dest*!: *H simp add*: *in-set-conv-decomp*)+

then have [*simp*]: *M1'* = *M1*
using *arg-cong*[*OF M*, *of dropWhile* (λ*L*. ¬*is-marked L* ∨ *level-of L* ≠ *Suc i*)]
unfolding *M'* **by** *auto*
show *?thesis* **using** *T U undef 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* ≠ [] **and**
tr-U: *hd-raw-trail U* = *Propagated L E* **and**
LE: *L* ∈# *mset-cls E* **and**
raw-U: *raw-conflicting U* = *Some D* **and**
LD: −*L* ∈# *mset-ccls D* **and**
get-maximum-level (*trail U*) (*mset-ccls* (*remove-clit* (−*L*) *D*)) = *backtrack-lvl U* **and**
V: *V* ∼ *update-conflicting* (*Some* (*union-ccls* (*remove-clit* (−*L*) *D*)
(*ccls-of-cls* (*remove-lit L E*))))
(*tl-trail U*)
using *resolve* **by** (*auto elim*!: *resolveE*)
have $cdcl_W$*-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*+
then have
S: *init-clss U* = *init-clss S*
learned-clss U = *learned-clss S*
backtrack-lvl U = *backtrack-lvl S*
conflicting S = *Some* (*mset-ccls D*)
using *rtranclp-skip-state-decomp*[*OF skip*] *U raw-U*
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-marked m*
using *rtranclp-skip-state-decomp*[*OF skip*] **by** *blast*

obtain K' i' $M1'$ $M2'$ L' D' **where**
raw-S′: *raw-conflicting S = Some D′* **and**
LD′: $L' \in\#$ *mset-ccls D′* **and**
decomp′: (*Marked K′* (*Suc i′*) # *M1′*, *M2′*) ∈ *set* (*get-all-marked-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*) (*mset-ccls D′*) **and**
i′: *get-maximum-level* (*trail S*) (*remove1-mset L′* (*mset-ccls D′*)) ≡ *i′* **and**
undef′: *undefined-lit M1′ L′* **and**
R: *T* ∼ *cons-trail* (*Propagated L′* (*cls-of-ccls D′*))
(*reduce-trail-to M1′*
(*add-learned-cls* (*cls-of-ccls D′*)
(*update-backtrack-lvl i′*
(*update-conflicting None S*))))
using *bt* **by** (*elim backtrack-levE*) (*fastforce simp*: *S state-eq-def simp del*:*state-simp*)+
obtain *c* **where** *M*: *trail S* = *c* @ *M2′* @ *Marked K′* (*i′* + *1*) # *M1′*
using *get-all-marked-decomposition-exists-prepend*[*OF decomp′*] **by** *auto*
have *marked*: *get-all-levels-of-marked* (*trail S*) = *rev* [*1*..<*1*+*backtrack-lvl S*]
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *i′* < *backtrack-lvl S*
unfolding *M* **by** (*force simp add*: *rev-swap*[*symmetric*] *dest*!: *arg-cong*[*of - - set*])

have *U*: *trail U* = *Propagated L* (*mset-cls E*) # *trail V*
using *tr-S hd-raw-trail*[*OF U*] *U S V tr-U* **by** (*auto simp*: *lits-of-def*)
have *DD′*[*simp*]: *mset-ccls D′* = *mset-ccls D*
using *raw-U raw-S′ S* **by** *auto*
have [*simp*]: $L' = -L$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have $-L \in\#$ *remove1-mset L′* (*mset-ccls D′*)
using *DD′ LD′ LD* **by** (*simp add*: *in-remove1-mset-neq*)
moreover
have *M′*: *trail S* = M_0 @ *Propagated L* (*mset-cls 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-all-levels-of-marked* (*trail S*) = *rev* [*1*..<*1*+*backtrack-lvl S*]
using *inv U* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *get-all-levels-of-marked* (*trail U*) = *rev* [*1*..<*1*+*backtrack-lvl S*]
using *nm M′ U* **by** (*simp add*: *get-all-levels-of-marked-no-marked*)
then have *get-lev-L*:
get-level(*Propagated L* (*mset-cls E*) # *trail V*) *L* = *backtrack-lvl S*
using *get-level-get-rev-level-get-all-levels-of-marked*[*OF atm-L-notin-M*,
of [*Propagated L* (*mset-cls E*)]] *U* **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*
by (*metis M′ get-lev-L get-rev-level-notin-end rev-append*)
ultimately
have *get-maximum-level* (*trail S*) (*remove1-mset L′* (*mset-ccls D′*)) ≥ *backtrack-lvl S*
by (*metis get-maximum-level-ge-get-level get-rev-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* (*mset-cls E*) # *trail V* $\models$*as CNot* (*mset-ccls D'*)
using $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def raw-U U* **by** *auto*
then have $\forall L' \in \#$ (*remove1-mset L'* (*mset-ccls D'*)) . *atm-of L'* $\in$ *atm-of* ' *lits-of-l* (*Propagated L* (*mset-cls 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 *get-maximum-level* (*trail S*) (*remove1-mset L'* (*mset-ccls D'*)) = *backtrack-lvl S*
using *get-maximum-level-skip-un-marked-not-present*[*of remove1-mset L'* (*mset-ccls D'*)
trail U M_0] *tr-S nm U*
‹*get-maximum-level* (*trail U*) (*mset-ccls* (*remove-clit* (− *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* ∧ *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* ∧ *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* ⟹ $skip^{**}$ *U' W*
using *skip* **by** *auto*
then have (*λS T. skip-or-resolve S T* ∧ *no-step backtrack S*)** *U V*
using *IH H* **by** *blast*
moreover have (*λS T. skip-or-resolve S T* ∧ *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*: ∀ *p pa pb pc*. ¬ *p* (*pa*) *pb* ∨ ¬ p^{**} *pb pc* ∨ p^{**} *pa pc*
by (*meson converse-rtranclp-into-rtranclp*)
have *skip-or-resolve T U* ∧ *no-step backtrack T*
using *RS*(*2*) *RS*(*3*) **by** *force*
then have (*λp pa. skip-or-resolve p pa* ∧ *no-step backtrack p*)** *T W*
proof −
have (∃ *vr19 vr16 vr17 vr18*. *vr19* (*vr16*::'*st*) *vr17* ∧ $vr19^{**}$ *vr17 vr18*
∧ ¬ $vr19^{**}$ *vr16 vr18*)
∨ ¬ (*skip-or-resolve T U* ∧ *no-step backtrack T*)
∨ ¬ (*λuu uua. skip-or-resolve uu uua* ∧ *no-step backtrack uu*)** *U W*
∨ (*λuu uua. skip-or-resolve uu uua* ∧ *no-step backtrack uu*)** *T W*
by *force*
then show *?thesis*
by (*metis* (*no-types*) ‹(*λS T. skip-or-resolve S T* ∧ *no-step backtrack S*)** *U W*›
‹*skip-or-resolve T U* ∧ *no-step backtrack T*› *f1*)
qed
then have (*λp pa. skip-or-resolve p pa* ∧ *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 ∼ 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 $T \sim U \lor$ $cdcl_W$*-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*

21.3 CDCL FW

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 ($\lambda S\ T$. $cdcl_W$*-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* ∧ $cdcl_W$*-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* ∧ *conflicting V* = *None*)
∨ (∃ *T U*. $cdcl_W$*-merge-restart*** *S T* ∧ *conflicting V* ≠ *None* ∧ *conflict T U* ∧ $cdcl_W$*-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* ≠ *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*
∨ *cdcl*$_W$*-merge-restart*** *S T* ∧ *conflicting U* ≠ *None*
∧ *conflict T T'* ∧ *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*: *backtrack-levE*
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 $V \neq$ *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) $V \longleftrightarrow$ *full* $cdcl_W$*-merge-restart* (*init-state* N) V
by (*rule conflicting-true-full-*$cdcl_W$*-iff-full-*$cdcl_W$*-merge*) *auto*

21.4 FW with strategy

21.4.1 The intermediate step

inductive $cdcl_W$*-s'* :: *'st* ⇒ *'st* ⇒ *bool* **where**
conflict': *full1* $cdcl_W$*-cp* S $S' \Longrightarrow$ $cdcl_W$*-s'* S S' |
decide': *decide* S $S' \Longrightarrow$ *no-step* $cdcl_W$*-cp* $S \Longrightarrow$ *full* $cdcl_W$*-cp* S' $S'' \Longrightarrow$ $cdcl_W$*-s'* S S'' |
bj': *full1* $cdcl_W$*-bj* S $S' \Longrightarrow$ *no-step* $cdcl_W$*-cp* $S \Longrightarrow$ *full* $cdcl_W$*-cp* S' $S'' \Longrightarrow$ $cdcl_W$*-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 $S' \Longrightarrow$ *full* $cdcl_W$*-cp* S' $S'' \Longrightarrow$ $cdcl_W$*-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*′ $S\ T \Longrightarrow$ $cdcl_W$*-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* $T'\ U$
    $\vee$ ($\exists\ U'\ U''$. *full* $cdcl_W$*-cp* $T'\ U''$ $\wedge$ *full1* $cdcl_W$*-bj* $U\ U'$ $\wedge$ *full* $cdcl_W$*-cp* $U'\ U''$
      $\wedge$ $cdcl_W$*-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: ∀ *x2*. (∃ *v3*. *cdcl*$_W$*-cp x2 v3*) = *cdcl*$_W$*-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-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* ⇒ *'st* **where**
              f1: ∀ *s sa sb*. (¬ *full1* $cdcl_W$*-bj s sa* ∨ $cdcl_W$*-cp s* (*ss s*) ∨ ¬ *full* $cdcl_W$*-cp sa sb*)
                ∨ $cdcl_W$*-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*
    ∨ (∃ *U' U''*. $cdcl_W$*-s' S U''* ∧ *full1* $cdcl_W$*-bj U U'* ∧ *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 ≠ *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: $S = T \vee cdcl_W\text{-}stgy^{**}\ S\ ss \wedge cdcl_W\text{-}stgy\ ss\ T$
    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 $\forall s.\ \neg$ *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 $cdcl_W\text{-}s'^{**}\ S\ T$
        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'*) $cdcl_W\text{-}s'^{**}\ S\ T$
        | (*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'*
          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* $V \neq$ *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* $T \neq$ *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-S*′ = *this*(*1*) **and** *bj-T* = *this*(*2*)
have $cdcl_W$*-bj*$^{++}$ S' V
using *skip bj-T* **by** (*metis* ‹$U = V$› $cdcl_W$*-bj.skip tranclp.simps*)
then show *?thesis* **using** *S-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*′) $cdcl_W$*-s*′** S T
| (*bj*) S' **where** $cdcl_W$*-s*′** S S' **and** $cdcl_W$*-bj*$^{++}$ S' T **and** *conflicting* $T \neq$ *None*
using *IH* **by** *blast*
then show *?thesis*
proof *cases*
case *s*′
have $cdcl_W$*-bj*$^{++}$ T V
using *resolve* **by** *force*
then show *?thesis* **using** *s*′ *confl-V* **by** *auto*
next
case (*bj* S') **note** *S-S*′ = *this*(*1*) **and** *bj-T* = *this*(*2*)
have $cdcl_W$*-bj*$^{++}$ S' V
using *resolve bj-T* **by** (*metis* ‹$U = V$› $cdcl_W$*-bj.resolve tranclp.simps*)
then show *?thesis* **using** *confl-V S-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* $cdcl_W$*-s*′ $S \longleftrightarrow$ *no-step* $cdcl_W$*-cp* $S \land$ *no-step* $cdcl_W$*-o* S (**is** *?S*′ $S \longleftrightarrow$ *?C* $S \land$ *?O* S)

proof
  assume *?C S* ∧ *?O S*
  then show *?S′ S*
    by (*auto simp*: $cdcl_W$*-s′.simps full1-def tranclp-unfold-begin*)
next
  assume *n-s*: *?S′ S*
  have *?C S*
    proof (*rule ccontr*)
      assume ¬ *?thesis*
      then obtain *S′* **where** $cdcl_W$*-cp S S′*
        by *auto*
      then obtain *T* **where** *full1* $cdcl_W$*-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* $cdcl_W$*-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′* ⟹ $cdcl_W^{++}$ *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**
    $\forall x0\ x1\ x2.\ (\exists v3.\ x2\ x1\ v3 \wedge x2^{**}\ v3\ x0) = (x2\ x1\ (ss\ x0\ x1\ x2) \wedge x2^{**}\ (ss\ x0\ x1\ x2)\ x0)$
    by *moura*
  then have *f3*: $\forall p\ s\ sa.\ \neg\ p^{++}\ s\ sa \vee p\ s\ (ss\ sa\ s\ p) \wedge p^{**}\ (ss\ sa\ s\ p)\ sa$
    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 $\longleftrightarrow$ *full cdcl*$_W$*-s'* S T (**is** *?S* $\longleftrightarrow$ *?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* ⇒ *'st* ⇒ *bool* **where**
conflict'[*intro*]: *conflict* S $T \Longrightarrow$ *full* $cdcl_W$*-bj* T $U \Longrightarrow$ $cdcl_W$*-merge-cp* S U |
propagate'[*intro*]: *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*)

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 ∨ 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 ∨ 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* ∧ *conflicting V = None*)
∨ (*cdcl$_W$-merge-cp** S V* ∧ *conflicting V ≠ None* ∧ *no-step cdcl$_W$-cp V* ∧ *no-step cdcl$_W$-bj V*)
∨ (∃ *T*. *cdcl$_W$-merge-cp** S T* ∧ *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*
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*
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** *auto*
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** *auto*
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*
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*$_W$*-cp*$^{++}$ *V ss*
using *False* **by** (*metis option-full-cdcl*$_W$*-cp full-def*)
have $\bigwedge$*s.* $\neg$ *cdcl*$_W$*-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*$_W$*-s'-without-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*$_W$*-s'-without-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 ¬ *?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*$_W$*-merge-cp*** *S V* **and** *conflicting V* = *None*
| (*cp-false*) *cdcl*$_W$*-merge-cp*** *S V* **and** *conflicting V* ≠ *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* ⟷ *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* **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* $\Rightarrow$ *'st* $\Rightarrow$ *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* ∨ *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* ∨ *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*
        next
          case *s'*
          then have *R* = *V*
            by (*metis full1-def inv local.conflict' 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*
            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 $\vee$ (V' = W $\wedge$ *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 *auto*
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 *auto*
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* $W \neq None$
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*) *map-option-is-None r-into-rtranclp*)
qed
next
case *propa*
moreover then have $cdcl_W$*-merge-cp* V' W
by *auto*
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 ‹$cdcl_W$*-merge-cp* U V'› *cp-confl*(*1*) **by** *force*
next
assume *confl*: *conflicting* $V' \neq None$
then have *no-step* $cdcl_W$*-merge-stgy* V'
by (*fastforce simp*: $cdcl_W$*-merge-stgy.simps full1-def full-def*
$cdcl_W$*-merge-cp.simps dest*!: *tranclpD elim*: *rulesE*)
have *no-step* $cdcl_W$*-merge-cp* V'
using *confl* **by** (*auto simp*: *full1-def full-def* $cdcl_W$*-merge-cp.simps*
dest!: *tranclpD elim*: *rulesE*)
moreover have $cdcl_W$*-merge-cp* U W

using V'-W ‹*cdcl*$_W$*-merge-cp* U V'› **by** *blast*
ultimately have *full1 cdcl*$_W$*-merge-cp* R V'
using *cp-confl*(*1*) V'-W **unfolding** *full1-def* **by** *auto*
then have *cdcl*$_W$*-merge-stgy* R V'
by *auto*
moreover have *no-step cdcl*$_W$*-merge-stgy* V'
using *confl* ‹*no-step cdcl*$_W$*-merge-cp* V'› **by** (*auto simp*: *cdcl*$_W$*-merge-stgy.simps full1-def dest*!: *tranclpD elim*: *rulesE*)
ultimately have *cdcl*$_W$*-merge-stgy*** R V' **by** *auto*
{ **fix** *ss* :: *'st*
have *cdcl*$_W$*-merge-cp* U W
using V'-W ‹*cdcl*$_W$*-merge-cp* U V'› **by** *blast*
then have ¬ *cdcl*$_W$*-bj* W *ss*
by (*meson conflicting-not-true-rtranclp-cdcl*$_W$*-merge-cp-no-step-cdcl*$_W$*-bj cp-confl*(*1*) *rtranclp.rtrancl-into-rtrancl step.prems*)
then have *cdcl*$_W$*-merge-stgy*** R W ∧ *conflicting* W = *None* ∨ *cdcl*$_W$*-merge-stgy*** R W ∧ ¬ *cdcl*$_W$*-bj* W *ss*
using V'-W ‹*cdcl*$_W$*-merge-stgy*** R V'› **by** *presburger* }
then show *?thesis*
by *presburger*
qed
next
case *propa*
moreover then have *cdcl*$_W$*-merge-cp* V' W
by *auto*
ultimately show *?thesis* **using** ‹*cdcl*$_W$*-merge-cp* U V'› *cp-confl*(*1*) **by** *force*
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-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

We will discharge the assumption later.

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* ($\lambda S\ T$. *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: ($\lambda S\ T$. *cdcl*$_W$-*all-struct-inv S* ∧ *cdcl*$_W$-*merge-cp S T*)** *R S* **and**
    n-s: *no-step* ($\lambda S\ T$. *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'* :: *'e* **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-W-Restart*
imports *CDCL-W-Merge*
begin

21.5 Adding Restarts

locale $cdcl_W$*-restart* =
conflict-driven-clause-learning$_W$

— functions for clauses:
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:
mset-ccls union-ccls insert-ccls remove-clit

— conversion
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
— changing state:
cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state
restart-state
for
mset-cls:: $'cls \Rightarrow 'v$ *clause* **and**
insert-cls :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**
remove-lit :: $'v$ *literal* $\Rightarrow 'cls \Rightarrow 'cls$ **and**

mset-clss:: $'clss \Rightarrow 'v$ *clauses* **and**
union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
in-clss :: $'cls \Rightarrow 'clss \Rightarrow$ *bool* **and**
insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
remove-from-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

mset-ccls:: $'ccls \Rightarrow 'v$ *clause* **and**
union-ccls :: $'ccls \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
insert-ccls :: $'v$ *literal* $\Rightarrow 'ccls \Rightarrow 'ccls$ **and**
remove-clit :: $'v$ *literal* $\Rightarrow 'ccls \Rightarrow 'ccls$ **and**

ccls-of-cls :: $'cls \Rightarrow 'ccls$ **and**
cls-of-ccls :: $'ccls \Rightarrow 'cls$ **and**

trail :: $'st \Rightarrow ('v,$ *nat*, $'v$ *clause*) *marked-lits* **and**
hd-raw-trail :: $'st \Rightarrow ('v,$ *nat*, $'cls)$ *marked-lit* **and**
raw-init-clss :: $'st \Rightarrow 'clss$ **and**
raw-learned-clss :: $'st \Rightarrow 'clss$ **and**
backtrack-lvl :: $'st \Rightarrow$ *nat* **and**
raw-conflicting :: $'st \Rightarrow 'ccls$ *option* **and**

cons-trail :: $('v,$ *nat*, $'cls)$ *marked-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-init-cls :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
add-learned-cls :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
remove-cls :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
update-backtrack-lvl :: *nat* $\Rightarrow 'st \Rightarrow 'st$ **and**
update-conflicting :: $'ccls$ *option* $\Rightarrow 'st \Rightarrow 'st$ **and**

init-state :: $'clss \Rightarrow 'st$ **and**

restart-state :: *'st* ⇒ *'st* +
fixes *f* :: *nat* ⇒ *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:
($cdcl_W$*-merge-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* ⟹ $cdcl_W$*-merge-with-restart* (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1* $cdcl_W$*-merge-stgy S T* ⟹ $cdcl_W$*-merge-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma $cdcl_W$*-merge-with-restart S T* ⟹ $cdcl_W$*-merge-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-*$cdcl_W$:
$cdcl_W$*-merge-with-restart S T* ⟹ $cdcl_W$** (*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*:
$cdcl_W$*-merge-with-restart S T* ⟹ *snd T* = *1* + *snd S*
by (*induction rule*: $cdcl_W$*-merge-with-restart.induct*) *auto*

lemma *full1* $cdcl_W$*-merge-stgy S T* ⟹ $cdcl_W$*-merge-with-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-mset* (*learned-clss S*) ⊆ *simple-clss* (*atms-of-mm* (*init-clss S*))
proof
fix *C*
assume *C*: *C* ∈ *set-mset* (*learned-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 ¬*tautology C*
using *C inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-learned-clause-def* **by** *auto*
moreover
have *atms-of C* ⊆ *atms-of-mm* (*learned-clss S*)
using *C* **by** *auto*
then have *atms-of C* ⊆ *atms-of-mm* (*init-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* ∈ *simple-clss* (*atms-of-mm* (*init-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*› **by** (*metis clauses-restart distinct-mset-union fstI*
mset-le-exists-conv restart.cases state-eq-clauses)
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* $\Longrightarrow$ *cdcl$_W$-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma *cdcl$_W$-with-restart-rtranclp-cdcl$_W$*:
cdcl$_W$-with-restart S T $\Longrightarrow$ *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*› **by** (*metis clauses-restart distinct-mset-union fstI mset-le-exists-conv restart.cases state-eq-clauses*)
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 (λn. 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*: $\exists ka.\ 2 \hat{}\ k - 1 = 2 \hat{}\ ka - 1$
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: $2 \hat{}\ (k - Suc\ 0) \leq i$ **and**
k': $i < 2 \hat{}\ k' - Suc\ 0$ **and**
k-k': $k > k'$
shows *False*
proof −
have $2 \hat{}\ k' - Suc\ 0 < 2 \hat{}\ (k - Suc\ 0)$
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: $2 \hat{}\ (k - 1) \leq i$ **and**
i-k: $i < 2\hat{}\ k - 1$
shows *luby-sequence-core i* = *luby-sequence-core* $(i - 2 \hat{}\ (k - 1) + 1)$
proof −
have *H*: $\neg\ (\exists ka.\ i = 2 \hat{}\ ka - 1)$
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *k'::nat* **where** *k'*: $i = 2 \hat{}\ k' - 1$ **by** *blast*
have $(2::nat) \hat{}\ k' - 1 < 2 \hat{}\ k - 1$
using *i-k* **unfolding** *k'* **.**
then have $(2::nat) \hat{}\ k' < 2 \hat{}\ k$
by *linarith*
then have $k' < k$
by *simp*
have $2 \hat{}\ (k - 1) \leq 2 \hat{}\ k' - (1::nat)$
using *k-i* **unfolding** *k'* **.**
then have $(2::nat) \hat{}\ (k-1) < 2 \hat{}\ k'$
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 $\bigwedge k\ k'.\ 2\ \hat{}\ (k - Suc\ 0) \leq i \Longrightarrow i < 2\ \hat{}\ k - Suc\ 0 \Longrightarrow 2\ \hat{}\ (k' - Suc\ 0) \leq i \Longrightarrow$
$i < 2\ \hat{}\ k' - Suc\ 0 \Longrightarrow k = k'$
by (*meson different-luby-decomposition-false linorder-neqE-nat*)
then have k: ($SOME\ k.\ 2\ \hat{}\ (k - Suc\ 0) \leq i \wedge i < 2\ \hat{}\ k - Suc\ 0) = k$
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* ^ (*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 clauses:
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:
mset-ccls union-ccls insert-ccls remove-clit

— conversion
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
— changing state:
cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl update-conflicting

— get state:
init-state
restart-state
for
ur :: *nat* **and**
mset-cls:: ′*cls* ⇒ ′*v clause* **and**
insert-cls :: ′*v literal* ⇒ ′*cls* ⇒ ′*cls* **and**
remove-lit :: ′*v literal* ⇒ ′*cls* ⇒ ′*cls* **and**

mset-clss:: ′*clss* ⇒ ′*v clauses* **and**
union-clss :: ′*clss* ⇒ ′*clss* ⇒ ′*clss* **and**
in-clss :: ′*cls* ⇒ ′*clss* ⇒ *bool* **and**
insert-clss :: ′*cls* ⇒ ′*clss* ⇒ ′*clss* **and**
remove-from-clss :: ′*cls* ⇒ ′*clss* ⇒ ′*clss* **and**

mset-ccls:: ′*ccls* ⇒ ′*v clause* **and**
union-ccls :: ′*ccls* ⇒ ′*ccls* ⇒ ′*ccls* **and**
insert-ccls :: ′*v literal* ⇒ ′*ccls* ⇒ ′*ccls* **and**
remove-clit :: ′*v literal* ⇒ ′*ccls* ⇒ ′*ccls* **and**

ccls-of-cls :: ′*cls* ⇒ ′*ccls* **and**
cls-of-ccls :: ′*ccls* ⇒ ′*cls* **and**

trail :: *'st* ⇒ (*'v*, *nat*, *'v clause*) *marked-lits* **and**
hd-raw-trail :: *'st* ⇒ (*'v*, *nat*, *'cls*) *marked-lit* **and**
raw-init-clss :: *'st* ⇒ *'clss* **and**
raw-learned-clss :: *'st* ⇒ *'clss* **and**
backtrack-lvl :: *'st* ⇒ *nat* **and**
raw-conflicting :: *'st* ⇒ *'ccls option* **and**

cons-trail :: (*'v*, *nat*, *'cls*) *marked-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-init-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
add-learned-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'ccls option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'clss* ⇒ *'st* **and**
restart-state :: *'st* ⇒ *'st*
begin

sublocale $cdcl_W$*-restart* - *luby-sequence*
apply *unfold-locales*
using *bounded-luby-sequence* **by** *blast*

end
end
theory *CDCL-WNOT*
imports *CDCL-NOT CDCL-W-Termination CDCL-W-Merge*
begin

22 Link between Weidenbach's and NOT's CDCL

22.1 Inclusion of the states

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

fun *convert-marked-lit-from-W* **where**
convert-marked-lit-from-W (*Propagated L -*) = *Propagated L* () |
convert-marked-lit-from-W (*Marked L -*) = *Marked L* ()

abbreviation *convert-trail-from-W* ::
(*'v*, *'lvl*, *'a*) *marked-lit list*
⇒ (*'v*, *unit*, *unit*) *marked-lit list* **where**
convert-trail-from-W ≡ *map convert-marked-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*: *marked-lit-list-induct*) *simp-all*

lemma *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-marked-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 $\models$*as C* $\longleftrightarrow$ *M* $\models$*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* $\longleftrightarrow$ *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-marked-lit-from-NOT*
:: (*'a*, *'e*, *'b*) *marked-lit* $\Rightarrow$ (*'a*, *nat*, *'cls*) *marked-lit* **where**
convert-marked-lit-from-NOT (*Propagated L -*) = *Propagated L dummy-cls* |
convert-marked-lit-from-NOT (*Marked L -*) = *Marked L 0*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT $\equiv$ *map convert-marked-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*: *marked-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*: *marked-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*: *marked-lit-list-induct*) *auto*

lemma *convert-trail-from-W-convert-lit-from-NOT*[*simp*]:
convert-marked-lit-from-W (*convert-marked-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-marked-lit-from-NOT*[*iff*]:
lit-of (*convert-marked-lit-from-NOT L*) = *lit-of L*
by (*cases L*) *auto*

sublocale $state_W \subseteq$ *dpll-state-ops*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
λS. *convert-trail-from-W* (*trail S*)
raw-clauses
λL *S*. *cons-trail* (*convert-marked-lit-from-NOT L*) *S*
λS. *tl-trail S*
λC *S*. *add-learned-cls C S*

$\lambda C\ S.\ remove\text{-}cls\ C\ S$
by *unfold-locales*

context $state_W$
begin
lemma *convert-marked-lit-from-W-convert-marked-lit-from-NOT*[*simp*]:
convert-marked-lit-from-W (*mmset-of-mlit* (*convert-marked-lit-from-NOT L*)) = *L*
by (*cases L*) *auto*
end

sublocale $state_W \subseteq$ *dpll-state*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail S*)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-marked-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*
by *unfold-locales* (*auto simp*: *map-tl o-def*)

context $state_W$
begin
declare $state\text{-}simp_{NOT}$[*simp del*]

end

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}ops$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail S*)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-marked-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*. *raw-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 $cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}proxy.axioms$
sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}proxy$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail S*)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-marked-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. *raw-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› *Marked-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' @ *Marked 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 ¬ *tautology* (C')
using ‹$F \models$*as CNot* C'› *consistent-CNot-not-tautology true-annots-true-cls* **by** *blast*
then have ¬ *tautology* (*?C'* + {#L#})
using ‹$F \models$*as CNot* C'› ‹*undefined-lit* F L› **by** (*metis CNot-remdups-mset*
Marked-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* ∈ *atms-of-mm* (*clauses S*)
∪ *atm-of* ‘ *lits-of-l* (*convert-trail-from-W* (*trail S*))
using ‹*convert-trail-from-W* (*trail S*) = F' @ *Marked K* () # F›
‹*atm-of L* ∈ *atms-of-mm* (*clauses S*) ∪ *atm-of* ‘ *lits-of-l* (F' @ *Marked K* () # F)› **by** *auto*
have *f4*: *clauses S* $\models$*pm remdups-mset* C' + {#L#}
by (*metis* (*no-types*) ‹$L \notin\# C'$› ‹*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'›)
obtain D **where** D: *mset-cls D* = *remdups-mset* C' + {#L#}
using *ex-mset-cls* **by** *blast*
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 D* **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* - - - - - - - -
  $\lambda S.$ *convert-trail-from-W* (*trail S*)
  raw-clauses
  $\lambda L\ S.$ *cons-trail* (*convert-marked-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*. *raw-conflicting S* = *None* *backjump-l-cond inv*$_{NOT}$
  by *unfold-locales*

sublocale *conflict-driven-clause-learning*$_W$ ⊆ *cdcl*$_{NOT}$*-merge-bj-learn* - - - - - - - -
  $\lambda S.$ *convert-trail-from-W* (*trail S*)
  raw-clauses
  $\lambda L\ S.$ *cons-trail* (*convert-marked-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*
  backjump-l-cond
  λ- -. *True*
  λ- *S*. *raw-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*)

22.2 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*))
    ∨ *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*) ⟹
  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-marked 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 dest*!:
hd-raw-trail)
qed
qed

22.3 More lemmas conflict–propagate and backjumping

22.4 CDCL FW

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*
$\vee$ (*no-step cdcl*$_W$*-merge T* $\wedge$ *conflicting T* $\neq$ *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*#} $\in$# *clauses S* **and**
M-C: *M* $\models$*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 raw-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* (*Marked L* (*Suc* (*backtrack-lvl S*)))
(*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 raw-clauses-def*
apply *auto*[]
using *T undef-L* **unfolding** *state-eq-def state-eq*$_{NOT}$*-def* **by** (*auto simp*: *raw-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* !∈! *raw-learned-clss S* **and**
¬(*trail S*) ⊨*asm clauses S* **and**
mset-cls C ∉ *set* (*get-all-mark-of-propagated* (*trail S*)) **and**
C-init: *mset-cls C* ∉# *init-clss S* **and**
T: *T* ∼ *remove-cls C S*
by (*auto elim*: *forgetE*)
have *init-clss S* ⊨*pm mset-cls C*
using *inv C-le* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-learned-clause-def raw-clauses-def*
by (*meson in-clss-mset-clss true-clss-clss-in-imp-true-clss-cls*)
then have *S-C*: *removeAll-mset* (*mset-cls C*) (*clauses S*) ⊨*pm mset-cls C*
using *C-init C-le* **unfolding** *raw-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*: *raw-clauses-def*)
using *T C-le C-init* **by** (*auto*
simp: *state-eq-def Un-Diff state-eq*$_{NOT}$*-def raw-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: *raw-conflicting T* = *Some CT* **and**
CT: *mset-ccls CT* = *mset-cls* C_S **and**
C_S: C_S !∈! *raw-clauses S* **and**
tr-S-C_S: *trail S* ⊨*as CNot* (*mset-cls* 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* $\neq$ *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′: *raw-conflicting T′* = *Some D* **and**
      LD: *L* ∈# *mset-ccls D* **and**
      M1-M2:(*Marked K* (*i*+*1*) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail T′*)) **and**
      get-level (*trail T′*) *L* = *backtrack-lvl T′* **and**
      get-level (*trail T′*) *L* = *get-maximum-level* (*trail T′*) (*mset-ccls D*) **and**
      get-maximum-level (*trail T′*) (*mset-ccls* (*remove-clit L D*)) = *i* **and**
      undef-L: *undefined-lit M1 L* **and**
      U: *U* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
              (*reduce-trail-to M1*
                  (*add-learned-cls* (*cls-of-ccls D*)
                    (*update-backtrack-lvl i*
                      (*update-conflicting None T′*))))
      using *bt* **by** (*auto elim*: *backtrack-levE*)
    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*
raw-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′* @ *Marked K* (*i*+*1*) # *tl* (*trail U*) **and**
tr-U: *trail U* = *Propagated L* (*mset-ccls D*) # *tl* (*trail U*)
using *U M1-M2 undef-L 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″* @ *Marked K* (*i*+*1*) # *tl* (*trail U*)
using *tr-T tr-T′ confl* **unfolding** *M″-def* **by** (*auto elim*: *rulesE*)
have *init-clss T′* + *learned-clss S* ⊨*pm mset-ccls D*
using *inv-T′ confl-T′* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-learned-clause-def*
raw-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* @ [*Marked K* (*Suc i*)]])
using *confl M1-M2* ‹*trail T* = *M* @ *trail T′*›
apply (*auto dest*!: *get-all-marked-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*) ⊨*as CNot* (*remove1-mset L* (*mset-ccls D*))
by (*metis append-self-conv2 tr-U*)
thm *backjump-l*[*of - - - - - L cls-of-ccls D - remove1-mset L* (*mset-ccls D*)]
have *backjump-l S U*
apply (*rule backjump-l*[*of - - - - - L cls-of-ccls D - remove1-mset L* (*mset-ccls 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 undef-L M1-M2 inv-T′ inv undef-L* **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 *U undef-L M1-M2 inv-T′ inv* **unfolding** *cdcl$_W$-all-struct-inv-def cdcl$_W$-M-level-inv-def* **apply** *auto*[]
using *undef-L atm-L* **apply** (*simp add*: *trail-reduce-trail-to$_{NOT}$-add-learned-cls*)
using ‹*init-clss T′ + learned-clss S ⊨pm mset-ccls D*› *LD* **unfolding** *raw-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 ≡ 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* ∨ (*no-step cdcl$_W$-merge T* ∧ *conflicting T* ≠ *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* ∨ (*no-step cdcl$_W$-merge T* ∧ *conflicting T* ≠ *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 ⟶ 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$_{NOT}$ ≤ 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 ⇒ nat* **where**
μ_{FW} S ≡ (*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*) ⊆ *atms-of-mm ?A*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def no-strange-atm-def raw-clauses-def* **by** *auto*
have *atm-trail*: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-mm ?A*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def no-strange-atm-def raw-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* ∧ $cdcl_W$*-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*)

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* ⟷ *full* $cdcl_W$*-merge-stgy R V* (**is** *?s'* ⟷ *?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* ∨ $cdcl_W$*-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* ≠ *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**
inv: $cdcl_W$*-all-struct-inv R*
shows *full* $cdcl_W$*-stgy R V* ⟷ *full* $cdcl_W$*-merge-stgy R V*
by (*simp add*: *assms*(*1*) *full-*$cdcl_W$*-s'-full-*$cdcl_W$*-merge-restart full-*$cdcl_W$*-stgy-iff-full-*$cdcl_W$*-s' inv*)

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* (*mset-clss N*)
shows (*conflicting S'* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*mset-clss N*)))
∨ (*conflicting S'* = *None* ∧ *trail S'* ⊨*asm mset-clss N* ∧ *satisfiable* (*set-mset* (*mset-clss 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-Incremental*
imports *CDCL-W-Termination*
begin

23 Incremental SAT solving

context *conflict-driven-clause-learning*$_W$
begin

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* $\longleftrightarrow$
conflict-is-false-with-level S
$\wedge$ *no-clause-is-false S*
$\wedge$ *no-smaller-confl S*
$\wedge$ *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 $\equiv$ *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 (*Marked L - # M*) *S* =
(*if* $-L \in\#$ *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* $-L \in\#$ *C then S*
else cut-trail-wrt-clause C M (*tl-trail S*))

definition *add-new-clause-and-update* :: *'ccls* ⇒ *'st* ⇒ *'st* **where**
add-new-clause-and-update C S =
(*if trail S* ⊨*as CNot* (*mset-ccls C*)
then update-conflicting (*Some C*) (*add-init-cls* (*cls-of-ccls C*)
(*cut-trail-wrt-clause* (*mset-ccls C*) (*trail S*) *S*))
else add-init-cls (*cls-of-ccls 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*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** *simp*
next
case (*marked L 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 (*proped 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-marked*:

assumes
backtrack-lvl T = *length* (*get-all-levels-of-marked* (*trail T*))
shows
backtrack-lvl (*cut-trail-wrt-clause C* (*trail T*) *T*) =
length (*get-all-levels-of-marked* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)))
using *assms*
proof (*induction trail T arbitrary*:*T rule*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** *simp*
next
case (*marked L 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 (*proped 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-get-all-levels-of-marked*:
assumes *get-all-levels-of-marked* (*trail T*) = *rev* [*Suc 0*..<
Suc (*length* (*get-all-levels-of-marked* (*trail T*)))]
shows
get-all-levels-of-marked (*trail* ((*cut-trail-wrt-clause C* (*trail T*) *T*))) = *rev* [*Suc 0*..<
Suc (*length* (*get-all-levels-of-marked* (*trail* ((*cut-trail-wrt-clause C* (*trail T*) *T*)))))]
using *assms*
proof (*induction trail T arbitrary*:*T rule*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** *simp*
next
case (*marked L 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** (*cases count C L* = *0*) *auto*
next
case (*proped L l M*) **note** *IH* = *this*(*1*)[*of tl-trail T*] **and** *M* = *this*(*2*)[*symmetric*] **and** *bt* = *this*(*3*)
then show *?case* **by** (*cases count C L* = *0*) *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*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** *simp*
next
case (*marked L 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 literal multiset* **where**
f6: (*mma* ∈ {{#− *l*#} |*l*. *l* ∈# *C*} ⟶ *M* ⊨*a mma*) ⟶ *M* ⊨*as* {{#− *l*#} |*l*. *l* ∈# *C*}
using *true-annots-def* **by** *blast*
have *mma* ∈ {{#− *l*#} |*l*. *l* ∈# *C*} ⟶ *trail T* ⊨*a mma*
using *CNot-def M bt* **by** (*metis* (*no-types*) *true-annots-def*)
then have *M* ⊨*as* {{#− *l*#} |*l*. *l* ∈# *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 (*proped 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 literal multiset* **where**
f6: (*mma* ∈ {{#− *l*#} |*l*. *l* ∈# *C*} ⟶ *M* ⊨*a mma*) ⟶ *M* ⊨*as* {{#− *l*#} |*l*. *l* ∈# *C*}
using *true-annots-def* **by** *blast*
have *mma* ∈ {{#− *l*#} |*l*. *l* ∈# *C*} ⟶ *trail T* ⊨*a mma*
using *CNot-def M bt* **by** (*metis* (*no-types*) *true-annots-def*)
then have *M* ⊨*as* {{#− *l*#} |*l*. *l* ∈# *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*:
((∀ *L* ∈#*C*. −*L* ∉ *lits-of-l* (*trail T*)) ∧ *trail* (*cut-trail-wrt-clause C* (*trail T*) *T*) = [])
∨ (−*lit-of* (*hd* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))) ∈# *C*
∧ *length* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)) ≥ *1*)
using *assms*
proof (*induction trail T arbitrary*:*T rule*: *marked-lit-list-induct*)
case *nil*
then show *?case* **by** *simp*
next
case (*marked L 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 (*proped 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 ⊨*asm init-clss S* ⟹ *distinct-mset* (*mset-ccls C*) ⟹ *conflicting S* = *None* ⟹
trail S ⊨*as CNot* (*mset-ccls C*) ⟹
full $cdcl_W$*-stgy*
(*update-conflicting* (*Some C*)

($add\text{-}init\text{-}cls$ ($cls\text{-}of\text{-}ccls$ C) ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ S) S))) T $\Longrightarrow$
$incremental\text{-}cdcl_W$ S T |
add-no-confl:
$trail$ S $\models asm$ $init\text{-}clss$ S $\Longrightarrow$ $distinct\text{-}mset$ ($mset\text{-}ccls$ C) $\Longrightarrow$ $conflicting$ S = $None$ $\Longrightarrow$
$\neg trail$ S $\models as$ $CNot$ ($mset\text{-}ccls$ C) $\Longrightarrow$
$full$ $cdcl_W\text{-}stgy$ ($add\text{-}init\text{-}cls$ ($cls\text{-}of\text{-}ccls$ C) S) T $\Longrightarrow$
$incremental\text{-}cdcl_W$ S T

lemma $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}add\text{-}new\text{-}clause\text{-}and\text{-}update\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv$:
assumes
inv-T: $cdcl_W\text{-}all\text{-}struct\text{-}inv$ T **and**
tr-T-N[*simp*]: $trail$ T $\models asm$ N **and**
tr-C[*simp*]: $trail$ T $\models as$ $CNot$ ($mset\text{-}ccls$ C) **and**
[*simp*]: $distinct\text{-}mset$ ($mset\text{-}ccls$ C)
shows $cdcl_W\text{-}all\text{-}struct\text{-}inv$ ($add\text{-}new\text{-}clause\text{-}and\text{-}update$ C T) (**is** $cdcl_W\text{-}all\text{-}struct\text{-}inv$ $?T'$)
proof –
let $?T$ = $update\text{-}conflicting$ ($Some$ C)
($add\text{-}init\text{-}cls$ ($cls\text{-}of\text{-}ccls$ C) ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T))
obtain M **where**
M: $trail$ T = M @ $trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T)
using *trail-cut-trail-wrt-clause*[*of T mset-ccls C*] **by** *blast*
have *H*[*dest*]: $\bigwedge x.$ $x \in lits\text{-}of\text{-}l$ ($trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T)) $\Longrightarrow$
$x \in lits\text{-}of\text{-}l$ ($trail$ T)
using *inv-T arg-cong*[*OF M*, *of lits-of-l*] **by** *auto*
have *H'*[*dest*]: $\bigwedge x.$ $x \in set$ ($trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T)) $\Longrightarrow$
$x \in set$ ($trail$ T)
using *inv-T arg-cong*[*OF M*, *of set*] **by** *auto*

have *H-proped*:$\bigwedge x.$ $x \in set$ ($get\text{-}all\text{-}mark\text{-}of\text{-}propagated$ ($trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C)
($trail$ T) T))) $\Longrightarrow$ $x \in set$ ($get\text{-}all\text{-}mark\text{-}of\text{-}propagated$ ($trail$ T))
using *inv-T arg-cong*[*OF M*, *of get-all-mark-of-propagated*] **by** *auto*

have [*simp*]: $no\text{-}strange\text{-}atm$ $?T$
using *inv-T* **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ *no-strange-atm-def add-new-clause-and-update-def*
$cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** (*auto 20 1*)
have *M-lev*: $cdcl_W\text{-}M\text{-}level\text{-}inv$ T
using *inv-T* **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ **by** *blast*
then have $no\text{-}dup$ (M @ $trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T))
unfolding $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **unfolding** *M*[*symmetric*] **by** *auto*
then have [*simp*]: $no\text{-}dup$ ($trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T))
by *auto*

have $consistent\text{-}interp$ ($lits\text{-}of\text{-}l$ (M @ $trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C) ($trail$ T) T)))
using *M-lev* **unfolding** $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **unfolding** *M*[*symmetric*] **by** *auto*
then have [*simp*]: $consistent\text{-}interp$ ($lits\text{-}of\text{-}l$ ($trail$ ($cut\text{-}trail\text{-}wrt\text{-}clause$ ($mset\text{-}ccls$ C)
($trail$ T) T)))
unfolding *consistent-interp-def* **by** *auto*

have [*simp*]: $cdcl_W\text{-}M\text{-}level\text{-}inv$ $?T$
using *M-lev cut-trail-wrt-clause-get-all-levels-of-marked*[*of T mset-ccls C*]
unfolding $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** (*auto dest*: *H H'*
simp: *M-lev* $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ *cut-trail-wrt-clause-backtrack-lvl-length-marked*)

have [*simp*]: $\bigwedge s.$ $s \in\#$ $learned\text{-}clss$ T $\Longrightarrow$ $\neg tautology$ s
using *inv-T* **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ **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* (*mset-ccls 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-marked-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-marked-decomposition* (*trail ?T*))
unfolding *all-decomposition-implies-def*
proof *clarify*
fix *a b*
assume (*a*, *b*) $\in$ *set* (*get-all-marked-decomposition* (*trail ?T*))
from *in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend*[*OF this*, *of M*]
obtain *b'* **where**
(*a*, *b'* @ *b*) $\in$ *set* (*get-all-marked-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*: *raw-clauses-def*)
show *?thesis*
using ‹*all-decomposition-implies-m* (*init-clss ?T*)
(*get-all-marked-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* (*mset-ccls C*) **and**
[*simp*]: *distinct-mset* (*mset-ccls 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* (*mset-ccls 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*) ⊨*as CNot* (*mset-ccls 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* (*mset-ccls C*) (*trail T*) *T*)
using *trail-cut-trail-wrt-clause* **by** *blast*
consider
(*false*) ∀ *L*∈#*mset-ccls C*. − *L* ∉ *lits-of-l* (*trail T*) **and**
trail (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*) = []
| (*not-false*)
− *lit-of* (*hd* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*))) ∈# (*mset-ccls C*) **and**
1 ≤ *length* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*))
using *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*[*of mset-ccls C T*] **by** *auto*
then show *?thesis*
proof *cases*
case *false* **note** *C* = *this*(*1*) **and** *empty-tr* = *this*(*2*)
then have [*simp*]: *mset-ccls 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* (*mset-ccls C*) (*trail T*) *T*)))
have *get-all-levels-of-marked* (*trail* (*add-new-clause-and-update C T*)) =
rev [*1*..<*1* + *length* (*get-all-levels-of-marked* (*trail* (*add-new-clause-and-update C T*)))]
using ‹*cdcl*$_W$*-all-struct-inv ?T'*› **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-M-level-inv-def*
by *blast*
moreover
have *backtrack-lvl* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*) =
length (*get-all-levels-of-marked* (*trail* (*add-new-clause-and-update C 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*)
moreover
have *no-dup* (*trail* (*cut-trail-wrt-clause* (*mset-ccls 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*)
then have *atm-of ?L* ∉ *atm-of* ' *lits-of-l*
(*tl* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*)))
by (*cases trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*))
(*auto simp*: *lits-of-def*)

ultimately have *L*: *get-level* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*)) (−*?L*)
= *length* (*get-all-levels-of-marked* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*)))
using *get-level-get-rev-level-get-all-levels-of-marked*[*OF*
‹*atm-of ?L* ∉ *atm-of* ' *lits-of-l* (*tl* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*)
(*trail T*) *T*)))›,
of [*hd* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*))]]

apply (*cases trail* (*add-init-cls* (*cls-of-ccls C*)
(*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*));
cases hd (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*)))
using *l* **by** (*auto split*: *if-split-asm*
simp:*rev-swap*[*symmetric*] *add-new-clause-and-update-def*)

have *L'*: *length* (*get-all-levels-of-marked* (*trail* (*cut-trail-wrt-clause* (*mset-ccls C*)
(*trail T*) *T*)))
= *backtrack-lvl* (*cut-trail-wrt-clause* (*mset-ccls 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* (*cls-of-ccls C*) (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail T*) *T*)))
unfolding *no-smaller-confl-def*
proof (*clarify*, *goal-cases*)
case (*1 M K i M' D*)
then consider
(*DC*) *D* = *mset-ccls C*
| (*D-T*) *D* ∈# *clauses T*
by (*auto simp*: *raw-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'*) @ *Marked K i* # *M* = *trail T*
using *MT 1*(*1*) **by** *auto*
thus *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*) ∈ *atm-of* ‘ (*lits-of-l M*)
using *1*(*3*) *C in-CNot-implies-uminus*(*2*) **by** *blast*
moreover
have *lit-of* (*hd* (*M'* @ *Marked K i* # [])) = −*?L*
using *l 1*(*1*)[*symmetric*] *inv*
by (*cases trail* (*add-init-cls* (*cls-of-ccls C*)
(*cut-trail-wrt-clause* (*mset-ccls 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*) ∈ *atm-of* ‘ (*lits-of-l* (*M'* @ *Marked K i* # []))
by (*cases* (*M'* @ *Marked K i* # [])) *auto*
moreover have *no-dup* (*trail* (*cut-trail-wrt-clause* (*mset-ccls 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*
unfolding $cdcl_W$*-all-struct-inv-def* **by** (*auto simp*: *add-new-clause-and-update-def*)
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* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S*))
    ∨ *conflicting T* = *None* ∧ *trail T* ⊨*asm init-clss S* ∧ *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* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss T*))
    ∨ *conflicting T* = *None* ∧ *trail T* ⊨*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* (*cls-of-ccls C*)
    (*cut-trail-wrt-clause* (*mset-ccls 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* (*cls-of-ccls C*) *S*)
using *inv* ‹*distinct-mset* (*mset-ccls 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 raw-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*: $\forall M.\ (\exists K\ i\ M'.\ trail\ S = M'\ @\ Marked\ K\ i\ \#\ M) \longrightarrow \neg\ M \models as\ CNot\ (mset\text{-}ccls\ C)$
using ‹$\neg$ *trail S* $\models$*as CNot* (*mset-ccls C*)›
by (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model*)

have *cdcl*$_W$*-stgy-invariant* (*add-init-cls* (*cls-of-ccls C*) *S*)
using *s-inv* ‹$\neg$ *trail S* $\models$*as CNot* (*mset-ccls 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*: *raw-clauses-def nc*)
then show *?case*
by (*metis* ‹*cdcl*$_W$*-all-struct-inv* (*add-init-cls* (*cls-of-ccls 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* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss T*))
$\vee$ *conflicting T* = *None* $\wedge$ *trail T* $\models$*asm init-clss T* $\wedge$ *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

24 2-Watched-Literal

theory *CDCL-Two-Watched-Literals*
imports *CDCL-WNOT*
begin

We will directly on the two-watched literals datastructure 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.

24.1 Datastructure 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*, *nat*, *'v twl-clause*) *marked-lit list*)
(*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*, *nat*, *'v twl-clause*) *marked-lit* ⇒ (*'v*, *nat*, *'v clause*) *marked-lit*
where

mmset-of-mlit′ (*Propagated L C*) = *Propagated L* (*mset* (*watched C* @ *unwatched C*)) |
mmset-of-mlit′ (*Marked L i*) = *Marked L i*

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*

abbreviation *raw-clss* :: ′*v twl-state* ⇒ ′*v clauses* **where**
  raw-clss S ≡ *clauses-of-l* (*map raw-clause* (*raw-init-clss S* @ *raw-learned-clss S*))

interpretation *raw-cls*
  λ*C*. *mset* (*raw-clause C*)
  λ*L C*. *TWL-Clause* (*watched C*) (*L* # *unwatched C*)
  λ*L C*. *TWL-Clause* [] (*remove1 L* (*raw-clause C*))
  apply (*unfold-locales*)
  by (*auto simp*:*hd-map comp-def map-tl ac-simps*
    mset-map-mset-remove1-cond ex-mset raw-clause-def
    simp del:)

lemma *XXX*:
  mset (*map* (λ*x*. *mset* (*unwatched x*) + *mset* (*watched x*))
    (*remove1-cond* (λ*D*. *mset* (*raw-clause D*) = *mset* (*raw-clause a*)) *Cs*)) =
   remove1-mset (*mset* (*raw-clause a*)) (*mset* (*map* (λ*x*. *mset* (*raw-clause x*)) *Cs*))
   apply (*induction Cs*)
     apply *simp*
   by (*auto simp*: *ac-simps remove1-mset-single-add raw-clause-def*)

interpretation *raw-clss*
  λ*C*. *mset* (*raw-clause C*)

  λ*L C*. *TWL-Clause* (*watched C*) (*L* # *unwatched C*)
  λ*L C*. *TWL-Clause* [] (*remove1 L* (*raw-clause C*))
  λ*C*. *clauses-of-l* (*map raw-clause C*) *op* @
  λ*L C*. *L* ∈ *set C op* # λ*C*. *remove1-cond* (λ*D*. *mset* (*raw-clause D*) = *mset* (*raw-clause C*))
  apply (*unfold-locales*)
  using *XXX* **by** (*auto simp*:*hd-map comp-def map-tl ac-simps raw-clause-def*
    union-mset-list mset-map-mset-remove1-cond ex-mset
    simp del:)

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

thm *CDCL-Two-Watched-Literals.raw-cls-axioms*
interpretation *twl*: $state_W$*-ops*
$\lambda C.$ *mset* (*raw-clause* C)

$\lambda L\ C.$ *TWL-Clause* (*watched* C) (L # *unwatched* C)
$\lambda L\ C.$ *TWL-Clause* [] (*remove1* L (*raw-clause* C))
$\lambda C.$ *clauses-of-l* (*map raw-clause* C) *op* @
$\lambda L\ C.$ $L \in$ *set* C *op* # $\lambda C.$ *remove1-cond* ($\lambda D.$ *mset* (*raw-clause* D) = *mset* (*raw-clause* C))

mset λ*xs ys. case-prod append* (*fold* (λx (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))
op # *remove1*

raw-clause $\lambda C.$ *TWL-Clause* [] C
trail $\lambda S.$ *hd* (*raw-trail* S)
raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

apply *unfold-locales* **apply** (*auto simp*: *hd-map comp-def map-tl ac-simps raw-clause-def union-mset-list mset-map-mset-remove1-cond ex-mset-unwatched-watched*)
done
declare *CDCL-Two-Watched-Literals.twl.mset-ccls-ccls-of-cls*[*simp del*]

lemma *mmset-of-mlit'-mmset-of-mlit*[*simp*]:
twl.mmset-of-mlit L = *mmset-of-mlit'* L
by (*metis mmset-of-mlit'.simps*(*1*) *mmset-of-mlit'.simps*(*2*) *twl.mmset-of-mlit.elims raw-clause-def*)

definition
candidates-propagate :: *'v twl-state* $\Rightarrow$ (*'v literal* $\times$ *'v twl-clause*) *set*
where
candidates-propagate S =
$\{(L, C) \mid L\ C.$
$C \in$ *set* (*twl.raw-clauses* S) $\wedge$
set (*watched* C) $-$ (*uminus* ` *lits-of-l* (*trail* S)) = $\{L\}$ $\wedge$
undefined-lit (*trail* S) $L\}$

definition *candidates-conflict* :: *'v twl-state* $\Rightarrow$ *'v twl-clause set* **where**
candidates-conflict S =
$\{C.\ C \in$ *set* (*twl.raw-clauses* S) $\wedge$
set (*watched* C) $\subseteq$ *uminus* ` *lits-of-l* (*trail* S)$\}$

primrec (*nonexhaustive*) *index* :: *'a list* $\Rightarrow$*'a* $\Rightarrow$ *nat* **where**
index (*a* # *l*) *c* = (*if a = c then 0 else 1+index l c*)

lemma *index-nth*:
$a \in$ *set* $l \Longrightarrow l$! (*index l a*) = a
by (*induction l*) *auto*

24.2 Invariants

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* it does not get swap with a watched literal L' such that $-\ L'$ is in the trail.

primrec *watched-decided-most-recently* :: ('*v*, '*lvl*, '*mark*) *marked-lit list* ⇒
'*v twl-clause* ⇒ *bool*
where
watched-decided-most-recently M (*TWL-Clause W UW*) ⟷
(∀ *L'*∈ *set W*. ∀ *L*∈ *set UW*.
−*L'* ∈ *lits-of-l M* ⟶ −*L* ∈ *lits-of-l M* ⟶ *L* ∉# *mset W* ⟶
index (*map lit-of M*) (−*L'*) ≤ *index* (*map lit-of M*) (−*L*))

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-twl-cls* :: ('*v*, '*lvl*, '*mark*) *marked-lit list* ⇒ '*v twl-clause* ⇒ *bool* **where**
wf-twl-cls M (*TWL-Clause W UW*) ⟷
distinct W ∧ *length W* ≤ *2* ∧ (*length W* < *2* ⟶ *set UW* ⊆ *set W*) ∧
(∀ *L* ∈ *set W*. −*L* ∈ *lits-of-l M* ⟶ (∀ *L'* ∈ *set UW*. *L'* ∉ *set W* ⟶ −*L'* ∈ *lits-of-l M*)) ∧
watched-decided-most-recently M (*TWL-Clause W UW*)

lemma *size-mset-2*: *size x1* = *2* ⟷ (∃ *a b*. *x1* = {#*a*, *b*#})
apply (*cases x1*)
apply *simp*
by (*metis* (*no-types*, *hide-lams*) *Suc-eq-plus1 one-add-one size-1-singleton-mset*
size-Diff-singleton size-Suc-Diff1 size-single union-single-eq-diff union-single-eq-member)

lemma *distinct-mset-size-2*: *distinct-mset* {#*a*, *b*#} ⟷ *a* ≠ *b*
unfolding *distinct-mset-def* **by** *auto*

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-Clause W UW*
by (*cases C*)
{ **fix** *L L'*
assume
LW: *L* ∈ *set W* **and**
LM: − *L* ∈ *lits-of-l M'* **and**
L'UW: *L'* ∈ *set UW* **and**
L'∉ *set W*
then have
L'M: − *L'* ∈ *lits-of-l M*
using *wf* **by** (*auto simp*: *C M*)

  have *watched-decided-most-recently 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-of-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-of-l* M' **and**
      $- L \in$ *lits-of-l* M' **and**
      $L \notin$ *set W*
    moreover
      have *lit-of l* $\neq - L'$
      using *n-d* **unfolding** *M*
        by (*metis* (*no-types*) *L'M M Marked-Propagated-in-iff-in-lits-of-l defined-lit-map*
          distinct.simps(*2*) *list.simps*(*9*) *set-map*)
    moreover have *watched-decided-most-recently 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

definition *wf-twl-state* :: *'v twl-state* $\Rightarrow$ *bool* **where**
  wf-twl-state S $\longleftrightarrow$ ($\forall$ *C* $\in$ *set* (*twl.raw-clauses S*). *wf-twl-cls* (*trail S*) *C*) $\wedge$ *no-dup* (*trail S*)

lemma *wf-candidates-propagate-sound*:
  assumes *wf*: *wf-twl-state S* **and**
    cand: (*L*, *C*) $\in$ *candidates-propagate S*
  shows *trail S* $\models$*as CNot* (*mset* (*removeAll L* (*raw-clause C*))) $\wedge$ *undefined-lit* (*trail S*) *L*
    (**is** *?Not* $\wedge$ *?undef*)
proof
  def *M* $\equiv$ *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*) $-$ *uminus* ‘ *lits-of-l M* = {*L*}
    undefined-lit M L
    using *cand* **unfolding** *candidates-propagate-def MNU-defs twl.raw-clauses-def* **by** *auto*

  obtain *W UW* **where** *cw-eq*: *C* = *TWL-Clause W UW*

by (*cases C*)

have *l-w*: $L \in set\ W$
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 add*: *twl.raw-clauses-def*)

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 $\forall L' \in set\ (raw\text{-}clause\ C) - \{L\}.\ -L' \in lits\text{-}of\text{-}l\ M$
proof (*cases length* $W < 2$)
case *True*
moreover have $size\ W \neq 0$
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** $set\ UW \subseteq set\ W$
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': $L' \in set\ (raw\text{-}clause\ C) - \{L\}$
have *ex-la*: $\exists La.\ La \neq L \wedge La \in set\ W$
proof (*cases W*)
case *w*: *Nil*
thus *?thesis*
using *l-w* **by** *auto*
next
case *lb*: (*Cons Lb* W')
show *?thesis*
proof (*cases* W')
case *Nil*
thus *?thesis*
using *lb sz2* **by** *simp*
next
case *lc*: (*Cons Lc* W'')
thus *?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*: $La \neq L$ $La \in set\ W$
by *blast*
then have $La \in uminus$ ‘ *lits-of-l M*
using *cw*(*2*)[*unfolded cw-eq*, *simplified*, *folded M-def*] ‹$La \in set\ W$› ‹$La \neq L$› **by** *auto*

then have *nla*: $-La \in$ *lits-of-l M*
by (*auto simp*: *image-image*)
then show $-L' \in$ *lits-of-l M*

proof −
have *f1*: $L' \in$ *set* (*raw-clause C*)
using l' by *blast*
have *f2*: $L' \notin \{L\}$
using l' by *fastforce*
have $\bigwedge l\ L.\ -\ (l::'a\ literal) \in L \vee l \notin uminus\ `\ L$
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*) *M-def* by *blast*
qed

lemma *wf-candidates-propagate-complete*:
assumes *wf*: *wf-twl-state S* and
c-mem: $C \in$ *set* (*twl.raw-clauses S*) and
l-mem: $L \in$ *set* (*raw-clause C*) and
unsat: *trail S* $\models$*as CNot* (*mset-set* (*set* (*raw-clause C*) $-$ $\{L\}$)) and
undef: *undefined-lit* (*trail S*) *L*
shows $(L, C) \in$ *candidates-propagate S*
proof −
def $M \equiv$ *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*

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*
$\bigwedge L\ L'.\ L \in set\ W \Longrightarrow -L \in lits\text{-}of\text{-}l\ M \Longrightarrow L' \in set\ UW \Longrightarrow L' \notin set\ W \Longrightarrow -L' \in lits\text{-}of\text{-}l\ M$
using *wf-c* unfolding *cw-eq* by (*auto simp*: *image-image*)

have *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*
hence *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 ¬ *M* ⊨*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*
thus *?thesis*
using *w-nw*(*2*) *cw-eq l-mem* **by** (*auto simp*: *raw-clause-def*)
next
case (*Cons La W′*)
thus *?thesis*
proof (*cases La* = *L*)
case *True*
thus *?thesis*
using *Cons* **by** *simp*
next
case *False*
have −*La* ∈ *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*: *Marked-Propagated-in-iff-in-lits-of-l raw-clause-def*)
qed
qed
moreover have *L* ∉# *mset-set* (*uminus* ‘ *lits-of-l M*)
using *undef* **by** (*auto simp*: *Marked-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* ∈ *candidates-conflict S*

shows *trail S* $\models$*as CNot* (*mset* (*raw-clause C*)) $\wedge$ *C* $\in$ *set* (*twl.raw-clauses S*)
proof
  def *M* $\equiv$ *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*] **unfolding** *twl.raw-clauses-def* **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 add*: *comp-def twl.raw-clauses-def*)

  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 $\forall$ *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-clause C*)
      show $-$*L* $\in$ *lits-of-l M*
      proof (*cases L* $\in$ *set W*)
        case *True*
        thus *?thesis*
          using *cw*(*2*) *cw-eq* **by** *fastforce*
      next
        case *False*
        thus *?thesis*
          by (*metis* (*no-types*, *hide-lams*) *M-def UnE W' contra-subsetD cw*(*2*) *cw-eq imageE*
            insertI1 l list.set(*2*) *set-append twl-clause.sel*(*1*) *twl-clause.sel*(*2*)
            uminus-of-uminus-id w-nw(*3*) *raw-clause-def*)
      qed
    qed
  qed
  then show *trail S* $\models$*as CNot* (*mset* (*raw-clause C*))
    unfolding *CNot-def true-annots-def* **by** *auto*

show $C \in$ *set* (*twl.raw-clauses S*)
  using *cw* **unfolding** *twl.raw-clauses-def* **by** *auto*
qed

lemma *wf-candidates-conflict-complete*:
  assumes *wf*: *wf-twl-state S* **and**
    c-mem: $C \in$ *set* (*twl.raw-clauses S*) **and**
    unsat: *trail S* $\models$*as CNot* (*mset* (*raw-clause C*))
  shows $C \in$ *candidates-conflict S*
proof −
  def $M \equiv$ *trail S*
  def $N \equiv$ *twl.init-clss S*
  def $U \equiv$ *twl.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 $\bigwedge L.\ L \in$ *set* (*raw-clause C*) $\Longrightarrow -L \in$ *lits-of-l M*
    unfolding *M-def* **using** *unsat*[*unfolded CNot-def true-annots-def*, *simplified*] **by** *auto*
  then have *set* (*raw-clause C*) $\subseteq$ *uminus* ‘ *lits-of-l M*
    by (*metis imageI subsetI uminus-of-uminus-id*)
  then have *set W* $\subseteq$ *uminus* ‘ *lits-of-l M*
    using *cw-eq* **by** (*auto simp*: *raw-clause-def*)
  then have *subset*: *set W* $\subseteq$ *uminus* ‘ *lits-of-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*, *nat*, *'v twl-clause*) *marked-lits*)
    [] [] *0 None* $\in$ {*S*:: *'v twl-state*. *wf-twl-state S*}
    by (*auto simp*: *wf-twl-state-def twl.raw-clauses-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*, *nat*, *'a twl-clause*) *marked-lit list* **where**
raw-trail-twl S ≡ *raw-trail* (*rough-state-of-twl S*)

abbreviation *trail-twl* :: *'a wf-twl* ⇒ (*'a*, *nat*, *'a literal multiset*) *marked-lit list* **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 ≡ *twl.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 *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* ∈ *set* (*raw-clauses-twl S*) **and**
unsat: *trail-twl S* ⊨*as CNot* (*mset* (*raw-clause C*))
shows *C* ∈ *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 ≡
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 ≡
TWL-State (*raw-trail S*) (*raw-init-clss S*) (*raw-learned-clss S*) (*backtrack-lvl S*) *C*

24.3 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**

$rewatch$ :: $'v$ $literal \Rightarrow 'v$ $twl\text{-}state \Rightarrow$
$'v$ $twl\text{-}clause \Rightarrow 'v$ $twl\text{-}clause$ **and**
$restart\text{-}learned$ :: $'v$ $twl\text{-}state \Rightarrow 'v$ $twl\text{-}clause$ $list$
assumes
$clause\text{-}watch$: $no\text{-}dup$ $(trail\ S) \Longrightarrow mset\ (raw\text{-}clause\ (watch\ S\ C)) = mset\ C$ **and**
$wf\text{-}watch$: $no\text{-}dup\ (trail\ S) \Longrightarrow wf\text{-}twl\text{-}cls\ (trail\ S)\ (watch\ S\ C)$ **and**
$clause\text{-}rewatch$: $mset\ (raw\text{-}clause\ (rewatch\ L'\ S\ C')) = mset\ (raw\text{-}clause\ C')$ **and**
$wf\text{-}rewatch$:
$no\text{-}dup\ (trail\ S) \Longrightarrow undefined\text{-}lit\ (trail\ S)\ (lit\text{-}of\ L) \Longrightarrow wf\text{-}twl\text{-}cls\ (trail\ S)\ C' \Longrightarrow$
$wf\text{-}twl\text{-}cls\ (L\ \#\ trail\ S)\ (rewatch\ (lit\text{-}of\ L)\ S\ C')$
and
$restart\text{-}learned$: $mset\ (restart\text{-}learned\ S) \subseteq\#\ mset\ (raw\text{-}learned\text{-}clss\ S)$ — We need *mset* and not *set* to take care of duplicates.
begin

definition
$cons\text{-}trail$ :: $('v,\ nat,\ 'v\ twl\text{-}clause)\ marked\text{-}lit \Rightarrow 'v\ twl\text{-}state \Rightarrow 'v\ twl\text{-}state$
where
$cons\text{-}trail\ L\ S =$
$TWL\text{-}State\ (L\ \#\ raw\text{-}trail\ S)\ (map\ (rewatch\ (lit\text{-}of\ L)\ S)\ (raw\text{-}init\text{-}clss\ S))$
$(map\ (rewatch\ (lit\text{-}of\ L)\ S)\ (raw\text{-}learned\text{-}clss\ S))\ (backtrack\text{-}lvl\ S)\ (raw\text{-}conflicting\ S)$

definition
$add\text{-}init\text{-}cls$ :: $'v\ literal\ list \Rightarrow 'v\ twl\text{-}state \Rightarrow 'v\ twl\text{-}state$
where
$add\text{-}init\text{-}cls\ C\ S =$
$TWL\text{-}State\ (raw\text{-}trail\ S)\ (watch\ S\ C\ \#\ raw\text{-}init\text{-}clss\ S)\ (raw\text{-}learned\text{-}clss\ S)\ (backtrack\text{-}lvl\ S)$
$(raw\text{-}conflicting\ S)$

definition
$add\text{-}learned\text{-}cls$ :: $'v\ literal\ list \Rightarrow 'v\ twl\text{-}state \Rightarrow 'v\ twl\text{-}state$
where
$add\text{-}learned\text{-}cls\ C\ S =$
$TWL\text{-}State\ (raw\text{-}trail\ S)\ (raw\text{-}init\text{-}clss\ S)\ (watch\ S\ C\ \#\ raw\text{-}learned\text{-}clss\ S)\ (backtrack\text{-}lvl\ S)$
$(raw\text{-}conflicting\ S)$

definition
$remove\text{-}cls$ :: $'v\ literal\ list \Rightarrow 'v\ twl\text{-}state \Rightarrow 'v\ twl\text{-}state$
where
$remove\text{-}cls\ C\ S =$
$TWL\text{-}State\ (raw\text{-}trail\ S)$
$(removeAll\text{-}cond\ (\lambda D.\ mset\ (raw\text{-}clause\ D) = mset\ C)\ (raw\text{-}init\text{-}clss\ S))$
$(removeAll\text{-}cond\ (\lambda D.\ mset\ (raw\text{-}clause\ D) = mset\ C)\ (raw\text{-}learned\text{-}clss\ S))$
$(backtrack\text{-}lvl\ S)$
$(raw\text{-}conflicting\ S)$

definition $init\text{-}state$ :: $'v\ literal\ list\ list \Rightarrow 'v\ twl\text{-}state$ **where**
$init\text{-}state\ N = fold\ add\text{-}init\text{-}cls\ N\ (TWL\text{-}State\ []\ []\ []\ 0\ None)$

lemma $unchanged\text{-}fold\text{-}add\text{-}init\text{-}cls$:
$raw\text{-}trail\ (fold\ add\text{-}init\text{-}cls\ Cs\ (TWL\text{-}State\ M\ N\ U\ k\ C)) = M$
$raw\text{-}learned\text{-}clss\ (fold\ add\text{-}init\text{-}cls\ Cs\ (TWL\text{-}State\ M\ N\ U\ k\ C)) = U$
$backtrack\text{-}lvl\ (fold\ add\text{-}init\text{-}cls\ Cs\ (TWL\text{-}State\ M\ N\ U\ k\ C)) = k$
$raw\text{-}conflicting\ (fold\ add\text{-}init\text{-}cls\ Cs\ (TWL\text{-}State\ M\ N\ U\ k\ C)) = C$
by $(induct\ Cs\ arbitrary\text{: }N)\ (auto\ simp\text{: }add\text{-}init\text{-}cls\text{-}def)$

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 *clauses-init-fold-add-init*:
no-dup M $\Longrightarrow$
twl.init-clss (*fold add-init-cls Cs* (*TWL-State M N U k C*)) =
clauses-of-l Cs + *clauses-of-l* (*map raw-clause N*)
by (*induct Cs arbitrary*: *N*) (*auto simp*: *add-init-cls-def clause-watch comp-def ac-simps*)

lemma *init-clss-init-state*[*simp*]: *twl.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

24.4 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* ($\lambda L.\ -L \notin$ *lits-of-l* (*raw-trail S*)) *C′*;
neg-assigned-sorted-by-trail = *filter* ($\lambda L.\ L \in$ *set C*) (*map* ($\lambda L.\ -$*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
$l = [] \Longrightarrow P$ **and**
$\bigwedge x.\ l = [x] \Longrightarrow P$ **and**
$\bigwedge x\ y\ xs.\ l = x\ \#\ y\ \#\ xs \Longrightarrow P$
shows *P*
by (*metis assms list.collapse*)

lemma *filter-in-list-prop-verifiedD*:
assumes [*L*←*P* . *Q L*] = *l*
shows $\forall x \in$ *set l*. $x \in$ *set P* $\wedge$ *Q x*
using *assms* **by** *auto*

lemma *no-dup-filter-diff*:
assumes *n-d*: *no-dup M* **and** *H*: [*L*←*map* ($\lambda L.\ -$ *lit-of L*) *M*. $L \in$ *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: $[L \leftarrow remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)] = l$ **and**
l': $[L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C] = l'$
shows $\forall\, x \in set\ l.\ \forall\, y \in set\ l'.\ x \neq y$
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
$xs \equiv [L \leftarrow remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
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
proof $-$
note *xs-def*[*simp*] **and** *ys-def*[*simp*]
have *dist*: $\bigwedge P.\ distinct\ [L \leftarrow remdups\ C\ .\ P\ L]$
by *auto*
then have H: $\bigwedge a\ b\ P\ xs.\ [L \leftarrow remdups\ C\ .\ P\ L] = a\ \#\ b\ \#\ xs \Longrightarrow a \neq b$
by (*metis distinct-length-2-or-more*)
show *?thesis*
apply (*cases* $[L \leftarrow remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$
rule: *list-cases2*;
cases $[L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$ *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
$xs \equiv [L \leftarrow remdups\ C\ .\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
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
$xs \equiv [L \leftarrow remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
assumes *n-d*: *no-dup* (*raw-trail S*)
shows $set\ C = set\ xs \cup set\ ys$
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 *mset* (*raw-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*)

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*
by (*auto simp*: *image-image comp-def*)

lemma *distinct-filter-eq*:
assumes *distinct xs*
shows $[L \leftarrow xs.\ L = a] = (if\ a \in set\ xs\ then\ [a]\ else\ [])$
using *assms* **by** (*induction xs*) *auto*

lemma *no-dup-distinct-map-uminus-lit-of*:
  no-dup xs $\Longrightarrow$ *distinct* (*map* ($\lambda L.$ $-$ *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* $\equiv$ *filter* ($\lambda L.$ $-L \notin$ *lits-of-l* (*raw-trail S*)) (*remdups C*) **and**
     tr: *neg-assigned-sorted-by-trail* $\equiv$ *filter* ($\lambda L.$ $L \in$ *set C*) (*map* ($\lambda L.$ $-$*lit-of L*) (*raw-trail S*))
   defines
      W: $W \equiv$ *take 2* (*neg-not-assigned* @ *neg-assigned-sorted-by-trail*)
  assumes
    n-d[*simp*]: *no-dup* (*raw-trail S*)
  shows *wf-twl-cls* (*trail S*) (*TWL-Clause W* (*foldr remove1 W C*))
  unfolding *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-of-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-decided-most-recently.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-decided-most-recently.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-decided-most-recently.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* (*trail S*) (*watch-nat S C*)
using *wf-watch-witness*[*of S C*] *watch-nat-def* **by** *metis*

definition
rewatch-nat ::
'v literal $\Rightarrow$ *'v twl-state* $\Rightarrow$ *'v twl-clause* $\Rightarrow$ *'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*
[] $\Rightarrow$ *C*
| L' # - $\Rightarrow$
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 *mset* (*raw-clause* (*rewatch-nat L S C*)) = *mset* (*raw-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*
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* ≤ *2*
  shows *W* = {#} ∨ (∃ *a*. *W* = {#*a*#}) ∨ (∃ *a b*. *W* = {#*a*,*b*#})
  by (*metis One-nat-def Suc-1 Suc-eq-plus1-left assms linorder-not-less nat-less-le*
    not-less-eq-eq le-iff-add size-1-singleton-mset
    size-eq-0-iff-empty size-mset-2)

lemma *filter-sorted-list-of-multiset-eqD*:
  assumes [*x* ← *sorted-list-of-multiset A*. *p x*] = *x* # *xs* (**is** *?comp* = -)
  shows *x* ∈# *A*
proof −
  have *x* ∈ *set ?comp*
    using *assms* **by** *simp*
  then have *x* ∈ *set* (*sorted-list-of-multiset A*)
    by *simp*
  then show *x* ∈# *A*
    by *simp*
qed

lemma *clause-rewatch-witness′*:
  assumes
    wf: *wf-twl-cls* (*trail S*) *C* **and**
    n-d: *no-dup* (*raw-trail S*) **and**
    undef: *undefined-lit* (*trail S*) (*lit-of L*)
  shows *wf-twl-cls* (*L* # *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 n-d undef* **unfolding** *rewatch-nat-def*
    by (*auto simp*: *uminus-lit-swap Marked-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*)
      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 $\bigwedge p\ l.$ *filter p* (*unwatched C*) $\neq$ [] $\vee$ $l \notin$ *set UW* $\vee$ $\neg$ *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*)
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*: [$l \leftarrow$*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 $\neg$ *set UW* $\subseteq$ *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* $\leq$ *Suc 0* $\longleftrightarrow$ *length W* = *0* $\vee$ *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*: $\forall$ *L*$\in$ *set W*. $-$ *L* $\in$ *lits-of-l* (*trail S*) $\longrightarrow$
($\forall$ *L′*$\in$ *set UW*. $L' \notin$ *set W* $\longrightarrow$ $-$ L' $\in$ *lits-of-l* (*trail S*))
using *wf* **by** (*auto simp*: *C*)
have *W*: *length W* $\leq$ *2* **and** *W-UW*: *length W* < *2* $\longrightarrow$ *set UW* $\subseteq$ *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-decided-most-recently.simps Ball-def twl-clause.sel*
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 split*: *if-split-asm*)
using *distinct* **apply** (*fastforce split*: *if-split-asm*)
using *distinct* **apply** (*fastforce split*: *if-split-asm*)
apply (*force dest*: *filter-in-list-prop-verifiedD*)
using *H* **by** (*auto simp*: *uminus-lit-swap*)
next
case *5*
have *H*: $\forall x.\ x \in$ *set W* $\longrightarrow -\ x \in$ *lits-of-l* (*trail S*) $\longrightarrow$ ($\forall x.\ x \in$ *set UW* $\longrightarrow x \notin$ *set W*
$\longrightarrow -\ x \in$ *lits-of-l* (*trail S*))
using *wf* **by** (*auto simp*: *C*)
show *?case*
unfolding *C watched-decided-most-recently.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*: $L' \in$ *set* [$l \leftarrow$*unwatched C*. $l \notin$ *set* (*watched* (*TWL-Clause W UW*))
$\wedge -\ l \notin$ *lits-of-l* (L # *trail S*)]
using *1*(*2*) *5 C* **by** *auto*
moreover then have *f11*: $-\ xW \in$ *lits-of-l* (*trail S*)
using *1*(*3*) *LxW* **by** (*auto simp*: *uminus-lit-swap*)
moreover then have $xW \notin$ *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

25 Invariants for 2 Watched-Literals

theory *CDCL-Two-Watched-Literals-Invariant*
imports *CDCL-Two-Watched-Literals DPLL-CDCL-W-Implementation*
begin

25.1 Interpretation for *conflict-driven-clause-learning*$_W$.*cdcl*$_W$

We define here the 2-WL with the invariant and show the role of the candidates.

context *abstract-twl*
begin

25.1.1 Direct Interpretation

lemma *mset-map-removeAll-cond*:
mset (*map* (λx. *mset* (*raw-clause* *x*))
(*removeAll-cond* (λD. *mset* (*raw-clause* *D*) = *mset* (*raw-clause* *C*)) *N*))
= *mset* (*removeAll* (*mset* (*raw-clause* *C*)) (*map* (λx. *mset* (*raw-clause* *x*)) *N*))
by (*induction* *N*) *auto*

lemma *mset-raw-init-clss-init-state*:
mset (*map* (λx. *mset* (*raw-clause* *x*)) (*raw-init-clss* (*init-state* (*map* *raw-clause* *N*))))
= *mset* (*map* (λx. *mset* (*raw-clause* *x*)) *N*)
by (*metis* (*no-types*, *lifting*) *init-clss-init-state map-eq-conv map-map o-def*)

interpretation *rough-cdcl*: *state*$_W$
λC. *mset* (*raw-clause* *C*)

λL *C*. *TWL-Clause* (*watched* *C*) (*L* # *unwatched* *C*)
λL *C*. *TWL-Clause* [] (*remove1* *L* (*raw-clause* *C*))
λC. *clauses-of-l* (*map* *raw-clause* *C*) *op* @
λL *C*. *L* $\in$ *set* *C* *op* # λC. *remove1-cond* (λD. *mset* (*raw-clause* *D*) = *mset* (*raw-clause* *C*))

mset λxs *ys*. *case-prod append* (*fold* (λx (*ys*, *zs*). (*remove1* *x* *ys*, *x* # *zs*)) *xs* (*ys*, []))
op # *remove1*

raw-clause λC. *TWL-Clause* [] *C*
trail λS. *hd* (*raw-trail* *S*)
raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
cons-trail tl-trail λC. *add-init-cls* (*raw-clause* *C*) λC. *add-learned-cls* (*raw-clause* *C*)
λC. *remove-cls* (*raw-clause* *C*)
update-backtrack-lvl
update-conflicting λN. *init-state* (*map* *raw-clause* *N*) *restart'*
apply *unfold-locales*
apply (*case-tac raw-trail S*)
apply (*simp-all add*: *add-init-cls-def add-learned-cls-def clause-rewatch clause-watch*

cons-trail-def remove-cls-def restart′-def tl-trail-def map-tl comp-def
ac-simps mset-map-removeAll-cond mset-raw-init-clss-init-state)

apply (*auto simp*: *mset-map image-mset-subseteq-mono*[*OF restart-learned*])
done

interpretation *rough-cdcl*: *conflict-driven-clause-learning*$_W$
λC. *mset* (*raw-clause* C)

$\lambda L\ C$. *TWL-Clause* (*watched* C) (L # *unwatched* C)
$\lambda L\ C$. *TWL-Clause* [] (*remove1* L (*raw-clause* C))
λC. *clauses-of-l* (*map raw-clause* C) *op* @
$\lambda L\ C$. $L \in$ *set* C *op* # λC. *remove1-cond* (λD. *mset* (*raw-clause* D) = *mset* (*raw-clause* C))

mset λ*xs ys*. *case-prod append* (*fold* (λx (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))
op # *remove1*

λC. *raw-clause* C λC. *TWL-Clause* [] C
trail λS. *hd* (*raw-trail* S)
raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
cons-trail tl-trail λC. *add-init-cls* (*raw-clause* C) λC. *add-learned-cls* (*raw-clause* C)
λC. *remove-cls* (*raw-clause* C)
update-backtrack-lvl
update-conflicting λN. *init-state* (*map raw-clause* N) *restart′*
by *unfold-locales*

declare *local.rough-cdcl.mset-ccls-ccls-of-cls*[*simp del*]

25.1.2 Opaque Type with Invariant

declare *rough-cdcl.state-simp*[*simp del*]

definition *cons-trail-twl* :: (′*v*, *nat*, ′*v twl-clause*) *marked-lit* ⇒ ′*v wf-twl* ⇒ ′*v wf-twl*
where
cons-trail-twl L S ≡ *twl-of-rough-state* (*cons-trail L* (*rough-state-of-twl S*))

lemma *wf-twl-state-cons-trail*:
assumes
undef: *undefined-lit* (*trail S*) (*lit-of L*) **and**
wf: *wf-twl-state S*
shows *wf-twl-state* (*cons-trail L S*)
using *undef wf wf-rewatch*[*of S mmset-of-mlit′ L*] **unfolding** *wf-twl-state-def Ball-def*
by (*auto simp*: *cons-trail-def defined-lit-map comp-def image-def twl.raw-clauses-def*)

lemma *rough-state-of-twl-cons-trail*:
undefined-lit (*trail-twl S*) (*lit-of L*) ⟹
rough-state-of-twl (*cons-trail-twl L S*) = *cons-trail L* (*rough-state-of-twl S*)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-cons-trail*
unfolding *cons-trail-twl-def* **by** *blast*

abbreviation *add-init-cls-twl* **where**
add-init-cls-twl C S ≡ *twl-of-rough-state* (*add-init-cls C* (*rough-state-of-twl S*))

lemma *wf-twl-add-init-cls*: *wf-twl-state S* ⟹ *wf-twl-state* (*add-init-cls L S*)
unfolding *wf-twl-state-def* **by** (*auto simp*: *wf-watch add-init-cls-def comp-def twl.raw-clauses-def*
split: *if-split-asm*)

lemma *rough-state-of-twl-add-init-cls*:
rough-state-of-twl (*add-init-cls-twl L S*) = *add-init-cls L* (*rough-state-of-twl S*)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-init-cls* **by** *blast*

abbreviation *add-learned-cls-twl* **where**
add-learned-cls-twl C S ≡ *twl-of-rough-state* (*add-learned-cls C* (*rough-state-of-twl S*))

lemma *wf-twl-add-learned-cls*: *wf-twl-state S* ⟹ *wf-twl-state* (*add-learned-cls L S*)
unfolding *wf-twl-state-def* **by** (*auto simp*: *wf-watch add-learned-cls-def twl.raw-clauses-def*
split: *if-split-asm*)

lemma *rough-state-of-twl-add-learned-cls*:
rough-state-of-twl (*add-learned-cls-twl L S*) = *add-learned-cls L* (*rough-state-of-twl S*)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-learned-cls* **by** *blast*

abbreviation *remove-cls-twl* **where**
remove-cls-twl C S ≡ *twl-of-rough-state* (*remove-cls C* (*rough-state-of-twl S*))

lemma *set-removeAll-condD*: $x \in$ *set* (*removeAll-cond f xs*) ⟹ $x \in$ *set xs*
by (*induction xs*) (*auto split*: *if-split-asm*)

lemma *wf-twl-remove-cls*: *wf-twl-state S* ⟹ *wf-twl-state* (*remove-cls L S*)
unfolding *wf-twl-state-def* **by** (*auto simp*: *wf-watch remove-cls-def twl.raw-clauses-def comp-def*
split: *if-split-asm dest*: *set-removeAll-condD*)

lemma *rough-state-of-twl-remove-cls*:
rough-state-of-twl (*remove-cls-twl L S*) = *remove-cls L* (*rough-state-of-twl S*)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-remove-cls* **by** *blast*

abbreviation *init-state-twl* **where**
init-state-twl N ≡ *twl-of-rough-state* (*init-state N*)

lemma *wf-twl-state-wf-twl-state-fold-add-init-cls*:
assumes *wf-twl-state S*
shows *wf-twl-state* (*fold add-init-cls N S*)
using *assms* **apply** (*induction N arbitrary*: *S*)
apply (*auto simp*: *wf-twl-state-def*)[]
by (*simp add*: *wf-twl-add-init-cls*)

lemma *wf-twl-state-epsilon-state*[*simp*]:
wf-twl-state (*TWL-State* [] [] [] *0 None*)
by (*auto simp*: *wf-twl-state-def twl.raw-clauses-def*)

lemma *wf-twl-init-state*: *wf-twl-state* (*init-state N*)
unfolding *init-state-def* **by** (*auto intro*!: *wf-twl-state-wf-twl-state-fold-add-init-cls*)

lemma *rough-state-of-twl-init-state*:
rough-state-of-twl (*init-state-twl N*) = *init-state N*
by (*simp add*: *twl-of-rough-state-inverse wf-twl-init-state*)

abbreviation *tl-trail-twl* **where**
tl-trail-twl S ≡ *twl-of-rough-state* (*tl-trail* (*rough-state-of-twl S*))

lemma *wf-twl-state-tl-trail*: *wf-twl-state S* ⟹ *wf-twl-state* (*tl-trail S*)

by (*auto simp add*: *twl-of-rough-state-inverse wf-twl-init-state wf-twl-cls-wf-twl-cls-tl tl-trail-def wf-twl-state-def distinct-tl map-tl comp-def twl.raw-clauses-def*)

lemma *rough-state-of-twl-tl-trail*:
rough-state-of-twl (*tl-trail-twl* S) = *tl-trail* (*rough-state-of-twl* S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-tl-trail* **by** *blast*

abbreviation *update-backtrack-lvl-twl* **where**
update-backtrack-lvl-twl k S $\equiv$ *twl-of-rough-state* (*update-backtrack-lvl* k (*rough-state-of-twl* S))

lemma *wf-twl-state-update-backtrack-lvl*:
wf-twl-state $S \Longrightarrow$ *wf-twl-state* (*update-backtrack-lvl* k S)
unfolding *wf-twl-state-def* **by** (*auto simp*: *comp-def twl.raw-clauses-def*)

lemma *rough-state-of-twl-update-backtrack-lvl*:
rough-state-of-twl (*update-backtrack-lvl-twl* k S) = *update-backtrack-lvl* k (*rough-state-of-twl* S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-backtrack-lvl* **by** *fast*

abbreviation *update-conflicting-twl* **where**
update-conflicting-twl k S $\equiv$ *twl-of-rough-state* (*update-conflicting* k (*rough-state-of-twl* S))

lemma *wf-twl-state-update-conflicting*:
wf-twl-state $S \Longrightarrow$ *wf-twl-state* (*update-conflicting* k S)
unfolding *wf-twl-state-def* **by** (*auto simp*: *twl.raw-clauses-def comp-def*)

lemma *rough-state-of-twl-update-conflicting*:
rough-state-of-twl (*update-conflicting-twl* k S) = *update-conflicting* k (*rough-state-of-twl* S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-conflicting* **by** *fast*

abbreviation *raw-clauses-twl* **where**
raw-clauses-twl S $\equiv$ *twl.raw-clauses* (*rough-state-of-twl* S)

abbreviation *restart-twl* **where**
restart-twl S $\equiv$ *twl-of-rough-state* (*restart′* (*rough-state-of-twl* S))
lemma *mset-union-mset-setD*:
mset $A \subseteq\#$ *mset* $B \Longrightarrow$ *set* $A \subseteq$ *set* B
by *auto*

lemma *wf-wf-restart′*: *wf-twl-state* $S \Longrightarrow$ *wf-twl-state* (*restart′* S)
unfolding *restart′-def wf-twl-state-def* **apply** *standard*
apply *clarify*
apply (*rename-tac* x)
apply (*subgoal-tac wf-twl-cls* (*trail* S) x)
apply (*case-tac* x)
using *restart-learned* **by** (*auto simp*: *twl.raw-clauses-def comp-def dest*: *mset-union-mset-setD*)

lemma *rough-state-of-twl-restart-twl*:
rough-state-of-twl (*restart-twl* S) = *restart′* (*rough-state-of-twl* S)
by (*simp add*: *twl-of-rough-state-inverse wf-wf-restart′*)

sublocale *conflict-driven-clause-learning*$_W$
λC. *mset* (*raw-clause* C)

$\lambda L\ C.\ TWL\text{-}Clause\ (watched\ C)\ (L\ \#\ unwatched\ C)$
$\lambda L\ C.\ TWL\text{-}Clause\ []\ (remove1\ L\ (raw\text{-}clause\ C))$
$\lambda C.\ clauses\text{-}of\text{-}l\ (map\ raw\text{-}clause\ C)\ op\ @$
$\lambda L\ C.\ L \in set\ C\ op\ \#\ \lambda C.\ remove1\text{-}cond\ (\lambda D.\ mset\ (raw\text{-}clause\ D) = mset\ (raw\text{-}clause\ C))$

$mset\ \lambda xs\ ys.\ case\text{-}prod\ append\ (fold\ (\lambda x\ (ys,\ zs).\ (remove1\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ []))$
$op\ \#\ remove1$

$\lambda C.\ raw\text{-}clause\ C\ \lambda C.\ TWL\text{-}Clause\ []\ C$
trail-twl $\lambda S.\ hd\ (raw\text{-}trail\text{-}twl\ S)$
raw-init-clss-twl
raw-learned-clss-twl
backtrack-lvl-twl
raw-conflicting-twl
cons-trail-twl
tl-trail-twl
$\lambda C.\ add\text{-}init\text{-}cls\text{-}twl\ (raw\text{-}clause\ C)$
$\lambda C.\ add\text{-}learned\text{-}cls\text{-}twl\ (raw\text{-}clause\ C)$
$\lambda C.\ remove\text{-}cls\text{-}twl\ (raw\text{-}clause\ C)$
update-backtrack-lvl-twl
update-conflicting-twl
$\lambda N.\ init\text{-}state\text{-}twl\ (map\ raw\text{-}clause\ N)$
restart-twl
apply *unfold-locales*
using *rough-cdcl.hd-raw-trail* **apply** *blast*
apply (*simp-all add*: *rough-state-of-twl-cons-trail rough-state-of-twl-tl-trail rough-state-of-twl-add-init-cls rough-state-of-twl-add-learned-cls rough-state-of-twl-remove-cls rough-state-of-twl-update-backtrack-lvl rough-state-of-twl-update-conflicting*)[*7*]
using *rough-cdcl.init-clss-cons-trail rough-cdcl.init-clss-tl-trail rough-cdcl.init-clss-add-init-cls rough-cdcl.init-clss-remove-cls rough-cdcl.init-clss-add-learned-cls rough-cdcl.init-clss-update-backtrack-lvl rough-cdcl.init-clss-update-conflicting*
apply (*auto simp add*: *rough-state-of-twl-cons-trail rough-state-of-twl-tl-trail rough-state-of-twl-add-init-cls rough-state-of-twl-add-learned-cls rough-state-of-twl-remove-cls rough-state-of-twl-update-backtrack-lvl rough-state-of-twl-update-conflicting comp-def*)[*7*]
using *rough-cdcl.learned-clss-cons-trail rough-cdcl.learned-clss-tl-trail rough-cdcl.learned-clss-add-init-cls rough-cdcl.learned-clss-remove-cls rough-cdcl.learned-clss-add-learned-cls rough-cdcl.learned-clss-update-backtrack-lvl rough-cdcl.learned-clss-update-conflicting*
apply (*auto simp add*: *rough-state-of-twl-cons-trail rough-state-of-twl-tl-trail rough-state-of-twl-add-init-cls rough-state-of-twl-add-learned-cls rough-state-of-twl-remove-cls rough-state-of-twl-update-backtrack-lvl rough-state-of-twl-update-conflicting comp-def*)[*7*]
apply (*auto simp add*: *rough-state-of-twl-cons-trail rough-state-of-twl-tl-trail rough-state-of-twl-add-init-cls rough-state-of-twl-add-learned-cls rough-state-of-twl-remove-cls rough-state-of-twl-update-backtrack-lvl rough-state-of-twl-update-conflicting comp-def*)[*14*]
using *init-clss-init-state* **apply** (*auto simp*: *rough-state-of-twl-init-state*)[*5*]
using *rough-cdcl.init-clss-restart-state rough-cdcl.learned-clss-restart-state*
apply (*auto simp*: *rough-state-of-twl-restart-twl*)[*5*]
done

declare *local.rough-cdcl.mset-ccls-ccls-of-cls*[*simp del*]
abbreviation *state-eq-twl* (**infix** $\sim$*TWL 51*) **where**
state-eq-twl S S' $\equiv$ *rough-cdcl.state-eq* (*rough-state-of-twl S*) (*rough-state-of-twl S'*)
notation *state-eq* (**infix** $\sim$ *51*)
declare *state-simp*[*simp del*]

To avoid ambiguities:

no-notation *state-eq-twl* (**infix** $\sim$ *51*)

inductive *propagate-twl* :: *'v wf-twl* $\Rightarrow$ *'v wf-twl* $\Rightarrow$ *bool* **where**
propagate-twl-rule: (*L*, *C*) $\in$ *candidates-propagate-twl S* $\Longrightarrow$
  S' $\sim$ *cons-trail-twl* (*Propagated L C*) *S* $\Longrightarrow$
  raw-conflicting-twl S = *None* $\Longrightarrow$
  propagate-twl S S'

inductive-cases *propagate-twlE*: *propagate-twl S T*
thm *propagateE*

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*

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 *propagate-twl-iff-propagate*:
  assumes *inv*: *cdcl$_W$-all-struct-inv S*
  shows *propagate S T* $\longleftrightarrow$ *propagate-twl S T* (**is** *?P* $\longleftrightarrow$ *?T*)
proof
  assume *?P*
  then obtain *L E* **where**
    raw-conflicting-twl S = *None* **and**
    CL-Clauses: *E* $\in$ *set* (*twl.raw-clauses S*) **and**
    LE: *L* $\in\#$ *mset* (*raw-clause E*) **and**
    tr-CNot: *trail-twl S* $\models$*as CNot* (*remove1-mset L* (*mset* (*raw-clause E*))) **and**
    undef-lot[*simp*]: *undefined-lit* (*trail-twl S*) *L* **and**
    T $\sim$ *cons-trail-twl* (*Propagated L E*) *S*
     by (*blast elim*: *propagateE*)
  have *distinct* (*raw-clause E*)
    using *inv CL-Clauses* **unfolding** *cdcl$_W$-all-struct-inv-def distinct-mset-set-def*
    distinct-cdcl$_W$-state-def raw-clauses-def **by** *auto*
  then have *X*: *remove1-mset L* (*mset* (*raw-clause E*)) = *mset-set* (*set* (*raw-clause E*) − {*L*})
    by (*auto simp*: *multiset-eq-iff raw-clause-def count-mset distinct-filter-eq-if*)
  have (*L*, *E*) $\in$ *candidates-propagate-twl S*
    apply (*rule wf-candidates-propagate-complete*)
         using *rough-state-of-twl* **apply** *auto*[]
        using *CL-Clauses* **unfolding** *raw-clauses-def twl.raw-clauses-def*
        apply *auto*[]
       using *LE* **apply** *simp*
      using *tr-CNot X* **apply** *simp*
     using *undef-lot* **apply** *blast*
     done
  show *?T*

apply (*rule propagate-twl-rule*)
apply (*rule* ‹(*L*, *E*) ∈ *candidates-propagate-twl S*›)
using ‹*T* ∼ *cons-trail-twl* (*Propagated L E*) *S*›
apply (*auto simp*: ‹*raw-conflicting-twl S* = *None*› *twl.state-eq-def*)
done
next
assume *?T*
then obtain *L C* **where**
LC: (*L*, *C*) ∈ *candidates-propagate-twl S* **and**
T: *T* ∼ *cons-trail-twl* (*Propagated L C*) *S* **and**
confl: *raw-conflicting-twl S* = *None*
by (*auto elim*: *propagate-twlE*)
have
C'S: *C* ∈ *set* (*raw-clauses-twl S*) **and**
L: *set* (*watched C*) − *uminus* ‘ *lits-of-l* (*trail-twl S*) = {*L*} **and**
undef: *undefined-lit* (*trail-twl S*) *L*
using *LC* **unfolding** *candidates-propagate-def raw-clauses-def* **by** *auto*
have *dist*: *distinct* (*raw-clause C*)
using *inv C'S* **unfolding** *cdcl*$_W$*-all-struct-inv-def distinct-cdcl*$_W$*-state-def*
distinct-mset-set-def twl.raw-clauses-def **by** *fastforce*
then have *C-L-L*: *mset-set* (*set* (*raw-clause C*) − {*L*}) = *mset* (*raw-clause C*) − {#*L*#}
by (*metis distinct-mset-distinct distinct-mset-minus distinct-mset-set-mset-ident mset-remove1*
set-mset-mset set-remove1-eq)

show *?P*
apply (*rule propagate-rule*[*of S C L*])
using *confl* **apply** *auto*[]
using *C'S* **unfolding** *twl.raw-clauses-def* **apply** (*simp add*: *raw-clauses-def*)
using *L* **unfolding** *candidates-propagate-def* **apply** (*auto simp*: *raw-clause-def*)[]
using *wf-candidates-propagate-sound*[*OF - LC*] *rough-state-of-twl dist*
apply (*simp add*: *distinct-mset-remove1-All*)
using *undef* **apply** *simp*
using *T undef* **by** (*smt backtrack-lvl-cons-trail confl init-clss-cons-trail*
learned-clss-cons-trail marked-lit.sel(*2*) *raw-conflicting-cons-trail state-eq-def*
trail-cons-trail twl2.mmset-of-mlit.simps(*1*) *twl2.mset-cls-cls-of-ccls*)
qed

no-notation *twl.state-eq-twl* (**infix** ∼*TWL 51*)

inductive *conflict-twl* **where**
conflict-twl-rule:
C ∈ *candidates-conflict-twl S* ⟹
S' ∼ *update-conflicting-twl* (*Some* (*raw-clause C*)) *S* ⟹
raw-conflicting-twl S = *None* ⟹
conflict-twl S S'

inductive-cases *conflict-twlE*: *conflict-twl S T*

lemma *conflict-twl-iff-conflict*:
shows *conflict S T* ⟷ *conflict-twl S T* (**is** *?C* ⟷ *?T*)
proof
assume *?C*
then obtain *D* **where**
S: *raw-conflicting-twl S* = *None* **and**
D: *D* ∈ *set* (*raw-clauses S*) **and**

MD: *trail-twl S* $\models$*as CNot* (*mset* (*raw-clause D*)) **and**
T: *T* $\sim$ *update-conflicting-twl* (*Some* (*raw-clause D*)) *S*
by (*elim conflictE*)

have *D* $\in$ *candidates-conflict-twl S*
apply (*rule wf-candidates-conflict-complete*)
apply *simp*
using *D* **apply** (*auto simp*: *raw-clauses-def twl.raw-clauses-def*)[]
using *MD S* **by** *auto*
moreover have *T* $\sim$ *twl-of-rough-state* (*update-conflicting* (*Some* (*raw-clause D*)) (*rough-state-of-twl S*))
using *T* **unfolding** *rough-cdcl.state-eq-def state-eq-def* **by** *auto*
ultimately show *?T*
using *S* **by** (*auto intro*: *conflict-twl-rule*)
next
assume *?T*
then obtain *C* **where**
C: *C* $\in$ *candidates-conflict-twl S* **and**
T: *T* $\sim$ *update-conflicting-twl* (*Some* (*raw-clause C*)) *S* **and**
confl: *raw-conflicting-twl S* = *None*
by (*auto elim*: *conflict-twlE*)
have
C $\in$ *set* (*raw-clauses S*)
using *C* **unfolding** *candidates-conflict-def raw-clauses-def twl.raw-clauses-def* **by** *auto*
moreover have *trail-twl S* $\models$*as CNot* (*mset* (*raw-clause C*))
using *wf-candidates-conflict-sound*[*OF - C*] **by** *auto*
ultimately show *?C* **apply** −
apply (*rule conflict.conflict-rule*[*of - C*])
using *confl T* **unfolding** *rough-cdcl.state-eq-def* **by** (*auto simp del*: *map-map*)
qed

inductive $cdcl_W$*-twl* :: *'v wf-twl* $\Rightarrow$ *'v wf-twl* $\Rightarrow$ *bool* **for** *S* :: *'v wf-twl* **where**
propagate: *propagate-twl S S'* $\Longrightarrow$ $cdcl_W$*-twl S S'* |
conflict: *conflict-twl S S'* $\Longrightarrow$ $cdcl_W$*-twl S S'* |
other: $cdcl_W$*-o S S'* $\Longrightarrow$ $cdcl_W$*-twl S S'*|
rf: $cdcl_W$*-rf S S'* $\Longrightarrow$ $cdcl_W$*-twl S S'*

lemma $cdcl_W$*-twl-iff-*$cdcl_W$:
assumes $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-twl S T* $\longleftrightarrow$ $cdcl_W$ *S T*
by (*simp add*: *assms* $cdcl_W$*.simps* $cdcl_W$*-twl.simps conflict-twl-iff-conflict propagate-twl-iff-propagate del*: *map-map*)

lemma *rtranclp-*$cdcl_W$*-twl-all-struct-inv-inv*:
assumes $cdcl_W$*-twl*** *S T* **and** $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-all-struct-inv T*
using *assms* **by** (*induction rule*: *rtranclp-induct*)
(*simp-all add*: $cdcl_W$*-twl-iff-*$cdcl_W$ $cdcl_W$*-all-struct-inv-inv del*: *map-map*)

lemma *rtranclp-*$cdcl_W$*-twl-iff-rtranclp-*$cdcl_W$:
assumes $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-twl*** *S T* $\longleftrightarrow$ $cdcl_W{}^{**}$ *S T* (**is** *?T* $\longleftrightarrow$ *?W*)
proof
assume *?W*
then show *?T*

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 $cdcl_W$*-twl T U*
using *assms st cdcl rtranclp-*$cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-twl-iff-*$cdcl_W$
by *blast*
then show *?case* **using** *IH* **by** *auto*
qed
next
assume *?T*
then show *?W*
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 $cdcl_W$ *T U*
using *assms st cdcl rtranclp-*$cdcl_W$*-twl-all-struct-inv-inv* $cdcl_W$*-twl-iff-*$cdcl_W$
by *blast*
then show *?case* **using** *IH* **by** *auto*
qed
qed

end

end
theory *Prop-Superposition*
imports *Partial-Clausal-Logic ../lib/Herbrand-Interpretation*
begin
sledgehammer-params[*verbose*]
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*:
$S \models h\ C \longleftrightarrow \{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\} \models C$
unfolding *Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def*
by *auto*

lemma *herbrand-consistent-interp*:
consistent-interp $(\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\})$
unfolding *consistent-interp-def* **by** *auto*

lemma *herbrand-total-over-set*:
total-over-set $(\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\})\ T$
unfolding *total-over-set-def* **by** *auto*

lemma *herbrand-total-over-m*:
total-over-m $(\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\})\ T$
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 =
$\{A.\ C \in N \land C \neq \{\#\} \land Max\ (set\text{-}mset\ C) = Pos\ A \land count\ C\ (Pos\ A) \leq 1$
$\land \neg (\bigcup D \in \{D.\ D \ \#{\subset}\#\ C\}.\ production\ D) \models h\ C \land S\ C = \{\#\}\}$
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*:
$production\ C = \{A.\ C \in N \land C \neq \{\#\} \land Max\ (set\text{-}mset\ C) = Pos\ A \land count\ C\ (Pos\ A) \leq 1 \land \neg interp\ C \models h\ C \land S\ C = \{\#\}\}$
unfolding *interp-def* **by** (*rule production.simps*)

abbreviation *productive A* ≡ (*production A* ≠ {})

abbreviation *produces* :: *'a clause* ⇒ *'a* ⇒ *bool* **where**
produces C A ≡ *production C* = {*A*}

lemma *producesD*:
$produces\ C\ A \Longrightarrow C \in N \land C \neq \{\#\} \land Pos\ A = Max\ (set\text{-}mset\ C) \land count\ C\ (Pos\ A) \leq 1 \land \neg interp\ C \models h\ C \land S\ C = \{\#\}$
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 $A \Longrightarrow \neg$ *Neg* $A \in\#$ 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 66 of CW.

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*

hence D #⊂# {#*Neg* A#}
  by (*auto intro*: *Max-pos-neg-less-multiset*)
  moreover have {#*Neg* A#} #⊆# 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 ∈# $C \Longrightarrow A \notin$ *Interp* $D \Longrightarrow C$ #⊆# $D \Longrightarrow \neg$ *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 #⊆# D **and**
    d-lt-d': D #⊂# 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 ∈# $C \wedge A \in$ *Interp* D)
  case *True*
  then obtain A **where** *a-in-c*: *Pos* A ∈# 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*
  thus *?thesis*
    using *subs a-in-c* **by** (*blast dest*: *contra-subsetD*)
next
  case *False*
  then obtain A **where** *a-in-c*: *Neg* A ∈# C **and** $A \notin$ *Interp* D
    using *c-at-d* **unfolding** *true-cls-def* **by** *blast*
  hence $\bigwedge D''.\ \neg$ *produces* D'' A
    using *c-le-d neg-notin-Interp-not-produce* **by** *simp*
  thus *?thesis*
    using *a-in-c subs not-produces-imp-notin-production* **by** *auto*
qed

lemma *true-Interp-imp-interp*: C #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *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 #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *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 #⊆# $D \Longrightarrow$ *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*
thus *?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*
hence $\bigwedge D''.\ \neg$ *produces* $D''\ A$
using *c-le-d* **by** (*auto dest*: *produces-imp-in-interp less-eq-imp-interp-subseteq-interp*)
thus *?thesis*
using *a-in-c subs not-produces-imp-notin-production* **by** *auto*
qed

This lemma corresponds to theorem 2.7.6 page 66 of CW. 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 66 of CW. 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*)

thus *?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*
    thus *?thesis*
      unfolding *production-unfold* **by** *auto*
  next
    case *P*
    thus *?thesis*
      unfolding *production-unfold* **by** *auto*
  qed
qed

This lemma corresponds to theorem 2.7.6 page 66 of CW.

lemma
  assumes *D*: $C+\{\#Neg\ P\#\}\ \#\subset\#\ D$
  shows *production D* $\neq \{P\}$
proof $-$
  note $D' = D$[*unfolded less-multiset$_{HO}$*]
  consider
    (*P*) *Neg P* $\in\#$ *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*
  thus *?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*
      hence *Max* (*set-mset D*) $>$ *Pos P*
        using *less-trans*[*of Pos P Neg P Max* (*set-mset D*)] **by** *auto*
      thus *?thesis*
        unfolding *production-unfold* **by** *auto*
    next
      case *P*
      hence *Max* (*set-mset D*) $>$ *Pos P*
        by (*meson Max-ge finite-set-mset le-less-trans linorder-not-le pos-less-neg*)
      thus *?thesis*
        unfolding *production-unfold* **by** *auto*
    qed
qed

lemma *in-interp-is-produced*:
  assumes $P \in$ *INTERP*
  shows $\exists D.\ D + \{\#Pos\ P\#\} \in N \wedge$ *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*)

25.2 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: $A \in N \Longrightarrow B \in N \Longrightarrow$ *superposition-rules A B C*
$\Longrightarrow$ *superposition N* ($N \cup \{C\}$)

definition *abstract-red* :: *'a::wellorder clause* ⇒ *'a clauses* ⇒ *bool* **where**
abstract-red C N = (*clss-lt N C* ⊨*p C*)

lemma *less-multiset*[*iff*]: $M < N \longleftrightarrow M$ #⊂# N
unfolding *less-multiset-def* **by** *auto*

lemma *less-eq-multiset*[*iff*]: $M \leq N \longleftrightarrow M$ #⊆# 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*. $P \in A$} ∪ {*Neg P* |*P*. $P \notin A$}
have *B*: *?I* ⊨*s B* **using** *AB*
by (*auto simp add*: *herbrand-interp-iff-partial-interp-clss*)

have *IH*: $\bigwedge I$. *total-over-set I* (*atms-of C*) $\Longrightarrow$ *total-over-m I B* $\Longrightarrow$ *consistent-interp I*
$\Longrightarrow$ *I* ⊨*s B* $\Longrightarrow$ *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 {$D \in N$. *D* #⊂# *C*} ⊆ {$D' \in N$. *D'* #⊂# *D*}
using *c-lt-d less-eq-imp-le-multiset* **by** *fastforce*
thus *?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*)
      hence $aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})\ (I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})$
        $\notin atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\} \vee Pos\ (aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
          $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})) \in I$
            $\cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\}$
          $\vee Neg\ (aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
            $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})) \in I$
            $\cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\}$
        by *auto*
     hence *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*)
      thus *?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*
  thus *?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_𝓘 = INTERP N
  assume ¬ ?thesis
  hence not-empty: {E∈N. ¬?N_𝓘 ⊨h E} ≠ {}
    unfolding true-clss-def Ball-def by auto
  def D ≡ Min {E∈N. ¬?N_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_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_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
        hence L ∈# D
          using S-selects-subseteq[of D] by auto
        hence 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)
        hence 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_𝓘 ⊨h E
        using cls-not-D not-le by fastforce
      hence ?N_𝓘 ⊨hs clss-lt N D
        unfolding clss-lt-def true-clss-def Ball-def by blast
```

thus *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** $Max\ (set\text{-}mset\ D) = Pos\ P$
| ($Lneg$) P **where** $L = Neg\ P$
using *LSD S-selects-neg-lits*[*of L D*] **by** (*cases L*) *auto*
thus *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*
hence *count*: $count\ D\ L = 1$
unfolding D **by** (*auto simp*: *not-in-iff*)
have $\neg ?N_{\mathcal{I}} \models h\ D$
using *not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms*
by *blast*
hence *produces N D P*
using *not-empty empty finite* ‹$D \in N$› *count L*
true-interp-imp-INTERP **unfolding** *production-iff-produces* **unfolding** *production-unfold*
by (*auto simp add*: *max not-empty*)
hence $INTERP\ N \models h\ D$
unfolding D
by (*metis pos-literal-in-imp-true-cls produces-imp-Pos-in-lits*
production-subseteq-INTERP singletonI subsetCE)
thus *False*
using *not-d-interp* **by** *blast*
qed
then have $Pos\ P \in\#\ C$
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*: $superposition\text{-}rules\ D\ D\ (D - \{\#L\#\})$
unfolding C' L **by** (*auto simp add*: *superposition-rules.simps*)
have $C' + \{\#Pos\ P\#\}\ \#\subset\#\ C' + \{\#Pos\ P\#\} + \{\#Pos\ P\#\}$
by *auto*
moreover have $\neg ?N_{\mathcal{I}} \models h\ (D - \{\#L\#\})$
using *not-d-interp* **unfolding** C' L **by** *auto*
ultimately have $C' + \{\#Pos\ P\#\} \notin N$
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 $D - \{\#L\#\}\ \#\subset\#\ D$
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* ‹$D \in N$›‹$C' + \{\#Pos\ P\#\} \notin N$› **unfolding** *saturated-def* C' L *c′-p-p*
by *blast*
moreover have $C' + \{\#Pos\ P\#\}\ \subseteq\#\ C' + \{\#Pos\ P\#\} + \{\#Pos\ P\#\}$
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*
hence *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*
hence $\bigwedge y.\ y \in\#\ (E + \{\#Pos\ P\#\})$
$\Longrightarrow count\ (E + \{\#Pos\ P\#\})\ (Neg\ P) < count\ (C + \{\#Neg\ P\#\})\ (Neg\ P)$
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 $less\text{-}multiset_{HO}$ **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 $?N_{\mathcal{I}} \models h\ C{+}E \lor C{+}E \notin N$
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*)
hence $\bigwedge y.\ y \in\#\ C{+}E$
$\Longrightarrow count\ (C{+}E)\ (Pos\ P) < count\ (E + \{\#Pos\ P\#\})\ (Pos\ P)$
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 $(C + E) \models p\ E + \{\#Pos\ P\#\} \lor$ *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*) ∪ {*E* + {#*Pos P*#}}) **and**
consistent-interp I **and**
I ⊨*s clss-lt N* (*C* + *E*)
hence *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*)
thus *?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*)
  thus *?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
theory *CDCL-W-Merge*
imports *CDCL-W-Termination*
begin

26 Link between Weidenbach's and NOT's CDCL

26.1 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 $\neg$*backtrack V T*
  using *cdcl inv*
  apply (*induction rule*: *cdcl*$_W$*-bj.induct*)
    apply (*elim skipE*, *force elim*!: *backtrack-levE*[*OF - inv*] *simp*: *cdcl*$_W$*-M-level-inv-def*)
   apply (*elim resolveE*, *force elim*!: *backtrack-levE*[*OF - inv*] *simp*: *cdcl*$_W$*-M-level-inv-def*)
  apply *standard*
  apply (*elim backtrack-levE*[*OF - inv*], *elim backtrackE*)
  apply (*force simp del*: *state-simp simp add*: *state-eq-def cdcl*$_W$*-M-level-inv-decomp*)
  done

inductive *skip-or-resolve* :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
s-or-r-skip[*intro*]: *skip S T* $\Longrightarrow$ *skip-or-resolve S T* |
s-or-r-resolve[*intro*]: *resolve S T* $\Longrightarrow$ *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

26.2 More lemmas conflict–propagate and backjumping

26.2.1 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-marked-decomposition-exists-prepend*
elim!: *backtrack-levE skipE resolveE*
simp: *cdcl$_W$-M-level-inv-def*)+

lemma *wf-cdcl$_W$-bj*:
wf {(*b*,*a*). *cdcl$_W$-bj a b* $\wedge$ *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 $\exists$ *T*. *full cdcl$_W$-bj S T*
proof −
obtain *T* **where** *T*: *full* (λ*a b*. *cdcl$_W$-bj a b* $\wedge$ *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
$\exists$*M*. *trail S = M @ trail T* $\wedge$ ($\forall$*m*$\in$*set M*. $\neg$*is-marked 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*)

26.2.2 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: *raw-conflicting V* = *Some D* **and**
  LD: *L* ∈# *mset-ccls D* **and**
  decomp: (*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-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*) (*mset-ccls D*) **and**
  max-D: *get-maximum-level* (*trail V*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* **and**
  undef: *undefined-lit M1 L* **and**
  W: *W* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
        (*reduce-trail-to M1*
          (*add-learned-cls* (*cls-of-ccls D*)
            (*update-backtrack-lvl i*
              (*update-conflicting None V*))))
using *bt inv* **by** (*elim backtrack-levE*) *metis+*
obtain *L′ C′ M E* **where**
  tr: *trail T* = *Propagated L′ C′* # *M* **and**
  raw: *raw-conflicting T* = *Some E* **and**
  LE: −*L′* ∉# *mset-ccls E* **and**
  E: *mset-ccls E* ≠ {#} **and**
  V: *V* ∼ *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*]: *mset-ccls D* = *mset-ccls 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-rev-level-ge-0-atm-of-in*[*of 0 rev* (*trail V*) *L*] **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* (*mset-ccls D*)
  using *inv′ raw* **unfolding** *cdcl*$_W$*-conflicting-def cdcl*$_W$*-all-struct-inv-def tr-M* **by** *auto*

then have $L' \notin\#$ *mset-ccls* (*remove-clit* L D)
  using *L-L' L'-M* ‹*Propagated* L' C' # *trail* V ⊨*as CNot* (*mset-ccls 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 undef 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: $W \sim$ *cons-trail* (*Propagated L* (*cls-of-ccls E*)) (*reduce-trail-to M1*
  (*add-learned-cls* (*cls-of-ccls E*) (*update-backtrack-lvl i* (*update-conflicting None T*))))
  using *W V undef M-lev decomp tr*
  by (*auto simp del*: *state-simp simp*: *state-eq-def* $cdcl_W$*-M-level-inv-def*)

obtain *M2'* **where**
  decomp': (*Marked K* (*i*+1) # *M1*, *M2'*) ∈ *set* (*get-all-marked-decomposition* (*trail T*))
  using *decomp V* **unfolding** *tr-M* **by** (*cases hd* (*get-all-marked-decomposition* (*trail V*)),
    cases get-all-marked-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* $L' \notin$ *atms-of* (*mset-ccls D*)
    by (*metis DE LE L-L'* ‹$L' \notin\#$ *mset-ccls* (*remove-clit L D*)› *in-remove1-mset-neq remove-clit*
      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* (*mset-ccls D*)
    using *calculation*(*2*) *lev-L max* **by** *auto*
moreover
  have *atm-of* $L' \notin$ *atms-of* (*mset-ccls* (*remove-clit L D*))
    by (*metis DE LE* ‹$L' \notin\#$ *mset-ccls* (*remove-clit L D*)›
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def in-remove1-mset-neq remove-clit
      in-atms-of-remove1-mset-in-atms-of)
  have *i* = *get-maximum-level ?M* (*mset-ccls* (*remove-clit L D*))
    using *max-D* ‹*atm-of* $L' \notin$ *atms-of* (*mset-ccls* (*remove-clit L D*))› **by** *auto*

ultimately have *backtrack T W*
  apply −
  apply (*rule backtrack-rule*[*of T - L K i M1 M2' W*, *OF raw*])
  unfolding *tr-M*[*symmetric*]
       using *LD* **apply** *simp*
      apply *simp*
     apply *simp*
    apply *simp*
   apply *auto*[]
  using *W-S* **by** *auto*
then show *?thesis* **using** *IH inv* **by** *blast*
qed

lemma *fst-get-all-marked-decomposition-prepend-not-marked*:
  assumes ∀ *m*∈*set MS*. ¬ *is-marked m*

shows *set* (*map fst* (*get-all-marked-decomposition M*))
= *set* (*map fst* (*get-all-marked-decomposition* (*MS* @ *M*)))
using *assms* **apply** (*induction MS rule*: *marked-lit-list-induct*)
apply *auto*[*2*]
by (*rename-tac L m xs*; *case-tac get-all-marked-decomposition* (*xs* @ *M*)) *simp-all*

See also $[\![$*skip*** *?S ?T*; *backtrack ?T ?W*; $cdcl_W$*-all-struct-inv ?S*$]\!] \Longrightarrow$ *backtrack ?S ?W*

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*!: *backtrack-levE*)
then obtain *K i M1 M2 L D* **where**
raw-S: *raw-conflicting S* = *Some D* **and**
LD: *L* ∈# *mset-ccls D* **and**
decomp: (*Marked K* (*Suc i*) # *M1*, *M2*) ∈ *set* (*get-all-marked-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*) (*mset-ccls D*) **and**
i: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls D*)) ≡ *i* **and**
undef: *undefined-lit M1 L* **and**
W: *W* ∼ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls D*)
(*update-backtrack-lvl i*
(*update-conflicting None S*))))

using *bt* **by** (*elim backtrack-levE*)
(*simp-all add*: $cdcl_W$*-M-level-inv-decomp state-eq-def del*: *state-simp*)
let *?D* = *remove1-mset L* (*mset-ccls 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-marked m*
using *rtranclp-skip-state-decomp*(*1*)[*OF skip*] *raw-S M-lev* **by** *auto*
have *T*: *state T* = (M_T, *init-clss S*, *learned-clss S*, *backtrack-lvl S*, *Some* (*mset-ccls D*))
using M_T *rtranclp-skip-state-decomp*[*of S T*] *skip raw-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* (*mset-ccls D*)
unfolding $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def* **using** *T* **by** *blast*

then have $\forall L\in\#$*mset-ccls D*. *atm-of L* $\in$ *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 $\forall L\in\#$*mset-ccls D*. *atm-of L* $\notin$ *atm-of ' lits-of-l MS*
unfolding *M* **unfolding** *lits-of-def* **by** *auto*
then have *H*: $\bigwedge L$. $L\in\#$*mset-ccls D* $\Longrightarrow$ *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*) (*mset-ccls D*) = *get-maximum-level* M_T (*mset-ccls D*)
using ⟨M_T $\models$*as CNot* (*mset-ccls D*)⟩ *M nm* **by** (*metis true-annots-CNot-all-atms-defined*
get-maximum-level-skip-un-marked-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-marked-invert*
get-all-marked-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp)
have *W*: *W* $\sim$ *cons-trail* (*Propagated L* (*cls-of-ccls D*)) (*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls D*) (*update-backtrack-lvl i* (*update-conflicting None T*))))
using *W T i decomp undef* **by** (*auto simp del*: *state-simp simp*: *state-eq-def*)
have *lev-l-D'*: *get-level* M_T *L* = *get-maximum-level* M_T (*mset-ccls 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 *Marked K* (*i* + *1*) # *M1* $\in$ *set* (*map fst* (*get-all-marked-decomposition* (*trail S*)))
using *Set.imageI*[*OF decomp*, *of fst*] **by** *auto*
then have *Marked K* (*i* + *1*) # *M1* $\in$ *set* (*map fst* (*get-all-marked-decomposition* M_T))
using *fst-get-all-marked-decomposition-prepend-not-marked*[*OF nm*] **unfolding** *M* **by** *auto*
then obtain *M2'* **where** *decomp'*:(*Marked K* (*i*+*1*) # *M1*, *M2'*) $\in$ *set* (*get-all-marked-decomposition*
M_T)
by *auto*
then show *backtrack T W*
using *T decomp' lev-l' lev-l-D' i' W LD undef*
by (*force intro*!: *backtrack.intros 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*
$\vee$ ($\exists$ *U*. *skip-or-resolve*** *S U* $\wedge$ *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** ($\exists$ *U*. *skip-or-resolve*** *S U* $\wedge$ *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 dest*: *hd-raw-trail*)

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**
raw-S: *raw-conflicting* S = *Some* D **and**
LD: L ∈# *mset-ccls* D **and**
decomp: (*Marked* K (*Suc* i) # *M1*, *M2*) ∈ *set* (*get-all-marked-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) (*mset-ccls* D) **and**
i: *get-maximum-level* (*trail* S) (*remove1-mset* L (*mset-ccls* D)) ≡ i **and**
undef: *undefined-lit M1* L **and**
T: $T \sim$ *cons-trail* (*Propagated* L (*cls-of-ccls* D))
(*reduce-trail-to M1*
(*add-learned-cls* (*cls-of-ccls* D)
(*update-backtrack-lvl* i
(*update-conflicting None* S))))
using *bt-T* **by** (*elim backtrack-levE*) (*force simp*: *cdcl*$_W$*-M-level-inv-def*)+

obtain K' i' *M1'* *M2'* L' D' **where**
raw-S': *raw-conflicting* S = *Some* D' **and**
LD': L' ∈# *mset-ccls* D' **and**
decomp': (*Marked* K' (*Suc* i') # *M1'*, *M2'*) ∈ *set* (*get-all-marked-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) (*mset-ccls* D') **and**
i': *get-maximum-level* (*trail* S) (*remove1-mset* L' (*mset-ccls* D')) ≡ i' **and**
undef': *undefined-lit M1'* L' **and**

```
    U: U ∼ cons-trail (Propagated L' (cls-of-ccls D'))
              (reduce-trail-to M1'
                (add-learned-cls (cls-of-ccls D')
                  (update-backtrack-lvl i'
                    (update-conflicting None S))))
    using bt-U lev by (elim backtrack-levE) (force simp: cdcl_W-M-level-inv-def)+
  obtain c where M: trail S = c @ M2 @ Marked K (i + 1) # M1
    using decomp by auto
  obtain c' where M': trail S = c' @ M2' @ Marked K' (i' + 1) # M1'
    using decomp' by auto
  have marked: get-all-levels-of-marked (trail S) = rev [1..<1+backtrack-lvl S]
    using inv unfolding cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def by auto
  then have i < backtrack-lvl S
    unfolding M by (force simp add: rev-swap[symmetric] dest!: arg-cong[of - - set])

  have [simp]: L' = L
    proof (rule ccontr)
      assume ¬ ?thesis
      then have L' ∈# remove1-mset L (mset-ccls D)
        using raw-S raw-S' LD LD' by (simp add: in-remove1-mset-neq)
      then have get-maximum-level (trail S) (remove1-mset L (mset-ccls D)) ≥ 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]: mset-ccls D' = mset-ccls D
    using raw-S raw-S' by auto
  have [simp]: i' = i
    using i i' by auto
```

Automation in a step later...

```
  have H: ⋀a A B. insert a A = B ⟹ a : B
    by blast
  have get-all-levels-of-marked (c@M2) = rev [i+2..<1+backtrack-lvl S] and
    get-all-levels-of-marked (c'@M2') = rev [i+2..<1+backtrack-lvl S]
    using marked unfolding M
    using marked unfolding M'
    unfolding rev-swap[symmetric] by (auto dest: append-cons-eq-upt-length-i-end)
  from arg-cong[OF this(1), of set] arg-cong[OF this(2), of set]
  have
    dropWhile (λL. ¬is-marked L ∨ level-of L ≠ Suc i) (c @ M2) = [] and
    dropWhile (λL. ¬is-marked L ∨ level-of L ≠ Suc i) (c' @ M2') = []
      unfolding dropWhile-eq-Nil-conv Ball-def
      by (intro allI; rename-tac x; case-tac x; auto dest!: H simp add: in-set-conv-decomp)+

  then have [simp]: M1' = M1
    using arg-cong[OF M, of dropWhile (λL. ¬is-marked L ∨ level-of L ≠ Suc i)]
    unfolding M' by auto
  show ?thesis using T U undef 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* ≠ [] **and**
tr-U: *hd-raw-trail U* = *Propagated L E* **and**
LE: *L* ∈# *mset-cls E* **and**
raw-U: *raw-conflicting U* = *Some D* **and**
LD: −*L* ∈# *mset-ccls D* **and**
get-maximum-level (*trail U*) (*mset-ccls* (*remove-clit* (−*L*) *D*)) = *backtrack-lvl U* **and**
V: *V* ∼ *update-conflicting* (*Some* (*union-ccls* (*remove-clit* (−*L*) *D*)
(*ccls-of-cls* (*remove-lit L E*))))
(*tl-trail U*)
using *resolve* **by** (*auto elim*!: *resolveE*)
have $cdcl_W$*-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*+
then have
S: *init-clss U* = *init-clss S*
learned-clss U = *learned-clss S*
backtrack-lvl U = *backtrack-lvl S*
conflicting S = *Some* (*mset-ccls D*)
using *rtranclp-skip-state-decomp*[*OF skip*] *U raw-U*
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-marked m*
using *rtranclp-skip-state-decomp*[*OF skip*] **by** *blast*

obtain *K′ i′ M1′ M2′ L′ D′* **where**
raw-S′: *raw-conflicting S* = *Some D′* **and**
LD′: *L′* ∈# *mset-ccls D′* **and**
decomp′: (*Marked K′* (*Suc i′*) # *M1′*, *M2′*) ∈ *set* (*get-all-marked-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*) (*mset-ccls D′*) **and**
i′: *get-maximum-level* (*trail S*) (*remove1-mset L′* (*mset-ccls D′*)) ≡ *i′* **and**
undef′: *undefined-lit M1′ L′* **and**
R: *T* ∼ *cons-trail* (*Propagated L′* (*cls-of-ccls D′*))
(*reduce-trail-to M1′*
(*add-learned-cls* (*cls-of-ccls D′*)
(*update-backtrack-lvl i′*
(*update-conflicting None S*))))
using *bt* **by** (*elim backtrack-levE*) (*fastforce simp*: *S state-eq-def simp del*:*state-simp*)+
obtain *c* **where** *M*: *trail S* = *c* @ *M2′* @ *Marked K′* (*i′* + *1*) # *M1′*
using *get-all-marked-decomposition-exists-prepend*[*OF decomp′*] **by** *auto*
have *marked*: *get-all-levels-of-marked* (*trail S*) = *rev* [*1*..<*1*+*backtrack-lvl S*]
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *i′* < *backtrack-lvl S*
unfolding *M* **by** (*force simp add*: *rev-swap*[*symmetric*] *dest*!: *arg-cong*[*of - - set*])

have *U*: *trail U* = *Propagated L* (*mset-cls E*) # *trail V*
using *tr-S hd-raw-trail*[*OF U*] *U S V tr-U* **by** (*auto simp*: *lits-of-def*)
have *DD'*[*simp*]: *mset-ccls D'* = *mset-ccls D*
using *raw-U raw-S' S* **by** *auto*
have [*simp*]: *L'* = −*L*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have −*L* ∈# *remove1-mset L'* (*mset-ccls D'*)
using *DD' LD' LD* **by** (*simp add*: *in-remove1-mset-neq*)
moreover
have *M'*: *trail S* = M_0 @ *Propagated L* (*mset-cls 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-all-levels-of-marked* (*trail S*) = *rev* [*1..<1+backtrack-lvl S*]
using *inv U* **unfolding** $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-M-level-inv-def* **by** *auto*
then have *get-all-levels-of-marked* (*trail U*) = *rev* [*1..<1+backtrack-lvl S*]
using *nm M' U* **by** (*simp add*: *get-all-levels-of-marked-no-marked*)
then have *get-lev-L*:
get-level(*Propagated L* (*mset-cls E*) # *trail V*) *L* = *backtrack-lvl S*
using *get-level-get-rev-level-get-all-levels-of-marked*[*OF atm-L-notin-M*,
of [*Propagated L* (*mset-cls E*)]] *U* **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*
by (*metis M' get-lev-L get-rev-level-notin-end rev-append*)
ultimately
have *get-maximum-level* (*trail S*) (*remove1-mset L'* (*mset-ccls D'*)) ≥ *backtrack-lvl S*
by (*metis get-maximum-level-ge-get-level get-rev-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* (*mset-cls E*) # *trail V* ⊨*as CNot* (*mset-ccls D'*)
using $cdcl_W$*-all-struct-inv-def* $cdcl_W$*-conflicting-def raw-U U* **by** *auto*
then have ∀*L'*∈# (*remove1-mset L'* (*mset-ccls D'*)) . *atm-of L'* ∈ *atm-of* ' *lits-of-l* (*Propagated L* (*mset-cls 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 *get-maximum-level* (*trail S*) (*remove1-mset L'* (*mset-ccls D'*)) = *backtrack-lvl S*
using *get-maximum-level-skip-un-marked-not-present*[*of remove1-mset L'* (*mset-ccls D'*)
trail U M_0] *tr-S nm U*
‹*get-maximum-level* (*trail U*) (*mset-ccls* (*remove-clit* (− *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
(∃ *T U V*. (λ*S T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T*
∧ (λ*T U*. *resolve T U* ∧ *no-step backtrack T*) *T U*
∧ *skip*** *U V* ∧ *backtrack V W*)
∨ (∃ *T U*. (λ*S T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *S T*
∧ (λ*T U*. *resolve T U* ∧ *no-step backtrack T*) *T U* ∧ *skip*** *U W*)
∨ (∃ *T*. *skip*** *S T* ∧ *backtrack T W*)
∨ *skip*** *S W* (**is** *?RB S W* ∨ *?R S W* ∨ *?SB S W* ∨ *?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**
          $(\lambda S\ T.$ *skip-or-resolve* S T $\wedge$ *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* $(\lambda S\ T.$ *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 $(\lambda S\ T.$ *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*: ∀ *p pa pb pc*. ¬ *p* (*pa*) *pb* ∨ ¬ *p*** *pb pc* ∨ *p*** *pa pc*
by (*meson converse-rtranclp-into-rtranclp*)
have *skip-or-resolve T U* ∧ *no-step backtrack T*
using *RS*(*2*) *RS*(*3*) **by** *force*
then have (λ*p pa*. *skip-or-resolve p pa* ∧ *no-step backtrack p*)** *T W*
proof −
have (∃ *vr19 vr16 vr17 vr18*. *vr19* (*vr16*::′*st*) *vr17* ∧ *vr19*** *vr17 vr18*
∧ ¬ *vr19*** *vr16 vr18*)
∨ ¬ (*skip-or-resolve T U* ∧ *no-step backtrack T*)
∨ ¬ (λ*uu uua*. *skip-or-resolve uu uua* ∧ *no-step backtrack uu*)** *U W*
∨ (λ*uu uua*. *skip-or-resolve uu uua* ∧ *no-step backtrack uu*)** *T W*
by *force*
then show *?thesis*
by (*metis* (*no-types*) ‹(λ*S T*. *skip-or-resolve S T* ∧ *no-step backtrack S*)** *U W*›
‹*skip-or-resolve T U* ∧ *no-step backtrack T*› *f1*)
qed
then have (λ*p pa*. *skip-or-resolve p pa* ∧ *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 (λ*S T*. *skip-or-resolve S T* ∧ *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 $(\lambda S\ T.\ \textit{skip-or-resolve}\ S\ T \wedge \textit{no-step backtrack}\ S)^{**}\ S\ V$
using *IH H* **by** *blast*
moreover have $(\lambda S\ T.\ \textit{skip-or-resolve}\ S\ T \wedge \textit{no-step backtrack}\ S)^{**}\ V\ W$

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 $\textit{cdcl}_W$*-bj-strongly-confluent*:
assumes
$\textit{cdcl}_W\textit{-bj}^{**}\ S\ V$ **and**
$\textit{cdcl}_W\textit{-bj}^{**}\ S\ T$ **and**
n-s: *no-step* $\textit{cdcl}_W$*-bj* V **and**
inv: $\textit{cdcl}_W$*-all-struct-inv* S
shows $T \sim V \vee \textit{cdcl}_W\textit{-bj}^{**}\ T\ V$
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 $\textit{cdcl}_W{}^{**}\ S\ T$
using *st mono-rtranclp*[*of* $\textit{cdcl}_W$*-bj* $\textit{cdcl}_W$] *other* **by** *blast*
then have *lev-T*: $\textit{cdcl}_W$*-M-level-inv* T
using *inv rtranclp-*$\textit{cdcl}_W$*-consistent-inv*[*of S T*]
unfolding $\textit{cdcl}_W$*-all-struct-inv-def* **by** *auto*

consider
(*TV*) $T \sim V$
| (*bj-TV*) $\textit{cdcl}_W\textit{-bj}^{**}\ T\ V$
using *IH* **by** *blast*
then show *?case*
proof *cases*
case *TV*
have *no-step* $\textit{cdcl}_W$*-bj* T
using ‹$\textit{cdcl}_W$*-M-level-inv T*› *n-s* $\textit{cdcl}_W$*-bj-state-eq-compatible*[*of T - V*] *TV*
by (*meson backtrack-state-eq-compatible* $\textit{cdcl}_W$*-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': $\textit{cdcl}_W$*-bj* $T\ U'$ **and**
$\textit{cdcl}_W\textit{-bj}^{**}\ U'\ V$
using *IH n-s s-o-r* **by** (*metis rtranclp-unfold tranclpD*)
have $\textit{cdcl}_W{}^{**}\ S\ T$
by (*metis* (*no-types*, *hide-lams*) *bj mono-rtranclp*[*of* $\textit{cdcl}_W$*-bj* $\textit{cdcl}_W$] *other st*)
then have *inv-T*: $\textit{cdcl}_W$*-all-struct-inv* T
by (*metis* (*no-types*, *hide-lams*) *inv rtranclp-*$\textit{cdcl}_W$*-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 $U \sim U'$
      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*

26.3 CDCL FW

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 (λ*S T*. *cdcl*$_W$*-all-struct-inv S* ∧ *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* ∧ *conflicting V* = *None*)
    ∨ (∃ *T U*. *cdcl*$_W$*-merge-restart*** *S T* ∧ *conflicting V* ≠ *None* ∧ *conflict T U* ∧ *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* ≠ *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*
                ∨ *cdcl*$_W$*-merge-restart*** *S T* ∧ *conflicting U* ≠ *None*
                  ∧ *conflict T T′* ∧ *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*: *backtrack-levE*
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 $V \neq$ *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* $\longleftrightarrow$ *full* $cdcl_W$*-merge-restart* (*init-state N*) *V*
by (*rule conflicting-true-full-*$cdcl_W$*-iff-full-*$cdcl_W$*-merge*) *auto*

26.4 FW with strategy

26.4.1 The intermediate step

inductive $cdcl_W$*-s'* :: *'st* ⇒ *'st* ⇒ *bool* **where**
conflict': *full1* $cdcl_W$*-cp S S'* $\Longrightarrow$ $cdcl_W$*-s' S S'* |
decide': *decide S S'* $\Longrightarrow$ *no-step* $cdcl_W$*-cp S* $\Longrightarrow$ *full* $cdcl_W$*-cp S' S''* $\Longrightarrow$ $cdcl_W$*-s' S S''* |
bj': *full1* $cdcl_W$*-bj S S'* $\Longrightarrow$ *no-step* $cdcl_W$*-cp S* $\Longrightarrow$ *full* $cdcl_W$*-cp S' S''* $\Longrightarrow$ $cdcl_W$*-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'* $\Longrightarrow$ *full* $cdcl_W$*-cp S' S''* $\Longrightarrow$ $cdcl_W$*-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 $T \Longrightarrow cdcl_W$-$stgy^{**}$ 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
$\vee$ ($\exists$ U' U''. *full* $cdcl_W$-*cp* T' U'' $\wedge$ *full1* $cdcl_W$-*bj* U U' $\wedge$ *full* $cdcl_W$-*cp* U' U''
$\wedge$ $cdcl_W$-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: $\forall$ *x2*. ($\exists$ *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*
∨ ($\exists$ *U'*. *full1* $cdcl_W$*-bj U U'* ∧ ($\forall$ *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: ∀ *x2*. (∃ *v3*. $cdcl_W$-*cp x2 v3*) = $cdcl_W$-*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-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* ⇒ *'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** $n\text{-}s = this(2)$ **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 *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'* $\wedge$ *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* $T \neq$ *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* $V \neq$ *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* $T \neq$ *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-$S' = this(1)$ **and** bj-$T = this(2)$
    have $cdcl_W$-bj^{++} S' V
      using *skip bj-T* **by** (*metis* ‹$U = V$› $cdcl_W$-*bj.skip tranclp.simps*)
    then show *?thesis* **using** S-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 ≠ None
          using resolve 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'
            have cdcl_W-bj++ T V
              using resolve by force
            then show ?thesis using s' confl-V by auto
          next
            case (bj S') note S-S' = this(1) and bj-T = this(2)
            have cdcl_W-bj++ S' V
              using resolve bj-T by (metis ‹U = V› cdcl_W-bj.resolve tranclp.simps)
            then show ?thesis using confl-V S-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-s' S ⟷ no-step cdcl_W-cp S ∧ no-step cdcl_W-o S (is ?S' S ⟷ ?C S ∧ ?O S)
proof
  assume ?C S ∧ ?O S
  then show ?S' S
    by (auto simp: cdcl_W-s'.simps full1-def tranclp-unfold-begin)
next
  assume n-s: ?S' S
  have ?C S
    proof (rule ccontr)
      assume ¬ ?thesis
      then obtain S' where cdcl_W-cp S S'
        by auto
      then obtain T where full1 cdcl_W-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-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'* ⟹ $cdcl_W{}^{++}$ *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'* ⟹ $cdcl_W{}^{++}$ *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'* ⟹ $cdcl_W{}^{**}$ *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 $\exists T.\ cdcl_W\text{-}stgy\ S\ T$
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*:
$cdcl_W\text{-}cp^{**}\ S\ T \Longrightarrow conflicting\ S = Some\ D \Longrightarrow S = T$
using *rtranclpD tranclpD* **by** *fastforce*

inductive $cdcl_W$*-merge-cp* :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
conflict'[*intro*]: $conflict\ S\ T \Longrightarrow full\ cdcl_W\text{-}bj\ T\ U \Longrightarrow cdcl_W\text{-}merge\text{-}cp\ S\ U$ |
propagate'[*intro*]: $propagate^{++}\ S\ S' \Longrightarrow cdcl_W\text{-}merge\text{-}cp\ S\ S'$

lemma $cdcl_W$*-merge-restart-cases*[*consumes 1*, *case-names conflict propagate*]:
assumes
$cdcl_W$*-merge-cp* S U **and**
$\bigwedge T.\ conflict\ S\ T \Longrightarrow full\ cdcl_W\text{-}bj\ T\ U \Longrightarrow P$ **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*:
$cdcl_W\text{-}merge\text{-}cp\ S\ T \Longrightarrow cdcl_W\text{-}merge^{++}\ S\ T$
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-*$cdcl_W$:
$cdcl_W\text{-}merge\text{-}cp^{**}\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
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*)

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*
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*
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* $\neq$ *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** *auto*
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** *auto*
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*
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* ⟷ *full* $cdcl_W$*-s'-without-decide S V* (**is** *?fw* ⟷ *?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*: ∀ *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'.cases*)
have *ff2*: (∀ *p s sa*. ¬ *full1 p* (*s*::*'st*) *sa* ∨ *p*$^{++}$ *s sa* ∧ *no-step p sa*)
∧ (∀ *p s sa*. (¬ *p*$^{++}$ (*s*::*'st*) *sa* ∨ (∃ *s*. *p sa s*)) ∨ *full1 p s sa*)
by (*meson full1-def*)

obtain *ssa* :: $('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st \Rightarrow 'st$ **where**
ff3: $\forall p\ s\ sa.\ \neg\ p^{++}\ s\ sa \lor p\ s\ (ssa\ p\ s\ sa) \land p^{**}\ (ssa\ p\ s\ sa)\ sa$
by (*metis* (*no-types*) *tranclpD*)
then have *a3*: $\neg$ *cdcl*$_W$*-cp*$^{++}$ *V ss*
using *False* **by** (*metis option-full-cdcl*$_W$*-cp full-def*)
have $\bigwedge s.\ \neg$ *cdcl*$_W$*-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*$_W$*-s'-without-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*$_W$*-s'-without-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*$_W$*-merge-cp*** *S V* **and** *conflicting V = None*
| (*cp-false*) *cdcl*$_W$*-merge-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* **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 $rtranclp\text{-}cdcl_W\text{-}s'\text{-}w\text{-}rtranclp\text{-}cdcl_W[of\ S\ U]$ $assms(2)$ $rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ $rtranclp\text{-}cdcl_W\text{-}s'\text{-}w\text{-}rtranclp\text{-}cdcl_W$ $step.hyps(1)$ **by** *blast*
ultimately show *?case* **using** $after\text{-}cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp$ **by** *fast*
qed

lemma $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy'\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp\text{-}or\text{-}eq$:
assumes $cdcl_W\text{-}merge\text{-}stgy^{**}\ S\ T$ **and** *inv*: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $S = T \vee no\text{-}step\ cdcl_W\text{-}cp\ T$
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step T U*)
moreover have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$
using $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W[of\ S\ U]$ $assms(2)$ $rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ $rtranclp\text{-}cdcl_W\text{-}s'\text{-}w\text{-}rtranclp\text{-}cdcl_W$ $step.hyps(1)$
by (*meson* $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W$)
ultimately show *?case*
using $after\text{-}cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp$ *inv* **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$
by (*metis* $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ $cdcl_W\text{-}merge\text{-}stgy.simps$ *full1-def full-def* $no\text{-}step\text{-}cdcl_W\text{-}merge\text{-}cp\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp$ $rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W$ $tranclp.intros(1)$ *tranclp-into-rtranclp*)
qed

lemma $no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj$:
assumes $no\text{-}step\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S$ **and** *inv*: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $no\text{-}step\ cdcl_W\text{-}bj\ S$
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain T **where** *S-T*: $cdcl_W\text{-}bj\ S\ T$
by *auto*
have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$
using *S-T* $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ *inv other* **by** *blast*
then obtain T' **where** $full1\ cdcl_W\text{-}bj\ S\ T'$
using $cdcl_W\text{-}bj\text{-}exists\text{-}normal\text{-}form[of\ T]$ *full-fullI S-T* **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$
by *metis*
moreover
then have $cdcl_W^{**}\ S\ T'$
using $rtranclp\text{-}mono[of\ cdcl_W\text{-}bj\ cdcl_W]$ $cdcl_W.other$ $cdcl_W\text{-}o.bj$ $tranclp\text{-}into\text{-}rtranclp[of\ cdcl_W\text{-}bj]$
unfolding *full1-def* **by** *blast*
then have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T'$
using *inv* $rtranclp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ **by** *blast*
then obtain U **where** $full\ cdcl_W\text{-}cp\ T'\ U$
using $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv$ **by** *blast*
moreover have $no\text{-}step\ cdcl_W\text{-}cp\ S$
using *S-T* **by** (*auto simp*: $cdcl_W\text{-}bj.simps$ *elim*: *rulesE*)
ultimately show *False*
using *assms* $cdcl_W\text{-}s'\text{-}without\text{-}decide.intros(2)[of\ S\ T'\ U]$ **by** *fast*
qed

lemma $cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj$:
assumes $cdcl_W\text{-}s'\text{-}w\ S\ T$ **and** $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $no\text{-}step\ cdcl_W\text{-}bj\ T$
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*
next
case *s'*
then have $R = V$
by (*metis full1-def inv local.conflict' tranclp-unfold-begin*

$rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy'\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp\text{-}or\text{-}eq$)
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 $tranclp\text{-}cdcl_W\text{-}cp\text{-}propagate\text{-}with\text{-}conflict\text{-}or\text{-}not[of\ V\ W]$ *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*
next
case *propa-confl*
moreover
then have $cdcl_W\text{-}merge\text{-}cp^{**}$ 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 $tranclp\text{-}cdcl_W\text{-}cp\text{-}propagate\text{-}with\text{-}conflict\text{-}or\text{-}not[of\ V\ W]$ *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 $cdcl_W\text{-}merge\text{-}cp^{**}$ 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 $tranclp\text{-}cdcl_W\text{-}cp\text{-}propagate\text{-}with\text{-}conflict\text{-}or\text{-}not[of\ V\ W]$ *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** $ns\text{-}cp\text{-}T = this(4)$
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 *auto*
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* $\vee$ ($V' = W \wedge$ *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 *auto*
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* $W \neq$ *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*) *map-option-is-None r-into-rtranclp*)
qed
next
case *propa*
moreover then have $cdcl_W$*-merge-cp* V' W
by *auto*
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 ‹$cdcl_W$*-merge-cp* *U* V'› *cp-confl*(*1*) **by** *force*
next
assume *confl*: *conflicting* $V' \neq$ *None*
then have *no-step* $cdcl_W$*-merge-stgy* V'
by (*fastforce simp*: $cdcl_W$*-merge-stgy.simps* *full1-def full-def* $cdcl_W$*-merge-cp.simps* *dest*!: *tranclpD elim*: *rulesE*)
have *no-step* $cdcl_W$*-merge-cp* V'
using *confl* **by** (*auto simp*: *full1-def full-def* $cdcl_W$*-merge-cp.simps* *dest*!: *tranclpD elim*: *rulesE*)
moreover have $cdcl_W$*-merge-cp* *U* *W*
using V'-W ‹$cdcl_W$*-merge-cp* *U* V'› **by** *blast*
ultimately have *full1* $cdcl_W$*-merge-cp* *R* V'
using *cp-confl*(*1*) V'-W **unfolding** *full1-def* **by** *auto*
then have $cdcl_W$*-merge-stgy* *R* V'
by *auto*
moreover have *no-step* $cdcl_W$*-merge-stgy* V'
using *confl* ‹*no-step* $cdcl_W$*-merge-cp* V'› **by** (*auto simp*: $cdcl_W$*-merge-stgy.simps* *full1-def dest*!: *tranclpD elim*: *rulesE*)
ultimately have $cdcl_W$*-merge-stgy*** *R* V' **by** *auto*
{ fix *ss* :: *'st*
have $cdcl_W$*-merge-cp* *U* *W*
using V'-W ‹$cdcl_W$*-merge-cp* *U* V'› **by** *blast*
then have ¬ $cdcl_W$*-bj* *W ss*
by (*meson conflicting-not-true-rtranclp-*$cdcl_W$*-merge-cp-no-step-*$cdcl_W$*-bj* *cp-confl*(*1*) *rtranclp.rtrancl-into-rtrancl step.prems*)
then have $cdcl_W$*-merge-stgy*** *R W* ∧ *conflicting W* = *None* ∨ $cdcl_W$*-merge-stgy*** *R W* ∧ ¬ $cdcl_W$*-bj* *W ss*
using V'-W ‹$cdcl_W$*-merge-stgy*** *R* V'› **by** *presburger* **}**
then show *?thesis*
by *presburger*
qed
next

case *propa*
moreover then have $cdcl_W$*-merge-cp* V' W
by *auto*
ultimately show *?thesis* **using** ‹$cdcl_W$*-merge-cp* U V'› *cp-confl*(*1*) **by** *force*
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-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 $(s'\text{-}st\ T')$ **note** $s'\text{-}T' = this(1)$ **and** $st = this(2)$
have $cdcl_W\text{-}s'^{**}\ S\ T'$
using $s'\text{-}T'$
proof *cases*
case *conflict'*
then have $cdcl_W\text{-}s'\ S\ T'$
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 $cdcl_W\text{-}s'\ S\ T$
using *dec* $cdcl_W$*-s'.decide' n-s-S* **by** (*simp add*: *full-unfold*)
then show *?thesis* **using** *decide'* $s'\text{-}T'$ **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 $cdcl_W\text{-}s'^{**}\ R\ V$
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 $\bigwedge s.\ \neg$ *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* :: $'st \Rightarrow 'st \Rightarrow 'st$ **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* :: $('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st \Rightarrow 'st$ **where**
ff2: $\bigwedge p\ s\ sa.$ ¬ p^{++} *s sa* ∨ *p s* (*ssb p s sa*)
by (*meson tranclp-unfold-begin*)
obtain *ssc* :: $'st \Rightarrow 'st$ **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 $\bigwedge s.\ \neg$ *cdcl*$_W$*-s'-without-decide R s*
using *ff5* **by** (*simp add*: *cdcl*$_W$*-s'-without-decide.simps full1-def*)
then have $\neg$ *cdcl*$_W$*-s'-without-decide*$^{++}$ *R ss*
using *ff2* **by** *blast*
then have $\neg$ *full1 cdcl*$_W$*-s'-without-decide R ss*
by (*simp add*: *full1-def*)
then have $\neg$ *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

We will discharge the assumption later.

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* $\wedge$ *cdcl*$_W$*-merge S T*$\}$
begin

lemma *wf-tranclp-cdcl*$_W$*-merge*: *wf* $\{(T, S).$ *cdcl*$_W$*-all-struct-inv S* $\wedge$ *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* $\wedge$ *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* $\wedge$ *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* $(\lambda S\ T.$ *cdcl*$_W$*-all-struct-inv S* $\wedge$ *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: $(\lambda S\ T.$ *cdcl*$_W$*-all-struct-inv S* $\wedge$ *cdcl*$_W$*-merge-cp S T*$)^{**}$ *R S* **and**
n-s: *no-step* $(\lambda S\ T.$ *cdcl*$_W$*-all-struct-inv S* $\wedge$ *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$'$* :: *$'$e* **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

27 Link between Weidenbach's and NOT's CDCL

27.1 Inclusion of the states

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

fun *convert-marked-lit-from-W* **where**
convert-marked-lit-from-W (*Propagated L* -) = *Propagated L* () |
convert-marked-lit-from-W (*Marked L* -) = *Marked L* ()

abbreviation *convert-trail-from-W* ::
('*v*, '*lvl*, '*a*) *marked-lit list*
⇒ ('*v*, *unit*, *unit*) *marked-lit list* **where**
convert-trail-from-W ≡ *map convert-marked-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*: *marked-lit-list-induct*) *simp-all*

lemma *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-marked-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 $\models$*as C* $\longleftrightarrow$ *M* $\models$*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* $\longleftrightarrow$ *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-marked-lit-from-NOT*
:: (*'a*, *'e*, *'b*) *marked-lit* $\Rightarrow$ (*'a*, *nat*, *'cls*) *marked-lit* **where**
convert-marked-lit-from-NOT (*Propagated L -*) = *Propagated L dummy-cls* |
convert-marked-lit-from-NOT (*Marked L -*) = *Marked L 0*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT $\equiv$ *map convert-marked-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*: *marked-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*: *marked-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*: *marked-lit-list-induct*) *auto*

lemma *convert-trail-from-W-convert-lit-from-NOT*[*simp*]:
convert-marked-lit-from-W (*convert-marked-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-marked-lit-from-NOT*[*iff*]:
lit-of (*convert-marked-lit-from-NOT L*) = *lit-of L*
by (*cases L*) *auto*

sublocale $state_W$ $\subseteq$ *dpll-state-ops*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
λS. *convert-trail-from-W* (*trail S*)
raw-clauses
$\lambda L\ S$. *cons-trail* (*convert-marked-lit-from-NOT L*) *S*
λS. *tl-trail S*
$\lambda C\ S$. *add-learned-cls C S*

$\lambda C\ S.\ remove\text{-}cls\ C\ S$
by *unfold-locales*

context $state_W$
begin
lemma *convert-marked-lit-from-W-convert-marked-lit-from-NOT*[*simp*]:
convert-marked-lit-from-W (*mmset-of-mlit* (*convert-marked-lit-from-NOT* L)) = L
by (*cases* L) *auto*
end

sublocale $state_W \subseteq$ *dpll-state*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-marked-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
by *unfold-locales* (*auto simp*: *map-tl o-def*)

context $state_W$
begin
declare $state\text{-}simp_{NOT}$[*simp del*]

end

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}ops$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-marked-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- $S.$ *raw-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 $cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}proxy.axioms$
sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}\text{-}merge\text{-}bj\text{-}learn\text{-}proxy$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-marked-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*. *raw-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* ∉# *C′*
using ‹*F* ⊨*as CNot C′*› ‹*undefined-lit F L*› *Marked-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′* @ *Marked 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 ¬ *tautology* (*C′*)
using ‹*F* ⊨*as CNot C′*› *consistent-CNot-not-tautology true-annots-true-cls* **by** *blast*
then have ¬ *tautology* (*?C′* + {#*L*#})
using ‹*F* ⊨*as CNot C′*› ‹*undefined-lit F L*› **by** (*metis CNot-remdups-mset*
Marked-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* ∈ *atms-of-mm* (*clauses S*)
∪ *atm-of* ‘ *lits-of-l* (*convert-trail-from-W* (*trail S*))
using ‹*convert-trail-from-W* (*trail S*) = *F′* @ *Marked K* () # *F*›
‹*atm-of L* ∈ *atms-of-mm* (*clauses S*) ∪ *atm-of* ‘ *lits-of-l* (*F′* @ *Marked K* () # *F*)› **by** *auto*
have *f4*: *clauses S* ⊨*pm remdups-mset C′* + {#*L*#}
by (*metis* (*no-types*) ‹*L* ∉# *C′*› ‹*clauses S* ⊨*pm C′* + {#*L*#}› *remdups-mset-singleton-sum*(*2*)
true-clss-cls-remdups-mset union-commute)
have *F* ⊨*as CNot* (*remdups-mset C′*)
by (*simp add*: ‹*F* ⊨*as CNot C′*›)
obtain *D* **where** *D*: *mset-cls D* = *remdups-mset C′* + {#*L*#}
using *ex-mset-cls* **by** *blast*
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* ⊨*as CNot* (*remdups-mset C′*)›
state-eq$_{NOT}$*-ref D* **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*)
  raw-clauses
  λL S. cons-trail (*convert-marked-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. raw-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*)
  raw-clauses
  λL S. cons-trail (*convert-marked-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. raw-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*)

27.2 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*))
    ∨ *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*) ⟹
  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-marked* $m)$
    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 dest*!:
        hd-raw-trail)
    qed
qed

27.3 More lemmas conflict–propagate and backjumping

27.4 CDCL FW

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
    $\vee$ (*no-step cdcl*$_W$*-merge* T $\wedge$ *conflicting* $T \neq$ *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#} $\in$# *clauses* S **and**
    M-C: M $\models$*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 raw-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* (*Marked L* (*Suc* (*backtrack-lvl S*)))
(*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 raw-clauses-def*
apply *auto*[]
using *T undef-L* **unfolding** *state-eq-def state-eq*$_{NOT}$*-def* **by** (*auto simp*: *raw-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* !∈! *raw-learned-clss S* **and**
¬(*trail S*) ⊨*asm clauses S* **and**
mset-cls C ∉ *set* (*get-all-mark-of-propagated* (*trail S*)) **and**
C-init: *mset-cls C* ∉# *init-clss S* **and**
T: *T* ∼ *remove-cls C S*
by (*auto elim*: *forgetE*)
have *init-clss S* ⊨*pm mset-cls C*
using *inv C-le* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-learned-clause-def raw-clauses-def*
by (*meson in-clss-mset-clss true-clss-clss-in-imp-true-clss-cls*)
then have *S-C*: *removeAll-mset* (*mset-cls C*) (*clauses S*) ⊨*pm mset-cls C*
using *C-init C-le* **unfolding** *raw-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*: *raw-clauses-def*)
using *T C-le C-init* **by** (*auto*
simp: *state-eq-def Un-Diff state-eq*$_{NOT}$*-def raw-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: *raw-conflicting T* = *Some CT* **and**
CT: *mset-ccls CT* = *mset-cls* C_S **and**
C_S: C_S !∈! *raw-clauses S* **and**
tr-S-C_S: *trail S* ⊨*as CNot* (*mset-cls* 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* $\neq$ *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': *raw-conflicting T'* = *Some D* **and**
      LD: *L* $\in\#$ *mset-ccls D* **and**
      M1-M2:(*Marked K* (*i*+*1*) # *M1*, *M2*) $\in$ *set* (*get-all-marked-decomposition* (*trail T'*)) **and**
      get-level (*trail T'*) *L* = *backtrack-lvl T'* **and**
      get-level (*trail T'*) *L* = *get-maximum-level* (*trail T'*) (*mset-ccls D*) **and**
      get-maximum-level (*trail T'*) (*mset-ccls* (*remove-clit L D*)) = *i* **and**
      undef-L: *undefined-lit M1 L* **and**
      U: *U* $\sim$ *cons-trail* (*Propagated L* (*cls-of-ccls D*))
              (*reduce-trail-to M1*
                  (*add-learned-cls* (*cls-of-ccls D*)
                      (*update-backtrack-lvl i*
                          (*update-conflicting None T'*))))
      using *bt* **by** (*auto elim*: *backtrack-levE*)
    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*
  raw-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'* @ *Marked K* (*i+1*) # *tl* (*trail U*) **and**
  tr-U: *trail U* = *Propagated L* (*mset-ccls D*) # *tl* (*trail U*)
  using *U M1-M2 undef-L 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''* @ *Marked K* (*i+1*) # *tl* (*trail U*)
  using *tr-T tr-T' confl* **unfolding** *M''-def* **by** (*auto elim*: *rulesE*)
have *init-clss T'* + *learned-clss S* ⊨*pm mset-ccls D*
  using *inv-T' confl-T'* **unfolding** *cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-learned-clause-def*
  raw-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* @ [*Marked K* (*Suc i*)]])
  using *confl M1-M2* ‹*trail T* = *M* @ *trail T'*›
    apply (*auto dest*!: *get-all-marked-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*) ⊨*as CNot* (*remove1-mset L* (*mset-ccls D*))
  by (*metis append-self-conv2 tr-U*)
thm *backjump-l*[*of - - - - - L cls-of-ccls D - remove1-mset L* (*mset-ccls D*)]
have *backjump-l S U*
  apply (*rule backjump-l*[*of - - - - - L cls-of-ccls D - remove1-mset L* (*mset-ccls 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 undef-L M1-M2 inv-T' inv undef-L* **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 *U undef-L M1-M2 inv-T′ inv* **unfolding** *cdcl$_W$-all-struct-inv-def*
cdcl$_W$-M-level-inv-def **apply** *auto*[]
using *undef-L atm-L* **apply** (*simp add*: *trail-reduce-trail-to$_{NOT}$-add-learned-cls*)
using ‹*init-clss T′ + learned-clss S* ⊨*pm mset-ccls D*› *LD* **unfolding** *raw-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 ≡ *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* ∨ (*no-step cdcl$_W$-merge T* ∧ *conflicting T* ≠ *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* ∨ (*no-step cdcl$_W$-merge T* ∧ *conflicting T* ≠ *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* ⟶ *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$_{NOT}$* ≤ *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* ⇒ *nat* **where**
μ_{FW} *S* ≡ (*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*) ⊆ *atms-of-mm ?A*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def no-strange-atm-def raw-clauses-def* **by** *auto*
have *atm-trail*: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-mm ?A*
using *inv* **unfolding** $cdcl_W$*-all-struct-inv-def no-strange-atm-def raw-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* ∧ $cdcl_W$*-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*)

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* ⟷ *full* $cdcl_W$*-merge-stgy R V* (**is** *?s'* ⟷ *?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* ∨ $cdcl_W$*-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 ≠ 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**
inv: $cdcl_W$*-all-struct-inv R*
shows *full* $cdcl_W$*-stgy R V* ⟷ *full* $cdcl_W$*-merge-stgy R V*
by (*simp add*: *assms*(*1*) *full-*$cdcl_W$*-s′-full-*$cdcl_W$*-merge-restart full-*$cdcl_W$*-stgy-iff-full-*$cdcl_W$*-s′ inv*)

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* (*mset-clss N*)
shows (*conflicting S′ = Some* {#} ∧ *unsatisfiable* (*set-mset* (*mset-clss N*)))
∨ (*conflicting S′ = None* ∧ *trail S′* ⊨*asm mset-clss N* ∧ *satisfiable* (*set-mset* (*mset-clss 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 *Weidenbach-Book*
imports
Prop-Normalisation

Prop-Resolution

Prop-Superposition

CDCL-NOT DPLL-NOT DPLL-W-Implementation CDCL-W-Implementation CDCL-W-Incremental CDCL-WNOT

begin

end