Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

December 6, 2019

Contents

theory *Bits-Natural*
imports

Refine-Monadic.Refine-Monadic
Native-Word.Native-Word-Imperative-HOL
Native-Word.Code-Target-Bits-Int Native-Word.Uint32 Native-Word.Uint64
HOL−Word.More-Word
begin

instantiation *nat* :: *bit-comprehension*
begin

definition *test-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *bool*› **where**
test-bit i j = *test-bit* (*int i*) *j*

definition *lsb-nat* :: ‹*nat* ⇒ *bool*› **where**
lsb i = (*int i* :: *int*) !! *0*

definition *set-bit-nat* :: *nat* ⇒ *nat* ⇒ *bool* ⇒ *nat* **where**
set-bit i n b = *nat* (*bin-sc n b* (*int i*))

definition *set-bits-nat* :: (*nat* ⇒ *bool*) ⇒ *nat* **where**
set-bits f =
(*if* ∃ *n*. ∀ *n'*≥*n*. ¬ *f n'* *then*
let n = *LEAST n*. ∀ *n'*≥*n*. ¬ *f n'*
in nat (*bl-to-bin* (*rev* (*map f* [*0*..<*n*])))
else if ∃ *n*. ∀ *n'*≥*n*. *f n'* *then*
let n = *LEAST n*. ∀ *n'*≥*n*. *f n'*
in nat (*sbintrunc n* (*bl-to-bin* (*True* # *rev* (*map f* [*0*..<*n*]))))
else 0 :: *nat*)

definition *shiftl-nat* **where**
shiftl x n = *nat* ((*int x*) ∗ *2* ^ *n*)

definition *shiftr-nat* **where**
shiftr x n = *nat* (*int x div 2* ^ *n*)

definition *bitNOT-nat* :: *nat* ⇒ *nat* **where**
bitNOT i = *nat* (*bitNOT* (*int i*))

definition *bitAND-nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bitAND i j = *nat* (*bitAND* (*int i*) (*int j*))

definition *bitOR-nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bitOR i j = *nat* (*bitOR* (*int i*) (*int j*))

definition *bitXOR-nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bitXOR i j = *nat* (*bitXOR* (*int i*) (*int j*))

instance ..

end

lemma *nat-shiftr*[*simp*]:
m >> *0* = *m*
‹((*0*::*nat*) >> *m*) = *0*›
‹(*m* >> *Suc n*) = (*m div 2* >> *n*)› **for** *m* :: *nat*
by (*auto simp*: *shiftr-nat-def zdiv-int zdiv-zmult2-eq*[*symmetric*])

lemma *nat-shifl-div*: ‹$m >> n = m\ div\ (2\hat{}n)$› **for** $m :: nat$
  by (*induction n arbitrary*: *m*) (*auto simp*: *div-mult2-eq*)

lemma *nat-shiftl*[*simp*]:
  $m << 0 = m$
  ‹$((0::nat) << m) = 0$›
  ‹$(m << Suc\ n) = ((m * 2) << n)$› **for** $m :: nat$
  by (*auto simp*: *shiftl-nat-def zdiv-int zdiv-zmult2-eq*[*symmetric*])

lemma *nat-shiftr-div2*: ‹$m >> 1 = m\ div\ 2$› **for** $m :: nat$
  by *auto*

lemma *nat-shiftr-div*: ‹$m << n = m * (2\hat{}n)$› **for** $m :: nat$
  by (*induction n arbitrary*: *m*) (*auto simp*: *div-mult2-eq*)

definition *shiftl1* :: ‹$nat \Rightarrow nat$› **where**
  ‹$shiftl1\ n = n << 1$›

definition *shiftr1* :: ‹$nat \Rightarrow nat$› **where**
  ‹$shiftr1\ n = n >> 1$›

instantiation *natural* :: *bit-comprehension*
begin

context includes *natural.lifting* **begin**

lift-definition *test-bit-natural* :: ‹$natural \Rightarrow nat \Rightarrow bool$› **is** *test-bit* **.**

lift-definition *lsb-natural* :: ‹$natural \Rightarrow bool$› **is** *lsb* **.**

lift-definition *set-bit-natural* :: $natural \Rightarrow nat \Rightarrow bool \Rightarrow natural$ **is**
  set-bit **.**

lift-definition *set-bits-natural* :: ‹$(nat \Rightarrow bool) \Rightarrow natural$›
  is ‹$set\text{-}bits :: (nat \Rightarrow bool) \Rightarrow nat$› **.**

lift-definition *shiftl-natural* :: ‹$natural \Rightarrow nat \Rightarrow natural$›
  is ‹$shiftl :: nat \Rightarrow nat \Rightarrow nat$› **.**

lift-definition *shiftr-natural* :: ‹$natural \Rightarrow nat \Rightarrow natural$›
  is ‹$shiftr :: nat \Rightarrow nat \Rightarrow nat$› **.**

lift-definition *bitNOT-natural* :: ‹$natural \Rightarrow natural$›
  is ‹$bitNOT :: nat \Rightarrow nat$› **.**

lift-definition *bitAND-natural* :: ‹$natural \Rightarrow natural \Rightarrow natural$›
  is ‹$bitAND :: nat \Rightarrow nat \Rightarrow nat$› **.**

lift-definition *bitOR-natural* :: ‹$natural \Rightarrow natural \Rightarrow natural$›
  is ‹$bitOR :: nat \Rightarrow nat \Rightarrow nat$› **.**

lift-definition *bitXOR-natural* :: ‹$natural \Rightarrow natural \Rightarrow natural$›
  is ‹$bitXOR :: nat \Rightarrow nat \Rightarrow nat$› **.**

end

instance ..
end

context includes *natural.lifting* **begin**
lemma [*code*]:
integer-of-natural (*m* >> *n*) = (*integer-of-natural m*) >> *n*
apply *transfer*
by (*smt integer-of-natural.rep-eq msb-int-def msb-shiftr nat-eq-iff2 negative-zle shiftr-int-code shiftr-int-def shiftr-nat-def shiftr-natural.rep-eq type-definition.Rep-inject type-definition-integer*)

lemma [*code*]:
integer-of-natural (*m* << *n*) = (*integer-of-natural m*) << *n*
apply *transfer*
by (*smt integer-of-natural.rep-eq msb-int-def msb-shiftl nat-eq-iff2 negative-zle shiftl-int-code shiftl-int-def shiftl-nat-def shiftl-natural.rep-eq type-definition.Rep-inject type-definition-integer*)

end

lemma *bitXOR-1-if-mod-2*: ‹*bitXOR L 1* = (*if L mod 2* = *0 then L* + *1 else L* − *1*)› **for** *L* :: *nat*
apply *transfer*
apply (*subst int-int-eq*[*symmetric*])
apply (*rule bin-rl-eqI*)
apply (*auto simp*: *bitXOR-nat-def*)
unfolding *bin-rest-def bin-last-def bitXOR-nat-def*
apply *presburger*+
done

lemma *bitAND-1-mod-2*: ‹*bitAND L 1* = *L mod 2*› **for** *L* :: *nat*
apply *transfer*
apply (*subst int-int-eq*[*symmetric*])
apply (*subst bitAND-nat-def*)
by (*auto simp*: *zmod-int bin-rest-def bin-last-def bitval-bin-last*[*symmetric*])

lemma *shiftl-0-uint32*[*simp*]: ‹*n* << *0* = *n*› **for** *n* :: *uint32*
by *transfer auto*

lemma *shiftl-Suc-uint32*: ‹*n* << *Suc m* = (*n* << *m*) << *1*› **for** *n* :: *uint32*
apply *transfer*
apply *transfer*
by *auto*

lemma *nat-set-bit-0*: ‹*set-bit x 0 b* = *nat* ((*bin-rest* (*int x*)) *BIT b*)› **for** *x* :: *nat*
by (*auto simp*: *set-bit-nat-def*)

lemma *nat-test-bit0-iff*: ‹*n* !! *0* ⟷ *n mod 2* = *1*› **for** *n* :: *nat*
proof −
have *2*: ‹*2* = *int 2*›
by *auto*
have [*simp*]: ‹*int n mod 2* = *1* ⟷ *n mod 2* = *Suc 0*›
unfolding *2 zmod-int*[*symmetric*]
by *auto*

show *?thesis*
unfolding *test-bit-nat-def*
by (*auto simp*: *bin-last-def zmod-int*)
qed
lemma *test-bit-2*: ‹$m > 0 \Longrightarrow (2*n)$!! $m \longleftrightarrow n$!! $(m - 1)$› **for** n :: *nat*
by (*cases m*)
(*auto simp*: *test-bit-nat-def bin-rest-def*)

lemma *test-bit-Suc-2*: ‹$m > 0 \Longrightarrow$ *Suc* $(2 * n)$!! $m \longleftrightarrow (2 * n)$!! m› **for** n :: *nat*
by (*cases m*)
(*auto simp*: *test-bit-nat-def bin-rest-def*)

lemma *bin-rest-prev-eq*:
assumes [*simp*]: ‹$m > 0$›
shows ‹*nat* ((*bin-rest* (*int w*))) !! (m − *Suc* (*0*::*nat*)) = w !! m›
proof −
define m' **where** ‹$m' = w$ *div 2*›
have w: ‹$w = 2 * m' \lor w =$ *Suc* $(2 * m')$›
unfolding *m′-def*
by *auto*
moreover have ‹*bin-nth* (*int* m') (m − *Suc 0*) = m' !! (m − *Suc 0*)›
unfolding *test-bit-nat-def test-bit-int-def* **..**
ultimately show *?thesis*
by (*auto simp*: *bin-rest-def test-bit-2 test-bit-Suc-2*)
qed

lemma *bin-sc-ge0*: ‹w >= *0* ==> (*0*::*int*) ≤ *bin-sc n b w*›
by (*induction n arbitrary*: *w*) *auto*

lemma *bin-to-bl-eq-nat*:
‹*bin-to-bl* (*size a*) (*int a*) = *bin-to-bl* (*size b*) (*int b*) ==> *a*=*b*›
by (*metis Nat.size-nat-def size-bin-to-bl*)

lemma *nat-bin-nth-bl*: $n < m \Longrightarrow w$!! n = *nth* (*rev* (*bin-to-bl m* (*int w*))) n **for** w :: *nat*
apply (*induct n arbitrary*: *m w*)
subgoal for *m w*
apply *clarsimp*
apply (*case-tac m*, *clarsimp*)
using *bin-nth-bl bin-to-bl-def test-bit-int-def test-bit-nat-def* **apply** *presburger*
done
subgoal for *n m w*
apply (*clarsimp simp*: *bin-to-bl-def*)
apply (*case-tac m*, *clarsimp*)
apply (*clarsimp simp*: *bin-to-bl-def*)
apply (*subst bin-to-bl-aux-alt*)
apply (*simp add*: *bin-nth-bl test-bit-nat-def*)
done
done

lemma *bin-nth-ge-size*: ‹*nat na* ≤ $n \Longrightarrow 0 \leq$ *na* ⟹ *bin-nth na n* = *False*›
proof (*induction* ‹*n*› *arbitrary*: *na*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n na*) **note** *IH* = *this*(*1*) **and** *H* = *this*(*2*−)

have ‹*na* = *1* ∨ *0* ≤ *na div 2*›
  using *H* **by** *auto*
moreover have
  ‹*na* = *0* ∨ *na* = *1* ∨ *nat* (*na div 2*) ≤ *n*›
  using *H* **by** *auto*
ultimately show *?case*
  using *IH*[*rule-format*, *of* ‹*bin-rest na*›] *H*
  by (*auto simp*: *bin-rest-def*)
qed

lemma *test-bit-nat-outside*: *n* > *size w* ⟹ ¬*w* !! *n* **for** *w* :: *nat*
unfolding *test-bit-nat-def*
by (*auto simp*: *bin-nth-ge-size*)

lemma *nat-bin-nth-bl′*:
‹*a* !! *n* ⟷ (*n* < *size a* ∧ (*rev* (*bin-to-bl* (*size a*) (*int a*)) ! *n*))›
by (*metis* (*full-types*) *Nat.size-nat-def bin-nth-ge-size leI nat-bin-nth-bl nat-int of-nat-less-0-iff test-bit-int-def test-bit-nat-def*)

lemma *nat-set-bit-test-bit*: ‹*set-bit w n x* !! *m* = (*if m* = *n then x else w* !! *m*)› **for** *w n* :: *nat*
unfolding *nat-bin-nth-bl′*
apply *auto*
apply (*metis bin-nth-bl bin-nth-sc bin-nth-simps*(*3*) *bin-to-bl-def int-nat-eq set-bit-nat-def*)
apply (*metis bin-nth-ge-size bin-nth-sc bin-sc-ge0 leI of-nat-less-0-iff set-bit-nat-def*)
apply (*metis bin-nth-bl bin-nth-ge-size bin-nth-sc bin-sc-ge0 bin-to-bl-def int-nat-eq leI of-nat-less-0-iff set-bit-nat-def*)
apply (*metis Nat.size-nat-def bin-nth-sc-gen bin-nth-simps*(*3*) *bin-to-bl-def int-nat-eq nat-bin-nth-bl′ set-bit-nat-def test-bit-int-def test-bit-nat-def*)
apply (*metis Nat.size-nat-def bin-nth-bl bin-nth-sc-gen bin-to-bl-def int-nat-eq nat-bin-nth-bl nat-bin-nth-bl′ of-nat-less-0-iff of-nat-less-iff set-bit-nat-def*)
apply (*metis* (*full-types*) *bin-nth-bl bin-nth-ge-size bin-nth-sc-gen bin-sc-ge0 bin-to-bl-def leI of-nat-less-0-iff set-bit-nat-def*)
by (*metis bin-nth-bl bin-nth-ge-size bin-nth-sc-gen bin-sc-ge0 bin-to-bl-def int-nat-eq leI of-nat-less-0-iff set-bit-nat-def*)

end
theory *WB-More-Refinement*
imports *Weidenbach-Book-Base.WB-List-More*
  HOL−Library.Cardinality
  HOL−Library.Rewrite
  HOL−Eisbach.Eisbach
  Refine-Monadic.Refine-Basic
  Automatic-Refinement.Automatic-Refinement
  Automatic-Refinement.Relators
  Refine-Monadic.Refine-While
  Refine-Monadic.Refine-Foreach
begin

hide-const *Autoref-Fix-Rel.CONSTRAINT*

definition *fref* :: (′*c* ⇒ *bool*) ⇒ (′*a* × ′*c*) *set* ⇒ (′*b* × ′*d*) *set*
  ⇒ ((′*a* ⇒ ′*b*) × (′*c* ⇒ ′*d*)) *set*
  ([-]$_f$ - → - [*0*,*60*,*60*] *60*)
where [*P*]$_f$ *R* → *S* ≡ {(*f*,*g*). ∀ *x y*. *P y* ∧ (*x*,*y*)∈*R* ⟶ (*f x*, *g y*)∈*S*}

abbreviation *freft* (- $\rightarrow_f$ - [*60*,*60*] *60*) **where** $R \rightarrow_f S \equiv ([\lambda\text{-}.\ True]_f\ R \rightarrow S)$

lemma *frefI*[*intro?*]:
  assumes $\bigwedge x\ y.\ [\![P\ y;\ (x,y){\in}R]\!] \Longrightarrow (f\ x,\ g\ y){\in}S$
  shows $(f,g){\in}fref\ P\ R\ S$
  using *assms*
  unfolding *fref-def*
  by *auto*
lemma *fref-mono*: $[\![\ \bigwedge x.\ P'\ x \Longrightarrow P\ x;\ R' \subseteq R;\ S \subseteq S'\ ]\!]$
    $\Longrightarrow fref\ P\ R\ S \subseteq fref\ P'\ R'\ S'$
    unfolding *fref-def*
    by *auto blast*

lemma *meta-same-imp-rule*: $([\![PROP\ P;\ PROP\ P]\!] \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow PROP\ Q)$
  by *rule*
lemma *split-prod-bound*: $(\lambda p.\ f\ p) = (\lambda(a,b).\ f\ (a,b))$ **by** *auto*

This lemma cannot be moved to *Weidenbach-Book-Base.WB-List-More*, because the syntax $CARD('a)$ does not exist there.

lemma *finite-length-le-CARD*:
  assumes ‹*distinct* (*xs* :: *'a* :: *finite list*)›
  shows ‹*length xs* ≤ *CARD*(*'a*)›
proof −
  have ‹*set xs* ⊆ *UNIV*›
    by *auto*
  show *?thesis*
    by (*metis assms card-ge-UNIV distinct-card le-cases*)
qed

0.0.1 Some Tooling for Refinement

The following very simple tactics remove duplicate variables generated by some tactic like *refine-rcg*. For example, if the problem contains $(i,\ C) = (xa,\ xb)$, then only i and C will remain. It can also prove trivial goals where the goals already appears in the assumptions.

method *remove-dummy-vars* **uses** *simps* =
  ((*unfold prod.inject*)*?*; (*simp only*: *prod.inject*)*?*; (*elim conjE*)*?*;
    hypsubst?; (*simp only*: *triv-forall-equality simps*)*?*)

From → to ⇓

lemma *Ball2-split-def*: ‹$(\forall (x,\ y) \in A.\ P\ x\ y) \longleftrightarrow (\forall x\ y.\ (x,\ y) \in A \longrightarrow P\ x\ y)$›
  by *blast*

lemma *in-pair-collect-simp*: $(a,b){\in}\{(a,b).\ P\ a\ b\} \longleftrightarrow P\ a\ b$
  by *auto*

ML ‹
signature MORE-REFINEMENT = *sig*
  val down-converse: *Proof.context* −> *thm* −> *thm*
end

structure More-Refinement: *MORE-REFINEMENT* = *struct*
  val unfold-refine = (*fn context* => *Local-Defs.unfold* (*context*)
   @{*thms refine-rel-defs nres-rel-def in-pair-collect-simp*})

val unfold-Ball = (*fn context* => *Local-Defs.unfold* (*context*)
  @{*thms Ball2-split-def all-to-meta*})
val replace-ALL-by-meta = (*fn context* => *fn thm* => *Object-Logic.rulify context thm*)
val down-converse = (*fn context* =>
  replace-ALL-by-meta context o (*unfold-Ball context*) *o* (*unfold-refine context*))
end
⟩

attribute-setup *to-*⇓ = ⟨
  Scan.succeed (*Thm.rule-attribute* [] (*More-Refinement.down-converse o Context.proof-of*))
⟩ *convert theorem from* @{*text* →}−*form to* @{*text* ⇓}−*form.*

method *to-*⇓ =
  (*unfold refine-rel-defs nres-rel-def in-pair-collect-simp*;
  unfold Ball2-split-def all-to-meta;
  intro allI impI)

Merge Post-Conditions

lemma *Down-add-assumption-middle*:
  assumes
    ⟨*nofail U*⟩ **and**
    ⟨$V \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge P\ T1 \wedge Q'\ T1\ T0\}\ U$⟩ **and**
    ⟨$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$⟩
  shows ⟨$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$⟩
  using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
  by *blast*

lemma *Down-del-assumption-middle*:
  assumes
    ⟨$S1 \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge P\ T1 \wedge Q'\ T1\ T0\}\ S0$⟩
  shows ⟨$S1 \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge Q'\ T1\ T0\}\ S0$⟩
  using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
  by *blast*

lemma *Down-add-assumption-beginning*:
  assumes
    ⟨*nofail U*⟩ **and**
    ⟨$V \le \Downarrow \{(T1,\ T0).\ P\ T1 \wedge Q'\ T1\ T0\}\ U$⟩ **and**
    ⟨$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$⟩
  shows ⟨$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$⟩
  using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
  by *blast*

lemma *Down-add-assumption-beginning-single*:
  assumes
    ⟨*nofail U*⟩ **and**
    ⟨$V \le \Downarrow \{(T1,\ T0).\ P\ T1\}\ U$⟩ **and**
    ⟨$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$⟩
  shows ⟨$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$⟩
  using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
  by *blast*

lemma *Down-del-assumption-beginning*:
  fixes U :: ⟨*'a nres*⟩ **and** V :: ⟨*'b nres*⟩ **and** $Q\ Q'$ :: ⟨$'b \Rightarrow 'a \Rightarrow bool$⟩
  assumes

⟨V ≤ ⇓ {(T1, T0). Q T1 T0 ∧ Q′ T1 T0} U⟩
shows ⟨V ≤ ⇓ {(T1, T0). Q′ T1 T0} U⟩
using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
by *blast*

method *unify-Down-invs2-normalisation-post* =
((*unfold meta-same-imp-rule True-implies-equals conj-assoc*)?)

method *unify-Down-invs2* =
(*match* **premises in**
— if the relation 2-1 has not assumption, we add True. Then we call out method again and this time it will match since it has an assumption.
I: ⟨S1 ≤ ⇓ R10 S0⟩ **and**
J[*thin*]: ⟨S2 ≤ ⇓ R21 S1⟩
for *S1*:: ⟨′b nres⟩ **and** *S0* :: ⟨′a nres⟩ **and** *S2* :: ⟨′c nres⟩ **and** *R10 R21* ⇒
⟨*insert True-implies-equals*[*where P* = ⟨S2 ≤ ⇓ R21 S1⟩, *symmetric*,
THEN equal-elim-rule1, *OF J*]⟩
| *I*[*thin*]: ⟨S1 ≤ ⇓ {(T1, T0). P T1} S0⟩ (*multi*) **and**
J[*thin*]: - **for** *S1*:: ⟨′b nres⟩ **and** *S0* :: ⟨′a nres⟩ **and** *P* :: ⟨′b ⇒ bool⟩ ⇒
⟨*match J*[*uncurry*] *in*
J[*curry*]: ⟨- ⟹ S2 ≤ ⇓ {(T2, T1). R T2 T1} S1⟩ *for S2* :: ⟨′c nres⟩ *and R* ⇒
⟨*insert Down-add-assumption-beginning-single*[*where P* = *P and R* = *R and*
W = *S2 and V* = *S1 and U* = *S0*, *OF - I J*];
unify-Down-invs2-normalisation-post⟩
| - ⇒ ⟨*fail*⟩⟩
| *I*[*thin*]: ⟨S1 ≤ ⇓ {(T1, T0). P T1 ∧ Q′ T1 T0} S0⟩ (*multi*) **and**
J[*thin*]: - **for** *S1*:: ⟨′b nres⟩ **and** *S0* :: ⟨′a nres⟩ **and** *Q′* **and** *P* :: ⟨′b ⇒ bool⟩ ⇒
⟨*match J*[*uncurry*] *in*
J[*curry*]: ⟨- ⟹ S2 ≤ ⇓ {(T2, T1). R T2 T1} S1⟩ *for S2* :: ⟨′c nres⟩ *and R* ⇒
⟨*insert Down-add-assumption-beginning*[*where Q′* = *Q′ and P* = *P and R* = *R and*
W = *S2 and V* = *S1 and U* = *S0*,
OF - I J];
insert Down-del-assumption-beginning[*where Q* = ⟨λS -. P S⟩ *and Q′* = *Q′ and V* = *S1 and*
U = *S0*, *OF I*];
unify-Down-invs2-normalisation-post⟩
| - ⇒ ⟨*fail*⟩⟩
| *I*[*thin*]: ⟨S1 ≤ ⇓ {(T1, T0). Q T0 T1∧ Q′ T1 T0} S0⟩ (*multi*) **and**
J: - **for** *S1*:: ⟨′b nres⟩ **and** *S0* :: ⟨′a nres⟩ **and** *Q Q′* ⇒
⟨*match J*[*uncurry*] *in*
J[*curry*]: ⟨- ⟹ S2 ≤ ⇓ {(T2, T1). R T2 T1} S1⟩ *for S2* :: ⟨′c nres⟩ *and R* ⇒
⟨*insert Down-del-assumption-beginning*[*where Q* = ⟨λ x y. Q y x⟩ *and Q′* = *Q′*, *OF I*];
unify-Down-invs2-normalisation-post⟩
| - ⇒ ⟨*fail*⟩⟩
)

Example:

lemma
assumes
⟨*nofail S0*⟩ **and**
1: ⟨S1 ≤ ⇓ {(T1, T0). Q T1 T0 ∧ P T1 ∧ P′ T1 ∧ P′′′ T1 ∧ Q′ T1 T0 ∧ P42 T1} S0⟩ **and**
2: ⟨S2 ≤ ⇓ {(T2, T1). R T2 T1} S1⟩
shows ⟨S2
≤ ⇓ {(T2, T1).
R T2 T1 ∧
P T1 ∧ P′ T1 ∧ P′′′ T1 ∧ P42 T1}
S1⟩

using *assms* **apply** −
apply *unify-Down-invs2*+
apply *fast*
done

Inversion Tactics

lemma *refinement-trans-long*:
‹$A = A' \Longrightarrow B = B' \Longrightarrow R \subseteq R' \Longrightarrow A \le \Downarrow R\ B \Longrightarrow A' \le \Downarrow R'\ B'$›
by (*meson pw-ref-iff subsetCE*)

lemma *mem-set-trans*:
‹$A \subseteq B \Longrightarrow a \in A \Longrightarrow a \in B$›
by *auto*

lemma *fun-rel-syn-invert*:
‹$a = a' \Longrightarrow b \subseteq b' \Longrightarrow a \rightarrow b \subseteq a' \rightarrow b'$›
by (*auto simp*: *refine-rel-defs*)

lemma *fref-param1*: $R{\rightarrow}S = \mathit{fref}\ (\lambda\text{-}.\ \mathit{True})\ R\ S$
by (*auto simp*: *fref-def fun-relD*)

lemma *fref-syn-invert*:
‹$a = a' \Longrightarrow b \subseteq b' \Longrightarrow a \rightarrow_f b \subseteq a' \rightarrow_f b'$›
unfolding *fref-param1*[*symmetric*]
by (*rule fun-rel-syn-invert*)

lemma *nres-rel-mono*:
‹$a \subseteq a' \Longrightarrow \langle a\rangle\ \mathit{nres\text{-}rel} \subseteq \langle a'\rangle\ \mathit{nres\text{-}rel}$›
by (*fastforce simp*: *refine-rel-defs nres-rel-def pw-ref-iff*)

method *match-spec* =
(*match* **conclusion in** ‹$(f,\ g) \in R$› **for** $f\ g\ R$ ⇒
‹*print-term f*; *match premises in I*[*thin*]: ‹$(f,\ g) \in R'$› *for* R'
⇒ ‹*print-term* R'; *rule mem-set-trans*[*OF - I*]››)

method *match-fun-rel* =
((*match* **conclusion in**
‹$- \rightarrow - \subseteq - \rightarrow -$› ⇒ ‹*rule fun-rel-mono*›
| ‹$- \rightarrow_f - \subseteq - \rightarrow_f -$› ⇒ ‹*rule fref-syn-invert*›
| ‹$\langle - \rangle \mathit{nres\text{-}rel} \subseteq \langle - \rangle \mathit{nres\text{-}rel}$› ⇒ ‹*rule nres-rel-mono*›
| ‹$[-]_f\ - \rightarrow - \subseteq [-]_f\ - \rightarrow -$› ⇒ ‹*rule fref-mono*›
)+)

lemma *weaken-SPEC2*: ‹$m' \le \mathit{SPEC}\ \Phi \Longrightarrow m = m' \Longrightarrow (\bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x) \Longrightarrow m \le \mathit{SPEC}\ \Psi$›
using *weaken-SPEC* **by** *auto*

method *match-spec-trans* =
(*match* **conclusion in** ‹$f \le \mathit{SPEC}\ R$› **for** f :: ‹*'a nres*› **and** R :: ‹$'a \Rightarrow \mathit{bool}$› ⇒
‹*print-term f*; *match premises in I*: ‹$- \Longrightarrow - \Longrightarrow f' \le \mathit{SPEC}\ R'$› *for* f' :: ‹*'a nres*› *and* R' :: ‹$'a \Rightarrow$ *bool*›
⇒ ‹*print-term* f'; *rule weaken-SPEC2*[*of* $f'\ R'\ f\ R$]››)

0.0.2 More Notations

abbreviation *uncurry2 f* ≡ *uncurry* (*uncurry f*)

abbreviation *curry2 f* ≡ *curry* (*curry f*)
abbreviation *uncurry3 f* ≡ *uncurry* (*uncurry2 f*)
abbreviation *curry3 f* ≡ *curry* (*curry2 f*)
abbreviation *uncurry4 f* ≡ *uncurry* (*uncurry3 f*)
abbreviation *curry4 f* ≡ *curry* (*curry3 f*)
abbreviation *uncurry5 f* ≡ *uncurry* (*uncurry4 f*)
abbreviation *curry5 f* ≡ *curry* (*curry4 f*)
abbreviation *uncurry6 f* ≡ *uncurry* (*uncurry5 f*)
abbreviation *curry6 f* ≡ *curry* (*curry5 f*)
abbreviation *uncurry7 f* ≡ *uncurry* (*uncurry6 f*)
abbreviation *curry7 f* ≡ *curry* (*curry6 f*)
abbreviation *uncurry8 f* ≡ *uncurry* (*uncurry7 f*)
abbreviation *curry8 f* ≡ *curry* (*curry7 f*)
abbreviation *uncurry9 f* ≡ *uncurry* (*uncurry8 f*)
abbreviation *curry9 f* ≡ *curry* (*curry8 f*)
abbreviation *uncurry10 f* ≡ *uncurry* (*uncurry9 f*)
abbreviation *curry10 f* ≡ *curry* (*curry9 f*)
abbreviation *uncurry11 f* ≡ *uncurry* (*uncurry10 f*)
abbreviation *curry11 f* ≡ *curry* (*curry10 f*)
abbreviation *uncurry12 f* ≡ *uncurry* (*uncurry11 f*)
abbreviation *curry12 f* ≡ *curry* (*curry11 f*)
abbreviation *uncurry13 f* ≡ *uncurry* (*uncurry12 f*)
abbreviation *curry13 f* ≡ *curry* (*curry12 f*)
abbreviation *uncurry14 f* ≡ *uncurry* (*uncurry13 f*)
abbreviation *curry14 f* ≡ *curry* (*curry13 f*)
abbreviation *uncurry15 f* ≡ *uncurry* (*uncurry14 f*)
abbreviation *curry15 f* ≡ *curry* (*curry14 f*)
abbreviation *uncurry16 f* ≡ *uncurry* (*uncurry15 f*)
abbreviation *curry16 f* ≡ *curry* (*curry15 f*)
abbreviation *uncurry17 f* ≡ *uncurry* (*uncurry16 f*)
abbreviation *curry17 f* ≡ *curry* (*curry16 f*)
abbreviation *uncurry18 f* ≡ *uncurry* (*uncurry17 f*)
abbreviation *curry18 f* ≡ *curry* (*curry17 f*)
abbreviation *uncurry19 f* ≡ *uncurry* (*uncurry18 f*)
abbreviation *curry19 f* ≡ *curry* (*curry18 f*)
abbreviation *uncurry20 f* ≡ *uncurry* (*uncurry19 f*)
abbreviation *curry20 f* ≡ *curry* (*curry19 f*)

abbreviation *comp4* (**infixl** *oooo 55*) **where** *f oooo g* ≡ λ*x*. *f ooo* (*g x*)
abbreviation *comp5* (**infixl** *ooooo 55*) **where** *f ooooo g* ≡ λ*x*. *f oooo* (*g x*)
abbreviation *comp6* (**infixl** *oooooo 55*) **where** *f oooooo g* ≡ λ*x*. *f ooooo* (*g x*)
abbreviation *comp7* (**infixl** *ooooooo 55*) **where** *f ooooooo g* ≡ λ*x*. *f oooooo* (*g x*)
abbreviation *comp8* (**infixl** *oooooooo 55*) **where** *f oooooooo g* ≡ λ*x*. *f ooooooo* (*g x*)
abbreviation *comp9* (**infixl** *ooooooooo 55*) **where** *f ooooooooo g* ≡ λ*x*. *f oooooooo* (*g x*)
abbreviation *comp10* (**infixl** *oooooooooo 55*) **where** *f oooooooooo g* ≡ λ*x*. *f ooooooooo* (*g x*)
abbreviation *comp11* (**infixl** o_{11} *55*) **where** *f* o_{11} *g* ≡ λ*x*. *f oooooooooo* (*g x*)
abbreviation *comp12* (**infixl** o_{12} *55*) **where** *f* o_{12} *g* ≡ λ*x*. *f* o_{11} (*g x*)
abbreviation *comp13* (**infixl** o_{13} *55*) **where** *f* o_{13} *g* ≡ λ*x*. *f* o_{12} (*g x*)
abbreviation *comp14* (**infixl** o_{14} *55*) **where** *f* o_{14} *g* ≡ λ*x*. *f* o_{13} (*g x*)
abbreviation *comp15* (**infixl** o_{15} *55*) **where** *f* o_{15} *g* ≡ λ*x*. *f* o_{14} (*g x*)
abbreviation *comp16* (**infixl** o_{16} *55*) **where** *f* o_{16} *g* ≡ λ*x*. *f* o_{15} (*g x*)
abbreviation *comp17* (**infixl** o_{17} *55*) **where** *f* o_{17} *g* ≡ λ*x*. *f* o_{16} (*g x*)
abbreviation *comp18* (**infixl** o_{18} *55*) **where** *f* o_{18} *g* ≡ λ*x*. *f* o_{17} (*g x*)
abbreviation *comp19* (**infixl** o_{19} *55*) **where** *f* o_{19} *g* ≡ λ*x*. *f* o_{18} (*g x*)
abbreviation *comp20* (**infixl** o_{20} *55*) **where** *f* o_{20} *g* ≡ λ*x*. *f* o_{19} (*g x*)

notation
comp4 (**infixl** $\circ\circ\circ$ *55*) **and**
comp5 (**infixl** $\circ\circ\circ\circ$ *55*) **and**
comp6 (**infixl** $\circ\circ\circ\circ\circ$ *55*) **and**
comp7 (**infixl** $\circ\circ\circ\circ\circ\circ$ *55*) **and**
comp8 (**infixl** $\circ\circ\circ\circ\circ\circ\circ$ *55*) **and**
comp9 (**infixl** $\circ\circ\circ\circ\circ\circ\circ\circ$ *55*) **and**
comp10 (**infixl** $\circ\circ\circ\circ\circ\circ\circ\circ\circ$ *55*) **and**
comp11 (**infixl** $\circ_{11}$ *55*) **and**
comp12 (**infixl** $\circ_{12}$ *55*) **and**
comp13 (**infixl** $\circ_{13}$ *55*) **and**
comp14 (**infixl** $\circ_{14}$ *55*) **and**
comp15 (**infixl** $\circ_{15}$ *55*) **and**
comp16 (**infixl** $\circ_{16}$ *55*) **and**
comp17 (**infixl** $\circ_{17}$ *55*) **and**
comp18 (**infixl** $\circ_{18}$ *55*) **and**
comp19 (**infixl** $\circ_{19}$ *55*) **and**
comp20 (**infixl** $\circ_{20}$ *55*)

0.0.3 More Theorems for Refinement

lemma *SPEC-add-information*: ‹$P \Longrightarrow A \leq SPEC\ Q \Longrightarrow A \leq SPEC(\lambda x.\ Q\ x \wedge P)$›
by *auto*

lemma *bind-refine-spec*: ‹$(\bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq \Downarrow R\ M) \Longrightarrow M' \leq SPEC\ \Phi \Longrightarrow M' \gg= f \leq \Downarrow R\ M$›
by (*auto simp add*: *pw-le-iff refine-pw-simps*)

lemma *intro-spec-iff*:
‹$(RES\ X \gg= f \leq M) = (\forall x \in X.\ f\ x \leq M)$›
using *intro-spec-refine-iff*[*of X f Id M*] **by** *auto*

lemma *case-prod-bind*:
assumes ‹$\bigwedge x1\ x2.\ x = (x1,\ x2) \Longrightarrow f\ x1\ x2 \leq \Downarrow R\ I$›
shows ‹$(case\ x\ of\ (x1,\ x2) \Rightarrow f\ x1\ x2) \leq \Downarrow R\ I$›
using *assms* **by** (*cases x*) *auto*

lemma (**in** *transfer*) *transfer-bool*[*refine-transfer*]:
assumes α *fa* $\leq$ *Fa*
assumes α *fb* $\leq$ *Fb*
shows α (*case-bool fa fb x*) $\leq$ *case-bool Fa Fb x*
using *assms* **by** (*auto split*: *bool.split*)

lemma *ref-two-step'*: ‹$A \leq B \Longrightarrow \Downarrow R\ A \leq\ \Downarrow R\ B$›
by (*auto intro*: *ref-two-step*)

lemma *RES-RETURN-RES*: ‹$RES\ \Phi \gg= (\lambda T.\ RETURN\ (f\ T)) = RES\ (f\ `\ \Phi)$›
by (*simp add*: *bind-RES-RETURN-eq setcompr-eq-image*)

lemma *RES-RES-RETURN-RES*: ‹$RES\ A \gg= (\lambda T.\ RES\ (f\ T)) = RES\ (\bigcup (f\ `\ A))$›
by (*auto simp*: *pw-eq-iff refine-pw-simps*)

lemma *RES-RES2-RETURN-RES*: ‹$RES\ A \gg= (\lambda(T,\ T').\ RES\ (f\ T\ T')) = RES\ (\bigcup (uncurry\ f\ `\ A))$›
by (*auto simp*: *pw-eq-iff refine-pw-simps uncurry-def*)

lemma *RES-RES3-RETURN-RES*:

$\langle RES\ A \gg\!= (\lambda(T,\ T',\ T'').\ RES\ (f\ T\ T'\ T'')) = RES\ (\bigcup((\lambda(a,\ b,\ c).\ f\ a\ b\ c)\ `\ A))\rangle$
by (*auto simp*: *pw-eq-iff refine-pw-simps uncurry-def*)

lemma *RES-RETURN-RES3*:
$\langle SPEC\ \Phi \gg\!= (\lambda(T,\ T',\ T'').\ RETURN\ (f\ T\ T'\ T'')) = RES\ ((\lambda(a,\ b,\ c).\ f\ a\ b\ c)\ `\ \{T.\ \Phi\ T\})\rangle$
using *RES-RETURN-RES*[*of* $\langle Collect\ \Phi\rangle$ $\langle\lambda(a,\ b,\ c).\ f\ a\ b\ c\rangle$]
apply (*subst* (*asm*)(*2*) *split-prod-bound*)
apply (*subst* (*asm*)(*3*) *split-prod-bound*)
by *auto*

lemma *RES-RES-RETURN-RES2*: $\langle RES\ A \gg\!= (\lambda(T,\ T').\ RETURN\ (f\ T\ T')) = RES\ (uncurry\ f\ `\ A)\rangle$
by (*auto simp*: *pw-eq-iff refine-pw-simps uncurry-def*)

lemma *bind-refine-res*: $\langle(\bigwedge x.\ x \in \Phi \Longrightarrow f\ x \le \Downarrow R\ M) \Longrightarrow M' \le RES\ \Phi \Longrightarrow M' \gg\!= f \le \Downarrow R\ M\rangle$
by (*auto simp add*: *pw-le-iff refine-pw-simps*)

lemma *RES-RETURN-RES-RES2*:
$\langle RES\ \Phi \gg\!= (\lambda(T,\ T').\ RETURN\ (f\ T\ T')) = RES\ (uncurry\ f\ `\ \Phi)\rangle$
using *RES-RES2-RETURN-RES*[*of* $\langle\Phi\rangle$ $\langle\lambda T\ T'.\ \{f\ T\ T'\}\rangle$]
apply (*subst* (*asm*)(*2*) *split-prod-bound*)
by (*auto simp*: *RETURN-def uncurry-def*)

This theorem adds the invariant at the beginning of next iteration to the current invariant, i.e., the invariant is added as a post-condition on the current iteration.

This is useful to reduce duplication in theorems while refining.

lemma *RECT-WHILEI-body-add-post-condition*:
$\langle REC_T\ (WHILEI\text{-}body\ (\gg\!=)\ RETURN\ I'\ b'\ f)\ x' =$
$(REC_T\ (WHILEI\text{-}body\ (\gg\!=)\ RETURN\ (\lambda x'.\ I'\ x' \wedge (b'\ x' \longrightarrow f\ x' = FAIL \vee f\ x' \le SPEC\ I'))\ b'\ f)\ x')\rangle$
(**is** $\langle REC_T\ ?f\ x' = REC_T\ ?f'\ x'\rangle$)
proof −
have *le*: $\langle flatf\text{-}gfp\ ?f\ x' \le flatf\text{-}gfp\ ?f'\ x'\rangle$ **for** x'
proof (*induct arbitrary*: x' *rule*: *flatf-ord.fixp-induct*[**where** $b = top$ **and** $f = ?f'$])
case *1*
then show *?case*
unfolding *fun-lub-def pw-le-iff*
by (*rule ccpo.admissibleI*)
(*smt chain-fun flat-lub-in-chain mem-Collect-eq nofail-simps*(*1*))
next
case *2*
then show *?case* **by** (*auto simp*: *WHILEI-mono-ge*)
next
case *3*
then show *?case* **by** *simp*
next
case (*4* *x*)
have $\langle(RES\ X \gg\!= f \le M) = (\forall x{\in}X.\ f\ x \le M)\rangle$ **for** $x\ f\ M\ X$
using *intro-spec-refine-iff*[*of* - - $\langle Id\rangle$] **by** *auto*
thm *bind-refine-RES*(*2*)[*of* - *Id*, *simplified*]
have [*simp*]: $\langle flatf\text{-}mono\ FAIL\ (WHILEI\text{-}body\ (\gg\!=)\ RETURN\ I'\ b'\ f)\rangle$
by (*simp add*: *WHILEI-mono-ge*)

have $\langle flatf\text{-}gfp\ ?f\ x' = ?f\ (?f\ (flatf\text{-}gfp\ ?f))\ x'\rangle$

apply (*subst flatf-ord.fixp-unfold*)
 apply (*solves* ‹*simp*›)
apply (*subst flatf-ord.fixp-unfold*)
 apply (*solves* ‹*simp*›)
..
also have ‹... = *WHILEI-body* ($\gg=$) *RETURN* ($\lambda x'$. I' x' $\wedge$ (b' x' $\longrightarrow$ f x' = *FAIL* $\vee$ f x' $\leq$ *SPEC* I')) b' f (*WHILEI-body* ($\gg=$) *RETURN* I' b' f (*flatf-gfp* (*WHILEI-body* ($\gg=$) *RETURN* I' b' f))) x'›
apply (*subst* (*1*) *WHILEI-body-def*, *subst* (*1*) *WHILEI-body-def*)
apply (*subst* (*2*) *WHILEI-body-def*, *subst* (*2*) *WHILEI-body-def*)
apply *simp-all*
apply (*cases* ‹f x'›)
 apply (*auto simp*: *RES-RETURN-RES nofail-def*[*symmetric*] *pw-RES-bind-choose*
split: *if-splits*)
done
also have ‹... = *WHILEI-body* ($\gg=$) *RETURN* ($\lambda x'$. I' x' $\wedge$ (b' x' $\longrightarrow$ f x' = *FAIL* $\vee$ f x' $\leq$ *SPEC* I')) b' f ((*flatf-gfp* (*WHILEI-body* ($\gg=$) *RETURN* I' b' f))) x'›
apply (*subst* (*2*) *flatf-ord.fixp-unfold*)
 apply (*solves* ‹*simp*›)
..
finally have *unfold1*: ‹*flatf-gfp* (*WHILEI-body* ($\gg=$) *RETURN* I' b' f) x' =
?f' (*flatf-gfp* (*WHILEI-body* ($\gg=$) *RETURN* I' b' f)) x'›
.
have [*intro*!]: ‹($\bigwedge x$. g x $\leq$ (h:: $'a \Rightarrow 'a$ *nres*) x) $\Longrightarrow$ *fx* $\gg=$ g $\leq$ *fx* $\gg=$ h› **for** g h *fx fy*
by (*refine-rcg bind-refine'*[**where** R = ‹*Id*›, *simplified*]) *fast*
show *?case*
apply (*subst unfold1*)
using *4* **unfolding** *WHILEI-body-def* **by** *auto*
qed

have *ge*: ‹*flatf-gfp ?f* x' $\geq$ *flatf-gfp ?f'* x'› **for** x'
proof (*induct arbitrary*: x' *rule*: *flatf-ord.fixp-induct*[**where** b = *top* **and**
f = *?f*])
case *1*
then show *?case*
unfolding *fun-lub-def pw-le-iff*
by (*rule ccpo.admissibleI*) (*smt chain-fun flat-lub-in-chain mem-Collect-eq nofail-simps*(*1*))
next
case *2*
then show *?case* **by** (*auto simp*: *WHILEI-mono-ge*)
next
case *3*
then show *?case* **by** *simp*
next
case (*4* x)
have ‹(*RES* X $\gg=$ f $\leq$ M) = ($\forall x \in X$. f x $\leq$ M)› **for** x f M X
using *intro-spec-refine-iff*[*of* - - ‹*Id*›] **by** *auto*
thm *bind-refine-RES*(*2*)[*of* - *Id*, *simplified*]
have [*simp*]: ‹*flatf-mono FAIL ?f'*›
by (*simp add*: *WHILEI-mono-ge*)
have H: ‹A = *FAIL* $\longleftrightarrow$ $\neg$*nofail* A› **for** A **by** (*auto simp*: *nofail-def*)
have ‹*flatf-gfp ?f'* x' = *?f'* (*?f'* (*flatf-gfp ?f'*)) x'›
apply (*subst flatf-ord.fixp-unfold*)
 apply (*solves* ‹*simp*›)
apply (*subst flatf-ord.fixp-unfold*)
 apply (*solves* ‹*simp*›)
..

also have ‹... = *?f* (*?f′* (*flatf-gfp* *?f′*)) *x′*›
apply (*subst* (*1*) *WHILEI-body-def*, *subst* (*1*) *WHILEI-body-def*)
apply (*subst* (*2*) *WHILEI-body-def*, *subst* (*2*) *WHILEI-body-def*)
apply *simp-all*
apply (*cases* ‹*f x′*›)
apply (*auto simp*: *RES-RETURN-RES nofail-def*[*symmetric*] *pw-RES-bind-choose*
eq-commute[*of* ‹*FAIL*›] *H*
split: *if-splits*
cong: *if-cong*)
done
also have ‹... = *?f* (*flatf-gfp* *?f′*) *x′*›
apply (*subst* (*2*) *flatf-ord.fixp-unfold*)
apply (*solves* ‹*simp*›)
..
finally have *unfold1*: ‹*flatf-gfp* *?f′* *x′* =
?f (*flatf-gfp* *?f′*) *x′*›
.
have [*intro*!]: ‹($\bigwedge x.\ g\ x \le (h{::}\ 'a \Rightarrow 'a\ nres)\ x) \Longrightarrow fx \gg\!= g \le fx \gg\!= h$› **for** *g h fx fy*
by (*refine-rcg bind-refine′*[**where** *R* = ‹*Id*›, *simplified*]) *fast*
show *?case*
apply (*subst unfold1*)
using *4*
unfolding *WHILEI-body-def*
by (*auto intro*: *bind-refine′*[**where** *R* = ‹*Id*›, *simplified*])
qed
show *?thesis*
unfolding *RECT-def*
using *le*[*of x′*] *ge*[*of x′*] **by** (*auto simp*: *WHILEI-body-trimono*)
qed

lemma *WHILEIT-add-post-condition*:
‹(*WHILEIT I′ b′ f′ x′*) =
(*WHILEIT* ($\lambda x'.\ I'\ x' \wedge (b'\ x' \longrightarrow f'\ x' = FAIL \vee f'\ x' \le SPEC\ I')$)
b′ f′ x′)›
unfolding *WHILEIT-def*
apply (*subst RECT-WHILEI-body-add-post-condition*)
..

lemma *WHILEIT-rule-stronger-inv*:
assumes
‹*wf R*› **and**
‹*I s*› **and**
‹*I′ s*› **and**
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \le SPEC\ (\lambda s'.\ I\ s' \wedge\ I'\ s' \wedge (s', s) \in R)$› **and**
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow \neg\ b\ s \Longrightarrow \Phi\ s$›
shows ‹$WHILE_T{}^I\ b\ f\ s \le SPEC\ \Phi$›
proof −
have ‹$WHILE_T{}^I\ b\ f\ s \le WHILE_T{}^{\lambda s.\ I\ s\ \wedge\ I'\ s}\ b\ f\ s$›
by (*metis* (*mono-tags*, *lifting*) *WHILEIT-weaken*)
also have ‹$WHILE_T{}^{\lambda s.\ I\ s\ \wedge\ I'\ s}\ b\ f\ s \le SPEC\ \Phi$›
by (*rule WHILEIT-rule*) (*use assms* **in** ‹*auto simp*: ›)
finally show *?thesis* **.**
qed

lemma *RES-RETURN-RES2*:

‹$SPEC\ \Phi \gg= (\lambda(T,\ T').\ RETURN\ (f\ T\ T')) = RES\ (uncurry\ f\ `\ \{T.\ \Phi\ T\})$›
using *RES-RETURN-RES*[*of* ‹*Collect* Φ› ‹*uncurry f*›]
apply (*subst* (*asm*)(*2*) *split-prod-bound*)
by *auto*

lemma *WHILEIT-rule-stronger-inv-RES*:
assumes
‹*wf R*› **and**
‹*I s*› **and**
‹$I'\ s$›
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \leq SPEC\ (\lambda s'.\ I\ s' \wedge\ I'\ s' \wedge (s',\ s) \in R)$› **and**
‹$\bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow \neg\ b\ s \Longrightarrow s \in \Phi$›
shows ‹$WHILE_T{}^I\ b\ f\ s \leq RES\ \Phi$›
proof −
have *RES-SPEC*: ‹$RES\ \Phi = SPEC(\lambda s.\ s \in \Phi)$›
by *auto*
have ‹$WHILE_T{}^I\ b\ f\ s \leq WHILE_T{}^{\lambda s.\ I\ s\ \wedge\ I'\ s}\ b\ f\ s$›
by (*metis* (*mono-tags*, *lifting*) *WHILEIT-weaken*)
also have ‹$WHILE_T{}^{\lambda s.\ I\ s\ \wedge\ I'\ s}\ b\ f\ s \leq RES\ \Phi$›
unfolding *RES-SPEC*
by (*rule WHILEIT-rule*) (*use assms* **in** ‹*auto simp*: ›)
finally show *?thesis* **.**
qed

lemma *fref-weaken-pre-weaken*:
assumes $\bigwedge x.\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in fref\ P'\ R\ S$
assumes ‹$S \subseteq S'$›
shows $(f,h) \in fref\ P\ R\ S'$
using *assms* **unfolding** *fref-def* **by** *blast*

lemma *bind-rule-complete-RES*: ‹$(M \gg= f \leq RES\ \Phi) = (M \leq SPEC\ (\lambda x.\ f\ x \leq RES\ \Phi))$›
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *fref-to-Down*:
‹$(f,\ g) \in [P]_f\ A \rightarrow \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'.\ P\ x' \Longrightarrow (x,\ x') \in A \Longrightarrow f\ x \leq \Downarrow B\ (g\ x'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry-left*:
fixes f:: ‹$'a \Rightarrow\ 'b \Rightarrow\ 'c\ nres$› **and**
A::‹$(('a \times\ 'b) \times\ 'd)\ set$›
shows
‹$(uncurry\ f,\ g) \in [P]_f\ A \rightarrow \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge a\ b\ x'.\ P\ x' \Longrightarrow ((a,\ b),\ x') \in A \Longrightarrow f\ a\ b \leq \Downarrow B\ (g\ x'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry*:
‹$(uncurry\ f,\ uncurry\ g) \in [P]_f\ A \rightarrow \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow f\ x\ y \leq \Downarrow B\ (g\ x'\ y'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry2*:
‹$(\mathit{uncurry2}\ f,\ \mathit{uncurry2}\ g) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'.\ P\ ((x',\ y'),\ z') \Longrightarrow (((x,\ y),\ z),\ ((x',\ y'),\ z')) \in A \Longrightarrow$
$f\ x\ y\ z \le \Downarrow B\ (g\ x'\ y'\ z'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry2'*:
‹$(\mathit{uncurry2}\ f,\ \mathit{uncurry2}\ g) \in A \to_f \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'.\ (((x,\ y),\ z),\ ((x',\ y'),\ z')) \in A \Longrightarrow$
$f\ x\ y\ z \le \Downarrow B\ (g\ x'\ y'\ z'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry3*:
‹$(\mathit{uncurry3}\ f,\ \mathit{uncurry3}\ g) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'\ a\ a'.\ P\ (((x',\ y'),\ z'),\ a') \Longrightarrow$
$((((x,\ y),\ z),\ a),\ (((x',\ y'),\ z'),\ a')) \in A \Longrightarrow$
$f\ x\ y\ z\ a \le \Downarrow B\ (g\ x'\ y'\ z'\ a'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry4*:
‹$(\mathit{uncurry4}\ f,\ \mathit{uncurry4}\ g) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'\ a\ a'\ b\ b'.\ P\ ((((x',\ y'),\ z'),\ a'),\ b') \Longrightarrow$
$(((((x,\ y),\ z),\ a),\ b),\ ((((x',\ y'),\ z'),\ a'),\ b')) \in A \Longrightarrow$
$f\ x\ y\ z\ a\ b \le \Downarrow B\ (g\ x'\ y'\ z'\ a'\ b'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry5*:
‹$(\mathit{uncurry5}\ f,\ \mathit{uncurry5}\ g) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'\ a\ a'\ b\ b'\ c\ c'.\ P\ (((((x',\ y'),\ z'),\ a'),\ b'),\ c') \Longrightarrow$
$((((((x,\ y),\ z),\ a),\ b),\ c),\ (((((x',\ y'),\ z'),\ a'),\ b'),\ c')) \in A \Longrightarrow$
$f\ x\ y\ z\ a\ b\ c \le \Downarrow B\ (g\ x'\ y'\ z'\ a'\ b'\ c'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry6*:
‹$(\mathit{uncurry6}\ f,\ \mathit{uncurry6}\ g) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'\ a\ a'\ b\ b'\ c\ c'\ d\ d'.\ P\ ((((((x',\ y'),\ z'),\ a'),\ b'),\ c'),\ d') \Longrightarrow$
$(((((((x,\ y),\ z),\ a),\ b),\ c),\ d),\ ((((((x',\ y'),\ z'),\ a'),\ b'),\ c'),\ d')) \in A \Longrightarrow$
$f\ x\ y\ z\ a\ b\ c\ d \le \Downarrow B\ (g\ x'\ y'\ z'\ a'\ b'\ c'\ d'))$›
unfolding *fref-def uncurry-def nres-rel-def* **by** *auto*

lemma *fref-to-Down-curry7*:
‹$(\mathit{uncurry7}\ f,\ \mathit{uncurry7}\ g) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'\ a\ a'\ b\ b'\ c\ c'\ d\ d'\ e\ e'.\ P\ (((((((x',\ y'),\ z'),\ a'),\ b'),\ c'),\ d'),\ e') \Longrightarrow$
$((((((((x,\ y),\ z),\ a),\ b),\ c),\ d),\ e),\ (((((((x',\ y'),\ z'),\ a'),\ b'),\ c'),\ d'),\ e')) \in A \Longrightarrow$
$f\ x\ y\ z\ a\ b\ c\ d\ e \le \Downarrow B\ (g\ x'\ y'\ z'\ a'\ b'\ c'\ d'\ e'))$›
unfolding *fref-def uncurry-def nres-rel-def* **by** *auto*

lemma *fref-to-Down-explode*:
‹$(f\ a,\ g\ a) \in [P]_f\ A \to \langle B\rangle\mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ b.\ P\ x' \Longrightarrow (x,\ x') \in A \Longrightarrow b = a \Longrightarrow f\ a\ x \le \Downarrow B\ (g\ b\ x'))$›

unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry-no-nres-Id*:
‹$(uncurry\ (RETURN\ oo\ f),\ uncurry\ (RETURN\ oo\ g)) \in [P]_f\ A \to \langle Id\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow f\ x\ y = g\ x'\ y')$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-no-nres*:
‹$((RETURN\ o\ f),\ (RETURN\ o\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'.\ P\ (x') \Longrightarrow (x,\ x') \in A \Longrightarrow (f\ x,\ g\ x') \in B)$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-curry-no-nres*:
‹$(uncurry\ (RETURN\ oo\ f),\ uncurry\ (RETURN\ oo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow (f\ x\ y,\ g\ x'\ y') \in B)$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *RES-RETURN-RES4*:
‹$SPEC\ \Phi \gg\!= (\lambda(T,\ T',\ T'',\ T''').\ RETURN\ (f\ T\ T'\ T''\ T''')) =$
$RES\ ((\lambda(a,\ b,\ c,\ d).\ f\ a\ b\ c\ d)\ `\ \{T.\ \Phi\ T\})$›
using *RES-RETURN-RES*[*of* ‹*Collect* Φ› ‹$\lambda(a,\ b,\ c,\ d).\ f\ a\ b\ c\ d$›]
apply (*subst* (*asm*)(*2*) *split-prod-bound*)
apply (*subst* (*asm*)(*3*) *split-prod-bound*)
apply (*subst* (*asm*)(*4*) *split-prod-bound*)
by *auto*

declare *RETURN-as-SPEC-refine*[*refine2 del*]

lemma *fref-to-Down-unRET-uncurry-Id*:
‹$(uncurry\ (RETURN\ oo\ f),\ uncurry\ (RETURN\ oo\ g)) \in [P]_f\ A \to \langle Id\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow f\ x\ y = (g\ x'\ y'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*
lemma *fref-to-Down-unRET-uncurry*:
‹$(uncurry\ (RETURN\ oo\ f),\ uncurry\ (RETURN\ oo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow (f\ x\ y,\ g\ x'\ y') \in B)$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-unRET-Id*:
‹$((RETURN\ o\ f),\ (RETURN\ o\ g)) \in [P]_f\ A \to \langle Id\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'.\ P\ x' \Longrightarrow (x,\ x') \in A \Longrightarrow f\ x = (g\ x'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-unRET*:
‹$((RETURN\ o\ f),\ (RETURN\ o\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'.\ P\ x' \Longrightarrow (x,\ x') \in A \Longrightarrow (f\ x,\ g\ x') \in B)$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *fref-to-Down-unRET-uncurry2*:
  fixes f :: ‹$'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'f$›
    and g :: ‹$'a2 \Rightarrow 'b2 \Rightarrow 'c2 \Rightarrow 'g$›
  shows
    ‹$(uncurry2\ (RETURN\ ooo\ f),\ uncurry2\ (RETURN\ ooo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
      $(\bigwedge(x :: 'a)\ x'\ y\ y'\ (z :: 'c)\ (z' :: 'c2).$
        $P\ ((x',\ y'),\ z') \Longrightarrow (((x,\ y),\ z),\ ((x',\ y'),\ z')) \in A \Longrightarrow$
        $(f\ x\ y\ z,\ g\ x'\ y'\ z') \in B)$›
  unfolding *fref-def uncurry-def nres-rel-def*
  by *auto*

lemma *fref-to-Down-unRET-uncurry3*:
  shows
    ‹$(uncurry3\ (RETURN\ oooo\ f),\ uncurry3\ (RETURN\ oooo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
      $(\bigwedge(x :: 'a)\ x'\ y\ y'\ (z :: 'c)\ (z' :: 'c2)\ a\ a'.$
        $P\ (((x',\ y'),\ z'),\ a') \Longrightarrow ((((x,\ y),\ z),\ a),\ (((x',\ y'),\ z'),\ a')) \in A \Longrightarrow$
        $(f\ x\ y\ z\ a,\ g\ x'\ y'\ z'\ a') \in B)$›
  unfolding *fref-def uncurry-def nres-rel-def*
  by *auto*

lemma *fref-to-Down-unRET-uncurry4*:
  shows
    ‹$(uncurry4\ (RETURN\ ooooo\ f),\ uncurry4\ (RETURN\ ooooo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
      $(\bigwedge(x :: 'a)\ x'\ y\ y'\ (z :: 'c)\ (z' :: 'c2)\ a\ a'\ b\ b'.$
        $P\ ((((x',\ y'),\ z'),\ a'),\ b') \Longrightarrow (((((x,\ y),\ z),\ a),\ b),\ ((((x',\ y'),\ z'),\ a'),\ b')) \in A \Longrightarrow$
        $(f\ x\ y\ z\ a\ b,\ g\ x'\ y'\ z'\ a'\ b') \in B)$›
  unfolding *fref-def uncurry-def nres-rel-def*
  by *auto*

More Simplification Theorems

lemma *nofail-Down-nofail*: ‹$nofail\ gS \Longrightarrow fS \le \Downarrow R\ gS \Longrightarrow nofail\ fS$›
  using *pw-ref-iff* **by** *blast*

This is the refinement version of $WHILE_T{}^{?I'}\ ?b'\ ?f'\ ?x' = WHILE_T{}^{\lambda x'.\ ?I'\ x' \wedge (?b'\ x' \longrightarrow ?f'\ x' = FAIL \vee ?f'\ x' \le}$
$?b'\ ?f'\ ?x'$.

lemma *WHILEIT-refine-with-post*:
  assumes *R0*: $I'\ x' \Longrightarrow (x,x')\in R$
  assumes *IREF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I'\ x'\ ]\!] \Longrightarrow I\ x$
  assumes *COND-REF*: $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ I\ x;\ I'\ x'\ ]\!] \Longrightarrow b\ x = b'\ x'$
  assumes *STEP-REF*:
    $\bigwedge x\ x'.\ [\![\ (x,x')\in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x';\ f'\ x' \le SPEC\ I'\ ]\!] \Longrightarrow f\ x \le \Downarrow R\ (f'\ x')$
  shows $WHILEIT\ I\ b\ f\ x \le\Downarrow R\ (WHILEIT\ I'\ b'\ f'\ x')$
  apply (*subst* (*2*) *WHILEIT-add-post-condition*)
  apply (*rule WHILEIT-refine*)
  subgoal using *R0* **by** *blast*
  subgoal using *IREF* **by** *blast*
  subgoal using *COND-REF* **by** *blast*
  subgoal using *STEP-REF* **by** *auto*
  done

0.0.4 Some Refinement

lemma *Collect-eq-comp*: ‹$\{(c,\ a).\ a = f\ c\}\ O\ \{(x,\ y).\ P\ x\ y\} = \{(c,\ y).\ P\ (f\ c)\ y\}$›
  by *auto*

lemma *Collect-eq-comp-right*:
  ‹$\{(x, y). P\ x\ y\}\ O\ \{(c, a). a = f\ c\} = \{(x, c). \exists y. P\ x\ y \wedge c = f\ y\}$ ›
  by *auto*

lemma *no-fail-spec-le-RETURN-itself*: ‹$nofail\ f \Longrightarrow f \leq SPEC(\lambda x. RETURN\ x \leq f)$›
  by (*metis RES-rule nres-order-simps*(*21*) *the-RES-inv*)

lemma *refine-add-invariants′*:
  assumes
    ‹$f\ S \leq \Downarrow \{(S, S'). Q'\ S\ S' \wedge Q\ S\}\ gS$› **and**
    ‹$y \leq \Downarrow \{((i, S), S'). P\ i\ S\ S'\}\ (f\ S)$› **and**
    ‹*nofail gS*›
  shows ‹$y \leq \Downarrow \{((i, S), S'). P\ i\ S\ S' \wedge Q\ S'\}\ (f\ S)$›
  using *assms* **unfolding** *pw-le-iff pw-conc-inres pw-conc-nofail*
  by *force*

lemma *weaken-⇓*: ‹$R' \subseteq R \Longrightarrow f \leq \Downarrow R'\ g \Longrightarrow f \leq \Downarrow R\ g$›
  by (*meson pw-ref-iff subset-eq*)

method *match-Down* =
  (*match* **conclusion in** ‹$f \leq \Downarrow R\ g$› **for** *f g R* ⇒
    ‹*match premises in I*: ‹$f \leq \Downarrow R'\ g$› *for* R'
      ⇒ ‹*rule weaken-⇓*[*OF - I*]››)

lemma *refine-SPEC-refine-Down*:
  ‹$f \leq SPEC\ C \longleftrightarrow f \leq \Downarrow \{(T', T). T = T' \wedge C\ T'\}\ (SPEC\ C)$›
  apply (*rule iffI*)
  subgoal
    by (*rule SPEC-refine*) *auto*
  subgoal
    by (*metis* (*no-types*, *lifting*) *RETURN-ref-SPECD SPEC-cons-rule dual-order.trans*
      in-pair-collect-simp no-fail-spec-le-RETURN-itself nofail-Down-nofail nofail-simps(*2*))
  done

0.0.5 More declarations

notation *prod-rel-syn* (**infixl** $\times_f$ *70*)

lemma *diff-add-mset-remove1*: ‹$NO\text{-}MATCH\ \{\#\}\ N \Longrightarrow M - add\text{-}mset\ a\ N = remove1\text{-}mset\ a\ (M - N)$›
  by *auto*

0.0.6 List relation

lemma *list-rel-take*:
  ‹$(ba, ab) \in \langle A\rangle list\text{-}rel \Longrightarrow (take\ b\ ba, take\ b\ ab) \in \langle A\rangle list\text{-}rel$›
  by (*auto simp*: *list-rel-def*)

lemma *list-rel-update′*:
  fixes *R*
  assumes *rel*: ‹$(xs, ys) \in \langle R\rangle list\text{-}rel$› **and**
   h: ‹$(bi, b) \in R$›
  shows ‹$(list\text{-}update\ xs\ ba\ bi, list\text{-}update\ ys\ ba\ b) \in \langle R\rangle list\text{-}rel$›

proof −
  have [*simp*]: ‹$(bi, b) \in R$›
    using *h* **by** *auto*
  have ‹*length xs* = *length ys*›
    using *assms list-rel-imp-same-length* **by** *blast*

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

lemma *list-rel-in-find-correspondanceE*:
  assumes ‹$(M, M') \in \langle R\rangle$*list-rel*› **and** ‹$L \in$ *set M*›
  obtains L' **where** ‹$(L, L') \in R$› **and** ‹$L' \in$ *set* M'›
  using *assms*[*unfolded in-set-conv-decomp*] **by** (*auto simp*: *list-rel-append1*
    elim!: *list-relE3*)

0.0.7 More Functions, Relations, and Theorems

definition *emptied-list* :: ‹$'a$ *list* ⇒ $'a$ *list*› **where**
  ‹*emptied-list l* = []›

lemma *Down-id-eq*: $\Downarrow$ *Id a* = *a*
  by *auto*

lemma *Down-itself-via-SPEC*:
  assumes ‹$I \le$ *SPEC P*› **and** ‹$\bigwedge x.\ P\ x \Longrightarrow (x, x) \in R$›
  shows ‹$I \le \Downarrow R\ I$›
  using *assms* **by** (*meson inres-SPEC pw-ref-I*)
lemma *RES-ASSERT-moveout*:
  ($\bigwedge a.\ a \in P \Longrightarrow Q\ a) \Longrightarrow$ *do* {*a* ← *RES P*; *ASSERT*(*Q a*); (*f a*)} =
   do {*a*← *RES P*; (*f a*)}
  apply (*subst order-class.eq-iff*)
  apply (*rule conjI*)
  subgoal
    by (*refine-rcg bind-refine-RES*[**where** *R*=*Id*, *unfolded Down-id-eq*])
      auto
  subgoal
    by (*refine-rcg bind-refine-RES*[**where** *R*=*Id*, *unfolded Down-id-eq*])
      auto
  done

lemma *bind-if-inverse*:
  ‹*do* {
    S ← *H*;
    if b then f S else g S
    } =
    (*if b then do* {*S* ← *H*; *f S*} *else do* {*S* ← *H*; *g S*})
  › **for** *H* :: ‹$'a$ *nres*›
  by *auto*

Ghost parameters

This is a trick to recover from consumption of a variable ($\mathcal{A}_{in}$) that is passed as argument and destroyed by the initialisation: We copy it as a zero-cost (by creating a ()), because we don't need it in the code and only in the specification.

This is a way to have ghost parameters, without having them: The parameter is replaced by () and we hope that the compiler will do the right thing.

definition *virtual-copy* **where**
  [*simp*]: ‹*virtual-copy* = *id*›

definition *virtual-copy-rel* **where**
  ‹*virtual-copy-rel* = {(*c*, *b*). *c* = ()}›

lemma *bind-cong-nres*: ‹$(\bigwedge x.\ g\ x = g'\ x) \Longrightarrow (do\ \{a \leftarrow f :: 'a\ nres;\ \ g\ a\}) = (do\ \{a \leftarrow f :: 'a\ nres;\ \ g'\ a\})$›
  by *auto*

lemma *case-prod-cong*:
  ‹$(\bigwedge a\ b.\ f\ a\ b = g\ a\ b) \Longrightarrow (case\ x\ of\ (a,\ b) \Rightarrow f\ a\ b) = (case\ x\ of\ (a,\ b) \Rightarrow g\ a\ b)$›
  by (*cases x*) *auto*

lemma *if-replace-cond*: ‹$(if\ b\ then\ P\ b\ else\ Q\ b) = (if\ b\ then\ P\ True\ else\ Q\ False)$›
  by *auto*

lemma *foldli-cong2*:
  assumes
    le: ‹$length\ l = length\ l'$› **and**
    σ: ‹$\sigma = \sigma'$› **and**
    c: ‹$c = c'$› **and**
    H: ‹$\bigwedge\sigma\ x.\ x < length\ l \Longrightarrow c'\ \sigma \Longrightarrow f\ (l\ !\ x)\ \sigma = f'\ (l'\ !\ x)\ \sigma$›
  shows ‹$foldli\ l\ c\ f\ \sigma = foldli\ l'\ c'\ f'\ \sigma'$›
proof −
  show *?thesis*
    using *le H* **unfolding** *c*[*symmetric*] σ[*symmetric*]
  proof (*induction l arbitrary*: $l'\ \sigma$)
    case *Nil*
    then show *?case* **by** *simp*
  next
    case ($Cons\ a\ l\ l''$) **note** *IH*=*this*(*1*) **and** *le* = *this*(*2*) **and** *H* = *this*(*3*)
    show *?case*
      using *le H*[*of* ‹*Suc* -›] *H*[*of 0*] *IH*[*of* ‹$tl\ l''$› ‹$f'\ (hd\ l'')\ \sigma$›]
      by (*cases* l'') *auto*
  qed
qed

lemma *foldli-foldli-list-nth*:
  ‹$foldli\ xs\ c\ P\ a = foldli\ [0..<length\ xs]\ c\ (\lambda i.\ P\ (xs\ !\ i))\ a$›
proof (*induction xs arbitrary*: *a*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons x xs*) **note** *IH* = *this*(*1*)
  have *1*: ‹$[0..<length\ (x\ \#\ xs)] = 0\ \#\ [1..<length\ (x\#xs)]$›

by (*subst upt-rec*) *simp*
have *2*: ‹[*1*..<*length* (*x*#*xs*)] = *map Suc* [*0*..<*length xs*]›
by (*induction xs*) *auto*
have *AB*: ‹*foldli* [*0*..<*length* (*x* # *xs*)] *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) *a* =
foldli (*0* # [*1*..<*length* (*x*#*xs*)]) *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) *a*›
(**is** ‹*?A* = *?B*›)
unfolding *1* ..
{
assume [*simp*]: ‹*c a*›
have ‹*foldli* (*0* # [*1*..<*length* (*x*#*xs*)]) *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) *a* =
foldli [*1*..<*length* (*x*#*xs*)] *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) (*P x a*)›
by *simp*
also have ‹. . . = *foldli* [*0*..<*length xs*] *c* (λ*i*. *P* (*xs* ! *i*)) (*P x a*)›
unfolding *2*
by (*rule foldli-cong2*) *auto*
finally have ‹*?A* = *foldli* [*0*..<*length xs*] *c* (λ*i*. *P* (*xs* ! *i*)) (*P x a*)›
using *AB*
by *simp*
}
moreover {
assume [*simp*]: ‹¬*c a*›
have ‹*?B* = *a*›
by *simp*
}
ultimately show *?case* **by** (*auto simp*: *IH*)
qed

lemma *RES-RES13-RETURN-RES*: ‹*do* {
(*M*, *N*, *D*, *Q*, *W*, *vm*, φ, *clvls*, *cach*, *lbd*, *outl*, *stats*, *fast-ema*, *slow-ema*, *ccount*,
vdom, *avdom*, *lcount*) ← *RES A*;
RES (*f M N D Q W vm* φ *clvls cach lbd outl stats fast-ema slow-ema ccount*
vdom avdom lcount)
} = *RES* (⋃(*M*, *N*, *D*, *Q*, *W*, *vm*, φ, *clvls*, *cach*, *lbd*, *outl*, *stats*, *fast-ema*, *slow-ema*, *ccount*,
vdom, *avdom*, *lcount*)∈*A*. *f M N D Q W vm* φ *clvls cach lbd outl stats fast-ema slow-ema ccount*
vdom avdom lcount)›
by (*force simp*: *pw-eq-iff refine-pw-simps uncurry-def*)

lemma *RES-SPEC-conv*: ‹*RES P* = *SPEC* (λ*v*. *v* ∈ *P*)›
by *auto*

lemma *add-invar-refineI-P*: ‹*A* ≤ ⇓ {(*x*,*y*). *R x y*} *B* ⟹ (*nofail A* ⟹*A* ≤ *SPEC P*) ⟹ *A* ≤ ⇓ {(*x*,*y*). *R x y* ∧ *P x*} *B*›
using *add-invar-refineI*[*of* ‹λ-. *A*› - - ‹λ-. *B*› *P*, **where** *R*=‹{(*x*,*y*). *R x y*}› **and** *I*=*P*]
by *auto*

lemma (**in** −) *WHILEIT-rule-stronger-inv-RES′*:
assumes
‹*wf R*› **and**
‹*I s*› **and**
‹*I′ s*›
‹⋀*s*. *I s* ⟹ *I′ s* ⟹ *b s* ⟹ *f s* ≤ *SPEC* (λ*s′*. *I s′* ∧ *I′ s′* ∧ (*s′*, *s*) ∈ *R*)› **and**
‹⋀*s*. *I s* ⟹ *I′ s* ⟹ ¬ *b s* ⟹ *RETURN s* ≤ ⇓ *H* (*RES* Φ)›
shows ‹$WHILE_T{}^I$ *b f s* ≤ ⇓ *H* (*RES* Φ)›
proof −

have *RES-SPEC*: ‹*RES* Φ = *SPEC*(λ*s*. *s* ∈ Φ)›
  by *auto*
have ‹$WHILE_T{}^{I}$ *b f s* ≤ $WHILE_T{}^{\lambda s.\ I\ s \wedge I'\ s}$ *b f s*›
  by (*metis* (*mono-tags*, *lifting*) *WHILEIT-weaken*)
also have ‹$WHILE_T{}^{\lambda s.\ I\ s \wedge I'\ s}$ *b f s* ≤ ⇓ *H* (*RES* Φ)›
  unfolding *RES-SPEC conc-fun-SPEC*
  apply (*rule WHILEIT-rule*[*OF assms*(*1*)])
  subgoal using *assms*(*2*,*3*) **by** *auto*
  subgoal using *assms*(*4*) **by** *auto*
  subgoal using *assms*(*5*) **unfolding** *RES-SPEC conc-fun-SPEC* **by** *auto*
  done
finally show *?thesis* **.**
qed

lemma *same-in-Id-option-rel*:
‹*x* = *x'* ⟹ (*x*, *x'*) ∈ ⟨*Id*⟩*option-rel*›
by *auto*

definition *find-in-list-between* :: ‹('*a* ⇒ *bool*) ⇒ *nat* ⇒ *nat* ⇒ '*a list* ⇒ *nat option nres*› **where**
‹*find-in-list-between P a b C* = *do* {
  (*x*, -) ← $WHILE_T{}^{\lambda(found,\ i).\ i \geq a \wedge i \leq length\ C \wedge i \leq b \wedge (\forall j \in \{a..<i\}.\ \neg P\ (C!j)) \wedge \quad (\forall j.\ found = Some\ j \longrightarrow (}$
    (λ(*found*, *i*). *found* = *None* ∧ *i* < *b*)
    (λ(-, *i*). *do* {
      ASSERT(*i* < *length C*);
      if P (*C*!*i*) *then RETURN* (*Some i*, *i*) *else RETURN* (*None*, *i*+*1*)
    })
    (*None*, *a*);
  RETURN x
}›

lemma *find-in-list-between-spec*:
assumes ‹*a* ≤ *length C*› **and** ‹*b* ≤ *length C*› **and** ‹*a* ≤ *b*›
shows
  ‹*find-in-list-between P a b C* ≤ *SPEC*(λ*i*.
    (*i* ≠ *None* ⟶ *P* (*C* ! *the i*) ∧ *the i* ≥ *a* ∧ *the i* < *b*) ∧
    (*i* = *None* ⟶ (∀*j*. *j* ≥ *a* ⟶ *j* < *b* ⟶ ¬*P* (*C*!*j*))))›
unfolding *find-in-list-between-def*
apply (*refine-vcg WHILEIT-rule*[**where** *R* = ‹*measure* (λ(*f*, *i*). *Suc* (*length C*) − (*i* + (*if f* = *None then 0 else 1*)))›])
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *less-Suc-eq*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

lemma *nfoldli-cong2*:
assumes
le: ‹$length\ l = length\ l'$› **and**
σ: ‹$\sigma = \sigma'$› **and**
c: ‹$c = c'$› **and**
H: ‹$\bigwedge\sigma\ x.\ x < length\ l \Longrightarrow c'\ \sigma \Longrightarrow f\ (l\ !\ x)\ \sigma = f'\ (l'\ !\ x)\ \sigma$›
shows ‹$nfoldli\ l\ c\ f\ \sigma = nfoldli\ l'\ c'\ f'\ \sigma'$›
proof −
show *?thesis*
using *le H* **unfolding** *c*[*symmetric*] σ[*symmetric*]
proof (*induction l arbitrary*: $l'\ \sigma$)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a l* l'') **note** *IH*=*this*(*1*) **and** *le* = *this*(*2*) **and** *H* = *this*(*3*)
show *?case*
using *le H*[*of* ‹*Suc* -›] *H*[*of 0*] *IH*[*of* ‹*tl* l''› ‹-›]
by (*cases* l'')
(*auto intro*: *bind-cong-nres*)
qed
qed

lemma *nfoldli-nfoldli-list-nth*:
‹$nfoldli\ xs\ c\ P\ a = nfoldli\ [0..<length\ xs]\ c\ (\lambda i.\ P\ (xs\ !\ i))\ a$›
proof (*induction xs arbitrary*: *a*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x xs*) **note** *IH* = *this*(*1*)
have *1*: ‹$[0..<length\ (x\ \#\ xs)] = 0\ \#\ [1..<length\ (x\#xs)]$›
by (*subst upt-rec*) *simp*
have *2*: ‹$[1..<length\ (x\#xs)] = map\ Suc\ [0..<length\ xs]$›
by (*induction xs*) *auto*
have *AB*: ‹$nfoldli\ [0..<length\ (x\ \#\ xs)]\ c\ (\lambda i.\ P\ ((x\ \#\ xs)\ !\ i))\ a =$
$nfoldli\ (0\ \#\ [1..<length\ (x\#xs)])\ c\ (\lambda i.\ P\ ((x\ \#\ xs)\ !\ i))\ a$›
(**is** ‹*?A* = *?B*›)
unfolding *1* **..**

{
  assume [*simp*]: ‹*c a*›
  have ‹*nfoldli* (*0* # [*1*..<*length* (*x*#*xs*)]) *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) *a* =
    do {
      σ ← (*P x a*);
      nfoldli [*1*..<*length* (*x*#*xs*)] *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) σ
     }›
    by *simp*
  moreover have ‹*nfoldli* [*1*..<*length* (*x*#*xs*)] *c* (λ*i*. *P* ((*x* # *xs*) ! *i*)) σ =
    nfoldli [*0*..<*length xs*] *c* (λ*i*. *P* (*xs* ! *i*)) σ› **for** σ
    unfolding *2*
    by (*rule nfoldli-cong2*) *auto*
  ultimately have ‹*?A* = *do* {
      σ ← (*P x a*);
      nfoldli [*0*..<*length xs*] *c* (λ*i*. *P* (*xs* ! *i*)) σ
     }›
    using *AB*
    by (*auto intro*: *bind-cong-nres*)
}
moreover {
  assume [*simp*]: ‹¬*c a*›
  have ‹*?B* = *RETURN a*›
    by *simp*
}
ultimately show *?case* **by** (*auto simp*: *IH intro*: *bind-cong-nres*)
qed

definition *list-mset-rel* ≡ *br mset* (λ-. *True*)

lemma
  Nil-list-mset-rel-iff:
    ‹([], *aaa*) ∈ *list-mset-rel* ⟷ *aaa* = {#}› **and**
  empty-list-mset-rel-iff:
    ‹(*a*, {#}) ∈ *list-mset-rel* ⟷ *a* = []›
  by (*auto simp*: *list-mset-rel-def br-def*)

definition *list-rel-mset-rel* **where** *list-rel-mset-rel-internal*:
‹*list-rel-mset-rel* ≡ λ*R*. ⟨*R*⟩*list-rel O list-mset-rel*›

lemma *list-rel-mset-rel-def*[*refine-rel-defs*]:
  ‹⟨*R*⟩*list-rel-mset-rel* = ⟨*R*⟩*list-rel O list-mset-rel*›
  unfolding *relAPP-def list-rel-mset-rel-internal* **..**

lemma *list-rel-mset-rel-imp-same-length*: ‹(*a*, *b*) ∈ ⟨*R*⟩*list-rel-mset-rel* ⟹ *length a* = *size b*›
  by (*auto simp*: *list-rel-mset-rel-def list-mset-rel-def br-def*
      dest: *list-rel-imp-same-length*)

lemma *while-upt-while-direct1*:
  $b \geq a \Longrightarrow$
  do {
    (-,σ) ← $WHILE_T$ (*FOREACH-cond c*) (λ*x*. *do* {*ASSERT* (*FOREACH-cond c x*); *FOREACH-body f x*}) ([*a*..<*b*],σ);
    RETURN σ
  } ≤ *do* {
   (-,σ) ← $WHILE_T$ (λ(*i*, *x*). *i* < *b* ∧ *c x*) (λ(*i*, *x*). *do* {*ASSERT* (*i* < *b*); σ′←*f i x*; *RETURN* (*i*+*1*,σ′)

}) (a,σ);
    RETURN σ
  }
  apply (*rewrite at* ‹- ≤ ¤› *Down-id-eq*[*symmetric*])
  apply (*refine-vcg WHILET-refine*[**where** R = ‹{((l, x'), (i::*nat*, x::'*a*)). x= x' ∧ $i \leq b$ ∧ $i \geq a$ ∧ l = *drop* ($i-a$) [a..<b]}›])
  subgoal by *auto*
  subgoal by (*auto simp*: *FOREACH-cond-def*)
  subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
  subgoal by *auto*
  done

lemma *while-upt-while-direct2*:
  $b \geq a \Longrightarrow$
  do {
    (-,σ) ← $WHILE_T$ (*FOREACH-cond* c) (λx. *do* {*ASSERT* (*FOREACH-cond* c x); *FOREACH-body* f x}) ([a..<b],σ);
    RETURN σ
  } ≥ *do* {
   (-,σ) ← $WHILE_T$ (λ(i, x). $i < b$ ∧ c x) (λ(i, x). *do* {*ASSERT* ($i < b$); σ'←f i x; *RETURN* (i+*1*,σ') }) (a,σ);
    RETURN σ
  }
  apply (*rewrite at* ‹- ≤ ¤› *Down-id-eq*[*symmetric*])
  apply (*refine-vcg WHILET-refine*[**where** R = ‹{((i::*nat*, x::'*a*), (l, x')). x= x' ∧ $i \leq b$ ∧ $i \geq a$ ∧ l = *drop* ($i-a$) [a..<b]}›])
  subgoal by *auto*
  subgoal by (*auto simp*: *FOREACH-cond-def*)
  subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
  subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
  subgoal by *auto*
  done

lemma *while-upt-while-direct*:
  $b \geq a \Longrightarrow$
  do {
    (-,σ) ← $WHILE_T$ (*FOREACH-cond* c) (λx. *do* {*ASSERT* (*FOREACH-cond* c x); *FOREACH-body* f x}) ([a..<b],σ);
    RETURN σ
  } = *do* {
   (-,σ) ← $WHILE_T$ (λ(i, x). $i < b$ ∧ c x) (λ(i, x). *do* {*ASSERT* ($i < b$); σ'←f i x; *RETURN* (i+*1*,σ') }) (a,σ);
    RETURN σ
  }
   using *while-upt-while-direct1*[*of a b*] *while-upt-while-direct2*[*of a b*] **unfolding** *order-class.eq-iff* **by** *fast*

lemma *while-nfoldli*:
  do {
    (-,σ) ← $WHILE_T$ (*FOREACH-cond* c) (λx. *do* {*ASSERT* (*FOREACH-cond* c x); *FOREACH-body* f x}) (l,σ);
    RETURN σ
  } ≤ *nfoldli* l c f σ
  apply (*induct* l *arbitrary*: σ)
  apply (*subst WHILET-unfold*)

apply (*simp add*: *FOREACH-cond-def*)

apply (*subst WHILET-unfold*)
apply (*auto*
simp: *FOREACH-cond-def FOREACH-body-def*
intro: *bind-mono Refine-Basic.bind-mono*(*1*))
done
lemma *nfoldli-while*: *nfoldli l c f σ*
$\leq$
($WHILE_T{}^I$
(*FOREACH-cond c*) (λ*x*. *do* {*ASSERT* (*FOREACH-cond c x*); *FOREACH-body f x*}) (*l*, *σ*)
$\gg=$
(λ(-, *σ*). *RETURN σ*))
proof (*induct l arbitrary*: *σ*)
case *Nil* **thus** *?case* **by** (*subst WHILEIT-unfold*) (*auto simp*: *FOREACH-cond-def*)
next
case (*Cons x ls*)
show *?case*
proof (*cases c σ*)
case *False* **thus** *?thesis*
apply (*subst WHILEIT-unfold*)
unfolding *FOREACH-cond-def*
by *simp*
next
case [*simp*]: *True*
from *Cons* **show** *?thesis*
apply (*subst WHILEIT-unfold*)
unfolding *FOREACH-cond-def FOREACH-body-def*
apply *clarsimp*
apply (*rule Refine-Basic.bind-mono*)
apply *simp-all*
done
qed
qed

lemma *while-eq-nfoldli*: *do* {
(-,*σ*) ← $WHILE_T$ (*FOREACH-cond c*) (λ*x*. *do* {*ASSERT* (*FOREACH-cond c x*); *FOREACH-body f x*}) (*l*,*σ*);
RETURN σ
} = *nfoldli l c f σ*
apply (*rule antisym*)
apply (*rule while-nfoldli*)
apply (*rule order-trans*[*OF nfoldli-while*[**where** *I*=λ-. *True*]])
apply (*simp add*: *WHILET-def*)
done

end
theory *WB-More-Refinement-List*
imports *Weidenbach-Book-Base.WB-List-More Automatic-Refinement.Automatic-Refinement*
HOL−Word.More-Word — provides some additional lemmas like *?n* < *length ?xs* ⟹ *rev ?xs* ! *?n* = *?xs* ! (*length ?xs* − *1* − *?n*)
Refine-Monadic.Refine-Basic
begin

0.1 More theorems about list

This should theorem and functions that defined in the Refinement Framework, but not in *HOL.List*. There might be moved somewhere eventually in the AFP or so.

0.1.1 Swap two elements of a list, by index

definition *swap* **where** *swap l i j* $\equiv$ *l*[*i* := *l*!*j*, *j*:=*l*!*i*]

lemma *swap-nth*[*simp*]: ⟦*i* < *length l*; *j*<*length l*; *k*<*length l*⟧ $\Longrightarrow$
  swap l i j!*k* = (
    if k=*i then l*!*j*
    else if k=*j then l*!*i*
    else l!*k*
)
  unfolding *swap-def*
  by *auto*

lemma *swap-set*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ $\Longrightarrow$ *set* (*swap l i j*) = *set l*
  unfolding *swap-def*
  by *auto*

lemma *swap-multiset*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ $\Longrightarrow$ *mset* (*swap l i j*) = *mset l*
  unfolding *swap-def*
  by (*auto simp*: *mset-swap*)

lemma *swap-length*[*simp*]: *length* (*swap l i j*) = *length l*
  unfolding *swap-def*
  by *auto*

lemma *swap-same*[*simp*]: *swap l i i* = *l*
  unfolding *swap-def* **by** *auto*

lemma *distinct-swap*[*simp*]:
  ⟦*i*<*length l*; *j*<*length l*⟧ $\Longrightarrow$ *distinct* (*swap l i j*) = *distinct l*
  unfolding *swap-def*
  by *auto*

lemma *map-swap*: ⟦*i*<*length l*; *j*<*length l*⟧
  $\Longrightarrow$ *map f* (*swap l i j*) = *swap* (*map f l*) *i j*
  unfolding *swap-def*
  by (*auto simp add*: *map-update*)

lemma *swap-nth-irrelevant*:
  ‹*k* $\neq$ *i* $\Longrightarrow$ *k* $\neq$ *j* $\Longrightarrow$ *swap xs i j* ! *k* = *xs* ! *k*›
  by (*auto simp*: *swap-def*)

lemma *swap-nth-relevant*:
  ‹*i* < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ *swap xs i j* ! *i* = *xs* ! *j*›
  by (*cases* ‹*i* = *j*›) (*auto simp*: *swap-def*)

lemma *swap-nth-relevant2*:
  ‹*i* < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ *swap xs j i* ! *i* = *xs* ! *j*›
  by (*auto simp*: *swap-def*)

lemma *swap-nth-if*:
‹$i < length\ xs \Longrightarrow j < length\ xs \Longrightarrow swap\ xs\ i\ j\ !\ k =$
$(if\ k = i\ then\ xs\ !\ j\ else\ if\ k = j\ then\ xs\ !\ i\ else\ xs\ !\ k)$›
by (*auto simp*: *swap-def*)

lemma *drop-swap-irrelevant*:
‹$k > i \Longrightarrow k > j \Longrightarrow drop\ k\ (swap\ outl'\ j\ i) = drop\ k\ outl'$›
by (*subst list-eq-iff-nth-eq*) *auto*

lemma *take-swap-relevant*:
‹$k > i \Longrightarrow k > j \Longrightarrow take\ k\ (swap\ outl'\ j\ i) = swap\ (take\ k\ outl')\ i\ j$›
by (*subst list-eq-iff-nth-eq*) (*auto simp*: *swap-def*)

lemma *tl-swap-relevant*:
‹$i > 0 \Longrightarrow j > 0 \Longrightarrow tl\ (swap\ outl'\ j\ i) = swap\ (tl\ outl')\ (i - 1)\ (j - 1)$›
by (*subst list-eq-iff-nth-eq*)
(*cases* ‹$outl' = []$›; *cases* *i*; *cases* *j*; *auto simp*: *swap-def tl-update-swap nth-tl*)

lemma *swap-only-first-relevant*:
‹$b \geq i \Longrightarrow a < length\ xs \Longrightarrow take\ i\ (swap\ xs\ a\ b) = take\ i\ (xs[a := xs\ !\ b])$›
by (*auto simp*: *swap-def*)

TODO this should go to a different place from the previous lemmas, since it concerns *Misc.slice*, which is not part of *HOL.List* but only part of the Refinement Framework.

lemma *slice-nth*:
‹$[\![from \leq length\ xs;\ i < to - from]\!] \Longrightarrow Misc.slice\ from\ to\ xs\ !\ i = xs\ !\ (from + i)$›
unfolding *slice-def Misc.slice-def*
apply (*subst nth-take*, *assumption*)
apply (*subst nth-drop*, *assumption*)
..

lemma *slice-irrelevant*[*simp*]:
‹$i < from \Longrightarrow Misc.slice\ from\ to\ (xs[i := C]) = Misc.slice\ from\ to\ xs$›
‹$i \geq to \Longrightarrow Misc.slice\ from\ to\ (xs[i := C]) = Misc.slice\ from\ to\ xs$›
‹$i \geq to \lor i < from \Longrightarrow Misc.slice\ from\ to\ (xs[i := C]) = Misc.slice\ from\ to\ xs$›
unfolding *Misc.slice-def* **apply** *auto*
by (*metis drop-take take-update-cancel*)+

lemma *slice-update-swap*[*simp*]:
‹$i < to \Longrightarrow i \geq from \Longrightarrow i < length\ xs \Longrightarrow$
$Misc.slice\ from\ to\ (xs[i := C]) = (Misc.slice\ from\ to\ xs)[(i - from) := C]$›
unfolding *Misc.slice-def* **by** (*auto simp*: *drop-update-swap*)

lemma *drop-slice*[*simp*]:
‹$drop\ n\ (Misc.slice\ from\ to\ xs) = Misc.slice\ (from + n)\ to\ xs$› **for** *from n to xs*
by (*auto simp*: *Misc.slice-def drop-take ac-simps*)

lemma *take-slice*[*simp*]:
‹$take\ n\ (Misc.slice\ from\ to\ xs) = Misc.slice\ from\ (min\ to\ (from + n))\ xs$› **for** *from n to xs*
using *antisym-conv* **by** (*fastforce simp*: *Misc.slice-def drop-take ac-simps min-def*)

lemma *slice-append*[*simp*]:
‹$to \leq length\ xs \Longrightarrow Misc.slice\ from\ to\ (xs\ @\ ys) = Misc.slice\ from\ to\ xs$›
by (*auto simp*: *Misc.slice-def*)

lemma *slice-prepend*[*simp*]:
‹*from* $\geq$ *length xs* $\Longrightarrow$
Misc.slice from to (*xs* @ *ys*) = *Misc.slice* (*from* − *length xs*) (*to* − *length xs*) *ys*›
by (*auto simp*: *Misc.slice-def*)

lemma *slice-len-min-If*:
‹*length* (*Misc.slice from to xs*) =
(*if from* < *length xs then min* (*length xs* − *from*) (*to* − *from*) *else 0*)›
unfolding *min-def* **by** (*auto simp*: *Misc.slice-def*)

lemma *slice-start0*: ‹*Misc.slice 0 to xs* = *take to xs*›
unfolding *Misc.slice-def*
by *auto*

lemma *slice-end-length*: ‹*n* $\geq$ *length xs* $\Longrightarrow$ *Misc.slice to n xs* = *drop to xs*›
unfolding *Misc.slice-def*
by *auto*

lemma *slice-swap*[*simp*]:
‹*l* $\geq$ *from* $\Longrightarrow$ *l* < *to* $\Longrightarrow$ *k* $\geq$ *from* $\Longrightarrow$ *k* < *to* $\Longrightarrow$ *from* < *length arena* $\Longrightarrow$
Misc.slice from to (*swap arena l k*) = *swap* (*Misc.slice from to arena*) (*k* − *from*) (*l* − *from*)›
by (*cases* ‹*k* = *l*›) (*auto simp*: *Misc.slice-def swap-def drop-update-swap list-update-swap*)

lemma *drop-swap-relevant*[*simp*]:
‹*i* $\geq$ *k* $\Longrightarrow$ *j* $\geq$ *k* $\Longrightarrow$ *j* < *length outl'* $\Longrightarrow$*drop k* (*swap outl' j i*) = *swap* (*drop k outl'*) (*j* − *k*) (*i* − *k*)›
by (*cases* ‹*j* = *i*›)
(*auto simp*: *Misc.slice-def swap-def drop-update-swap list-update-swap*)

lemma *swap-swap*: ‹*k* < *length xs* $\Longrightarrow$ *l* < *length xs* $\Longrightarrow$ *swap xs k l* = *swap xs l k*›
by (*cases* ‹*k* = *l*›)
(*auto simp*: *Misc.slice-def swap-def drop-update-swap list-update-swap*)

lemma *list-rel-append-single-iff*:
‹(*xs* @ [*x*], *ys* @ [*y*]) ∈ ⟨*R*⟩*list-rel* ⟷
(*xs*, *ys*) ∈ ⟨*R*⟩*list-rel* ∧ (*x*, *y*) ∈ *R*›
using *list-all2-lengthD*[*of* ‹(λ*x x'*. (*x*, *x'*) ∈ *R*)› ‹*xs* @ [*x*]› ‹*ys* @ [*y*]›]
using *list-all2-lengthD*[*of* ‹(λ*x x'*. (*x*, *x'*) ∈ *R*)› ‹*xs*› ‹*ys*›]
by (*auto simp*: *list-rel-def list-all2-append*)

lemma *nth-in-sliceI*:
‹*i* $\geq$ *j* $\Longrightarrow$ *i* < *k* $\Longrightarrow$ *k* $\leq$ *length xs* $\Longrightarrow$ *xs* ! *i* ∈ *set* (*Misc.slice j k xs*)›
by (*auto simp*: *Misc.slice-def in-set-take-conv-nth*
intro!: *bex-lessI*[*of* - ‹*i* − *j*›])

lemma *slice-Suc*:
‹*Misc.slice* (*Suc j*) *k xs* = *tl* (*Misc.slice j k xs*)›
apply (*auto simp*: *Misc.slice-def in-set-take-conv-nth drop-Suc take-tl tl-drop*
drop-take)
by (*metis drop-Suc drop-take tl-drop*)

lemma *slice-0*:
‹*Misc.slice 0 b xs* = *take b xs*›
by (*auto simp*: *Misc.slice-def*)

lemma *slice-end*:
  ‹$c = length\ xs \Longrightarrow Misc.slice\ b\ c\ xs = drop\ b\ xs$›
  by (*auto simp*: *Misc.slice-def*)

lemma *slice-append-nth*:
  ‹$a \le b \Longrightarrow Suc\ b \le length\ xs \Longrightarrow Misc.slice\ a\ (Suc\ b)\ xs = Misc.slice\ a\ b\ xs\ @\ [xs\ !\ b]$›
  by (*auto simp*: *Misc.slice-def take-Suc-conv-app-nth*
    Suc-diff-le)

lemma *take-set*: $set\ (take\ n\ l) = \{\ l!i \mid i.\ i{<}n \wedge i{<}length\ l\ \}$
  apply (*auto simp add*: *set-conv-nth*)
  apply (*rule-tac* $x{=}i$ **in** *exI*)
  apply *auto*
  done

fun *delete-index-and-swap* **where**
  ‹$delete\text{-}index\text{-}and\text{-}swap\ l\ i = butlast(l[i := last\ l])$›

lemma (**in** −) *delete-index-and-swap-alt-def*:
  ‹$delete\text{-}index\text{-}and\text{-}swap\ S\ i =$
    $(let\ x = last\ S\ in\ butlast\ (S[i := x]))$›
  by *auto*

lemma *swap-param*[*param*]: $[\![\ i{<}length\ l;\ j{<}length\ l;\ (l',l){\in}\langle A\rangle list\text{-}rel;\ (i',i){\in}nat\text{-}rel;\ (j',j){\in}nat\text{-}rel]\!]$
  $\Longrightarrow (swap\ l'\ i'\ j',\ swap\ l\ i\ j){\in}\langle A\rangle list\text{-}rel$
  unfolding *swap-def*
  by *parametricity*

lemma *mset-tl-delete-index-and-swap*:
  assumes
    ‹$0 < i$› **and**
    ‹$i < length\ outl'$›
  shows ‹$mset\ (tl\ (delete\text{-}index\text{-}and\text{-}swap\ outl'\ i)) =$
        $remove1\text{-}mset\ (outl'\ !\ i)\ (mset\ (tl\ outl'))$›
  using *assms*
  by (*subst mset-tl*)+
    (*auto simp*: *hd-butlast hd-list-update-If mset-butlast-remove1-mset*
      mset-update last-list-update-to-last ac-simps)

definition *length-ll* :: ‹$'a\ list\ list \Rightarrow nat \Rightarrow nat$› **where**
  ‹$length\text{-}ll\ l\ i = length\ (l!i)$›

definition *delete-index-and-swap-ll* **where**
  ‹$delete\text{-}index\text{-}and\text{-}swap\text{-}ll\ xs\ i\ j =$
     $xs[i{:=}\ delete\text{-}index\text{-}and\text{-}swap\ (xs!i)\ j]$›

definition *append-ll* :: $'a\ list\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list\ list$ **where**
  ‹$append\text{-}ll\ xs\ i\ x = list\text{-}update\ xs\ i\ (xs\ !\ i\ @\ [x])$›

definition (**in** −)*length-uint32-nat* **where**
  [*simp*]: ‹$length\text{-}uint32\text{-}nat\ C = length\ C$›

definition (**in** $-$)*length-uint64-nat* **where**
[*simp*]: ‹*length-uint64-nat C* = *length C*›

definition *nth-rll* :: *'a list list* ⇒ *nat* ⇒ *nat* ⇒ *'a* **where**
‹*nth-rll l i j* = *l* ! *i* ! *j*›

definition *reorder-list* :: ‹*'b* ⇒ *'a list* ⇒ *'a list nres*› **where**
‹*reorder-list* - *removed* = *SPEC* (λ*removed'*. *mset removed'* = *mset removed*)›

end
theory *WB-More-IICF-SML*
imports *Refine-Imperative-HOL.IICF WB-More-Refinement WB-More-Refinement-List*
begin

no-notation *Sepref-Rules.fref* ($[-]_f$ - → - [*0*,*60*,*60*] *60*)
no-notation *Sepref-Rules.freft* (- $\rightarrow_f$ - [*60*,*60*] *60*)
no-notation *prod-assn* (**infixr** $\times_a$ *70*)
notation *prod-assn* (**infixr** ∗*a* *70*)

hide-const *Autoref-Fix-Rel.CONSTRAINT IICF-List-Mset.list-mset-rel*

lemma *prod-assn-id-assn-destroy*:
fixes *R* :: ‹- ⇒ - ⇒ *assn*›
shows ‹R^d $*_a$ *id-assn*^{d} = (*R* ∗*a* *id-assn*)^d›
by (*auto simp*: *hfprod-def prod-assn-def*[*abs-def*] *invalid-assn-def pure-def intro*!: *ext*)

definition *list-mset-assn* **where**
list-mset-assn A ≡ *pure* (*list-mset-rel O* ⟨*the-pure A*⟩*mset-rel*)
declare *list-mset-assn-def*[*symmetric*,*fcomp-norm-unfold*]
lemma [*safe-constraint-rules*]: *is-pure* (*list-mset-assn A*) **unfolding** *list-mset-assn-def* **by** *simp*

lemma
shows *list-mset-assn-add-mset-Nil*:
‹*list-mset-assn R* (*add-mset q Q*) [] = *false*› **and**
list-mset-assn-empty-Cons:
‹*list-mset-assn R* {#} (*x* # *xs*) = *false*›
unfolding *list-mset-assn-def list-mset-rel-def mset-rel-def pure-def p2rel-def*
rel2p-def rel-mset-def br-def
by (*sep-auto simp*: *Collect-eq-comp*)+

lemma *list-mset-assn-add-mset-cons-in*:
assumes
assn: ‹*A* ⊨ *list-mset-assn R N* (*ab* # *list*)›
shows ‹∃ *ab'*. (*ab*, *ab'*) ∈ *the-pure R* ∧ *ab'* ∈# *N* ∧ *A* ⊨ *list-mset-assn R* (*remove1-mset ab' N*) (*list*)›
proof $-$
have *H*: ‹(∀ *x x'*. (*x'* = *x*) = ((*x'*, *x*) ∈ *P'*)) ⟷ *P'* = *Id*› **for** *P'*
by (*auto simp*: *the-pure-def*)
have [*simp*]: ‹*the-pure* (λ*a c*. ↑ (*c* = *a*)) = *Id*›
by (*auto simp*: *the-pure-def H*)
have [*iff*]: ‹(*ab* # *list*, *y*) ∈ *list-mset-rel* ⟷ *y* = *add-mset ab* (*mset list*)› **for** *y ab list*
by (*auto simp*: *list-mset-rel-def br-def*)
obtain *N' xs* **where**
N-N': ‹*N* = *mset N'*› **and**
‹*mset xs* = *add-mset ab* (*mset list*)› **and**

⟨*list-all2* (*rel2p* (*the-pure R*)) *xs N′*⟩
using *assn* **by** (*cases A*) (*auto simp*: *list-mset-assn-def mset-rel-def p2rel-def rel-mset-def rel2p-def*)
then obtain N'' **where**
⟨*list-all2* (*rel2p* (*the-pure R*)) (*ab* # *list*) N''⟩ **and**
⟨*mset* N'' = *mset* N'⟩
using *list-all2-reorder-left-invariance*[*of* ⟨*rel2p* (*the-pure R*)⟩ *xs N′* ⟨*ab* # *list*⟩, *unfolded eq-commute*[*of* ⟨*mset* (*ab* # *list*)⟩]] **by** *auto*
then obtain *n* N''' **where**
n: ⟨*add-mset n* (*mset* N''') = *mset* N''⟩ **and**
⟨(*ab*, *n*) ∈ *the-pure R*⟩ **and**
⟨*list-all2* (*rel2p* (*the-pure R*)) *list* N'''⟩
by (*auto simp*: *list-all2-Cons1 rel2p-def*)
moreover have ⟨*n* ∈ *set* N''⟩
using *n* **unfolding** *mset.simps*[*symmetric*] *eq-commute*[*of* ⟨*add-mset - -*⟩] **apply** −
by (*drule mset-eq-setD*) *auto*
ultimately have ⟨(*ab*, *n*) ∈ *the-pure R*⟩ **and**
⟨*n* ∈ *set* N''⟩ **and**
⟨*mset list* = *mset list*⟩ **and**
⟨*mset* N''' = *remove1-mset n* (*mset* N'')⟩ **and**
⟨*list-all2* (*rel2p* (*the-pure R*)) *list* N'''⟩
by (*auto dest*: *mset-eq-setD simp*: *eq-commute*[*of* ⟨*add-mset - -*⟩])
show *?thesis*
unfolding *list-mset-assn-def mset-rel-def p2rel-def rel-mset-def list.rel-eq list-mset-rel-def br-def N-N′*
using *assn* ⟨(*ab*, *n*) ∈ *the-pure R*⟩ ⟨*n* ∈ *set* N''⟩ ⟨*mset* N'' = *mset* N'⟩
⟨*list-all2* (*rel2p* (*the-pure R*)) *list* N'''⟩
⟨*mset* N'' = *mset* N'⟩ ⟨*mset* N''' = *remove1-mset n* (*mset* N'')⟩
by (*cases A*) (*auto simp*: *list-mset-assn-def mset-rel-def p2rel-def rel-mset-def add-mset-eq-add-mset list.rel-eq*
intro!: *exI*[*of - n*]
dest: *mset-eq-setD*)
qed

lemma *list-mset-assn-empty-nil*: ⟨*list-mset-assn R* {#} [] = *emp*⟩
by (*auto simp*: *list-mset-assn-def list-mset-rel-def mset-rel-def br-def p2rel-def rel2p-def Collect-eq-comp rel-mset-def pure-def*)

lemma *is-Nil-is-empty*[*sepref-fr-rules*]:
⟨(*return o is-Nil*, *RETURN o Multiset.is-empty*) ∈ (*list-mset-assn R*)^k $\rightarrow_a$ *bool-assn*⟩
apply *sepref-to-hoare*
apply (*rename-tac x xi*)
apply (*case-tac x*)
by (*sep-auto simp*: *Multiset.is-empty-def list-mset-assn-empty-Cons list-mset-assn-add-mset-Nil split*: *list.splits*)+

lemma *list-all2-remove*:
assumes
uniq: ⟨*IS-RIGHT-UNIQUE* (*p2rel R*)⟩ ⟨*IS-LEFT-UNIQUE* (*p2rel R*)⟩ **and**
Ra: ⟨*R a aa*⟩ **and**
all: ⟨*list-all2 R xs ys*⟩
shows
⟨∃ *xs′*. *mset xs′* = *remove1-mset a* (*mset xs*) ∧

(∃ ys'. *mset* ys' = *remove1-mset aa* (*mset ys*) ∧ *list-all2 R* xs' ys')›
  using *all*
proof (*induction xs ys rule*: *list-all2-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons x y xs ys*) **note** *IH* = *this*(*3*) **and** *p* = *this*(*1*, *2*)

  have *ax*: ‹{#*a*, *x*#} = {#*x*, *a*#}›
    by *auto*
  have *rem1*: ‹*remove1-mset a* (*remove1-mset x M*) = *remove1-mset x* (*remove1-mset a M*)› **for** *M*
    by (*auto simp*: *ax*)
  have *H*: ‹*x* = *a* ⟷ *y* = *aa*›
    using *uniq Ra p* **unfolding** *single-valued-def IS-LEFT-UNIQUE-def p2rel-def* **by** *blast*

  obtain xs' ys' **where**
   ‹*mset* xs' = *remove1-mset a* (*mset xs*)› **and**
   ‹*mset* ys' = *remove1-mset aa* (*mset ys*)› **and**
   ‹*list-all2 R* xs' ys'›
   using *IH p* **by** *auto*
  then show *?case*
   apply (*cases* ‹*x* ≠ *a*›)
   subgoal
     using *p*
     by (*auto intro*!: *exI*[*of* - ‹*x*#xs'›] *exI*[*of* - ‹*y*#ys'›]
         simp: *diff-add-mset-remove1 rem1 add-mset-remove-trivial-If in-remove1-mset-neq H*
         simp del: *diff-diff-add-mset*)
   subgoal
     using *p*
     by (*fastforce simp*: *diff-add-mset-remove1 rem1 add-mset-remove-trivial-If in-remove1-mset-neq*
         remove-1-mset-id-iff-notin H
         simp del: *diff-diff-add-mset*)
   done
qed

lemma *remove1-remove1-mset*:
  assumes *uniq*: ‹*IS-RIGHT-UNIQUE R*› ‹*IS-LEFT-UNIQUE R*›
  shows ‹(*uncurry* (*RETURN oo remove1*), *uncurry* (*RETURN oo remove1-mset*)) ∈
    $R \times_r$ (*list-mset-rel O* ⟨*R*⟩ *mset-rel*) $\rightarrow_f$
    ⟨*list-mset-rel O* ⟨*R*⟩ *mset-rel*⟩ *nres-rel*›
  using *list-all2-remove*[*of* ‹*rel2p R*›] *assms*
  by (*intro frefI nres-relI*) (*fastforce simp*: *list-mset-rel-def br-def mset-rel-def p2rel-def*
      rel2p-def[*abs-def*] *rel-mset-def Collect-eq-comp*)

lemma
  Nil-list-mset-rel-iff:
   ‹([], *aaa*) ∈ *list-mset-rel* ⟷ *aaa* = {#}› **and**
  empty-list-mset-rel-iff:
   ‹(*a*, {#}) ∈ *list-mset-rel* ⟷ *a* = []›
  by (*auto simp*: *list-mset-rel-def br-def*)

lemma *snd-hnr-pure*:
  ‹*CONSTRAINT is-pure B* ⟹ (*return* ∘ *snd*, *RETURN* ∘ *snd*) ∈ $A^d *_a B^k \rightarrow_a B$›
  apply *sepref-to-hoare*
  apply *sep-auto*

by (*metis SLN-def SLN-left assn-times-comm ent-pure-pre-iff-sng ent-refl ent-star-mono ent-true is-pure-assn-def is-pure-iff-pure-assn*)

This theorem is useful to debug situation where sepref is not able to synthesize a program (with the "[[unify_trace_failure]]" to trace what fails in rule rule and the *to-hnr* to ensure the theorem has the correct form).

lemma *Pair-hnr*: ‹(*uncurry* (*return oo* ($\lambda a\ b.$ *Pair a b*)), *uncurry* (*RETURN oo* ($\lambda a\ b.$ *Pair a b*))) $\in$ $A^d *_a B^d \rightarrow_a$ *prod-assn A B*›
by *sepref-to-hoare sep-auto*

This version works only for *pure* refinement relations:

lemma *the-hnr-keep*:
‹*CONSTRAINT is-pure A* $\Longrightarrow$ (*return o the*, *RETURN o the*) $\in [\lambda D.\ D \neq None]_a$ (*option-assn A*)^k $\rightarrow$ *A*›
using *pure-option*[*of A*]
by *sepref-to-hoare*
(*sep-auto simp*: *option-assn-alt-def is-pure-def split*: *option.splits*)

definition *list-rel-mset-rel* **where** *list-rel-mset-rel-internal*:
‹*list-rel-mset-rel* $\equiv \lambda R.\ \langle R\rangle$*list-rel O list-mset-rel*›

lemma *list-rel-mset-rel-def*[*refine-rel-defs*]:
‹$\langle R\rangle$*list-rel-mset-rel* = $\langle R\rangle$*list-rel O list-mset-rel*›
unfolding *relAPP-def list-rel-mset-rel-internal* **..**

lemma *list-mset-assn-pure-conv*:
‹*list-mset-assn* (*pure R*) = *pure* ($\langle R\rangle$*list-rel-mset-rel*)›
apply (*intro ext*)
using *list-all2-reorder-left-invariance*
by (*fastforce*
simp: *list-rel-mset-rel-def list-mset-assn-def*
mset-rel-def rel2p-def[*abs-def*] *rel-mset-def p2rel-def*
list-mset-rel-def[*abs-def*] *Collect-eq-comp br-def*
list-rel-def Collect-eq-comp-right
intro!: *arg-cong*[*of - -* ‹$\lambda b.$ *pure b - -*›])

lemma *list-assn-list-mset-rel-eq-list-mset-assn*:
assumes *p*: ‹*is-pure R*›
shows ‹*hr-comp* (*list-assn R*) *list-mset-rel* = *list-mset-assn R*›
proof −
define R' **where** ‹R' = *the-pure R*›
then have *R*: ‹*R* = *pure* R'›
using *p* **by** *auto*
show *?thesis*
apply (*auto simp*: *list-mset-assn-def*
list-assn-pure-conv
relcomp.simps hr-comp-pure mset-rel-def br-def
p2rel-def rel2p-def[*abs-def*] *rel-mset-def R list-mset-rel-def list-rel-def*)
using *list-all2-reorder-left-invariance* **by** *fastforce*
qed

lemma *id-ref*: ‹(*return o id*, *RETURN o id*) $\in R^d \rightarrow_a R$›
by *sepref-to-hoare sep-auto*

This functions deletes all elements of a resizable array, without resizing it.

definition *emptied-arl* :: ⟨$'a$ *array-list* ⇒ $'a$ *array-list*⟩ **where**
⟨*emptied-arl* = (λ(*a*, *n*). (*a*, *0*))⟩

lemma *emptied-arl-refine*[*sepref-fr-rules*]:
⟨(*return o emptied-arl*, *RETURN o emptied-list*) ∈ (*arl-assn R*)^d $\rightarrow_a$ *arl-assn R*⟩
unfolding *emptied-arl-def emptied-list-def*
by *sepref-to-hoare* (*sep-auto simp*: *arl-assn-def hr-comp-def is-array-list-def*)

lemma *bool-assn-alt-def*: ⟨*bool-assn a b* = ↑ (*a* = *b*)⟩
unfolding *pure-def* **by** *auto*

lemma *nempty-list-mset-rel-iff*: ⟨*M* ≠ {#} ⟹
(*xs*, *M*) ∈ *list-mset-rel* ⟷ (*xs* ≠ [] ∧ *hd xs* ∈# *M* ∧
(*tl xs*, *remove1-mset* (*hd xs*) *M*) ∈ *list-mset-rel*)⟩
by (*cases xs*) (*auto simp*: *list-mset-rel-def br-def dest*!: *multi-member-split*)

abbreviation *ghost-assn* **where**
⟨*ghost-assn* ≡ *hr-comp unit-assn virtual-copy-rel*⟩

lemma [*sepref-fr-rules*]:
⟨(*return o* (λ-. ()), *RETURN o virtual-copy*) ∈ R^k $\rightarrow_a$ *ghost-assn*⟩
by *sepref-to-hoare* (*sep-auto simp*: *virtual-copy-rel-def*)

lemma *id-mset-list-assn-list-mset-assn*:
assumes ⟨*CONSTRAINT is-pure R*⟩
shows ⟨(*return o id*, *RETURN o mset*) ∈ (*list-assn R*)^d $\rightarrow_a$ *list-mset-assn R*⟩
proof −
obtain *R'* **where** *R*: ⟨*R* = *pure R'*⟩
using *assms is-pure-conv* **unfolding** *CONSTRAINT-def* **by** *blast*
then have *R'*: ⟨*the-pure R* = *R'*⟩
unfolding *R* **by** *auto*
show *?thesis*
apply (*subst R*)
apply (*subst list-assn-pure-conv*)
apply *sepref-to-hoare*
by (*sep-auto simp*: *list-mset-assn-def R' pure-def list-mset-rel-def mset-rel-def*
p2rel-def rel2p-def[*abs-def*] *rel-mset-def br-def Collect-eq-comp list-rel-def*)
qed

0.1.2 Sorting

Remark that we do not *prove* that the sorting in correct, since we do not care about the correctness, only the fact that it is reordered. (Based on wikipedia's algorithm.) Typically *R* would be (<)

definition *insert-sort-inner* :: ⟨($'b$ ⇒ $'b$ ⇒ *bool*) ⇒ ($'a$ *list* ⇒ *nat* ⇒ $'b$) ⇒ $'a$ *list* ⇒ *nat* ⇒ $'a$ *list nres*⟩ **where**
⟨*insert-sort-inner R f xs i* = *do* {
(*j*, *ys*) ← $WHILE_T{}^{\lambda(j,\ ys).\ j \geq 0\ \wedge\ mset\ xs\ =\ mset\ ys\ \wedge\ j\ <\ length\ ys}$
(λ(*j*, *ys*). *j* > *0* ∧ *R* (*f ys j*) (*f ys* (*j* − *1*)))
(λ(*j*, *ys*). *do* {
ASSERT(*j* < *length ys*);
ASSERT(*j* > *0*);
ASSERT(*j*−*1* < *length ys*);
let xs = *swap ys j* (*j* − *1*);

```
          RETURN (j−1, xs)
        }
      )
      (i, xs);
    RETURN ys
  }›
```

```
lemma ‹RETURN [Suc 0, 2, 0] = insert-sort-inner (<) (λremove n. remove ! n) [2::nat, 1, 0] 1›
  by (simp add: WHILEIT-unfold insert-sort-inner-def swap-def)

definition insert-sort :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a list ⇒ nat ⇒ 'b) ⇒ 'a list ⇒ 'a list nres› where
  ‹insert-sort R f xs = do {
    (i, ys) ← WHILE_T^{λ(i, ys). (ys = [] ∨ i ≤ length ys) ∧ mset xs = mset ys}
      (λ(i, ys). i < length ys)
      (λ(i, ys). do {
          ASSERT(i < length ys);
          ys ← insert-sort-inner R f ys i;
          RETURN (i+1, ys)
        })
      (1, xs);
    RETURN ys
  }›

lemma insert-sort-inner:
   ‹(uncurry (insert-sort-inner R f), uncurry (λm m'. reorder-list m' m)) ∈
      [λ(xs, i). i < length xs]_f ⟨Id:: ('a × 'a) set⟩list-rel ×_r nat-rel → ⟨Id⟩ nres-rel›
  unfolding insert-sort-inner-def uncurry-def reorder-list-def
  apply (intro frefI nres-relI)
  apply clarify
  apply (refine-vcg WHILEIT-rule[where R = ‹measure (λ(i, -). i)›])
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by (auto dest: mset-eq-length)
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  done

lemma insert-sort-reorder-list:
  ‹(insert-sort R f, reorder-list vm) ∈ ⟨Id⟩list-rel →_f ⟨Id⟩ nres-rel›
proof −
  have H: ‹ba < length aa ⟹ insert-sort-inner R f aa ba ≤ SPEC (λm'. mset m' = mset aa)›
    for ba aa
    using insert-sort-inner[unfolded fref-def nres-rel-def reorder-list-def, simplified]
    by fast
  show ?thesis
    unfolding insert-sort-def reorder-list-def
    apply (intro frefI nres-relI)
```

apply (*refine-vcg WHILEIT-rule*[**where** R = ‹*measure* (λ(*i*, *ys*). *length ys* − *i*)›] *H*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto dest*: *mset-eq-length*)
subgoal by *auto*
subgoal by (*auto dest*!: *mset-eq-length*)
subgoal by *auto*
done
qed

definition *arl-replicate* **where**
arl-replicate init-cap x ≡ *do* {
let n = *max init-cap minimum-capacity*;
a ← *Array.new n x*;
return (*a*,*init-cap*)
}

definition ‹*op-arl-replicate* = *op-list-replicate*›
lemma *arl-fold-custom-replicate*:
‹*replicate* = *op-arl-replicate*›
unfolding *op-arl-replicate-def op-list-replicate-def* **..**

lemma *list-replicate-arl-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*CONSTRAINT is-pure R*›
shows ‹(*uncurry arl-replicate*, *uncurry* (*RETURN oo op-arl-replicate*)) ∈ *nat-assn*^{k} $*_a$ R^k $\rightarrow_a$ *arl-assn R*›
proof −
obtain R' **where**
R'[*symmetric*]: ‹R' = *the-pure R*› **and**
R-R': ‹R = *pure* R'›
using *assms* **by** *fastforce*
have [*simp*]: ‹*pure* R' *b bi* = ↑((*bi*, *b*) ∈ R')› **for** *b bi*
by (*auto simp*: *pure-def*)
have [*simp*]: ‹*min a* (*max a 16*) = *a*› **for** *a* :: *nat*
by *auto*
show *?thesis*
using *assms* **unfolding** *op-arl-replicate-def*
by *sepref-to-hoare*
(*sep-auto simp*: *arl-replicate-def arl-assn-def hr-comp-def* R' *R*-R' *list-rel-def*
is-array-list-def minimum-capacity-def
intro!: *list-all2-replicate*)
qed

lemma *option-bool-assn-direct-eq-hnr*:
‹(*uncurry* (*return oo* (=)), *uncurry* (*RETURN oo* (=))) ∈
(*option-assn bool-assn*)^k $*_a$ (*option-assn bool-assn*)^k $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *option-assn-alt-def split*:*option.splits*)

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

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

lemma *take1-hnr*[*sepref-fr-rules*]:
‹(*return o* (λ(*a*, -). (*a*, *1*::*nat*)), *RETURN o take1*) ∈ [λ*xs*. *xs* ≠ []]$_a$ (*arl-assn R*)^d → *arl-assn R*›

apply *sepref-to-hoare*
apply (*sep-auto simp*: *arl-assn-def hr-comp-def take1-def list-rel-def is-array-list-def*)
apply (*case-tac b*; *case-tac x*; *case-tac l′*)
apply (*auto*)
done

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

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

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

definition *find-in-list-between* :: ‹(′*a* ⇒ *bool*) ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ *nat option nres*› **where**
‹*find-in-list-between P a b C* = *do* {
  (*x*, -) ← $WHILE_T{}^{\lambda(found,\ i).\ i \geq a \wedge i \leq length\ C \wedge i \leq b \wedge (\forall j \in \{a..<i\}.\ \neg P\ (C!j)) \wedge \quad (\forall j.\ found = Some\ j \longrightarrow (i}$
    (λ(*found*, *i*). *found* = *None* ∧ *i* < *b*)
    (λ(-, *i*). *do* {
      ASSERT(*i* < *length C*);
      if P (*C*!*i*) *then RETURN* (*Some i*, *i*) *else RETURN* (*None*, *i*+*1*)
    })
    (*None*, *a*);
   RETURN x
}›

lemma *find-in-list-between-spec*:
assumes ‹$a \leq length\ C$› **and** ‹$b \leq length\ C$› **and** ‹$a \leq b$›
shows
‹*find-in-list-between P a b C* ≤ *SPEC*($\lambda i.$
  $(i \neq None \longrightarrow\ P\ (C\ !\ the\ i) \wedge the\ i \geq a \wedge the\ i < b) \wedge$
  $(i = None \longrightarrow (\forall j.\ j \geq a \longrightarrow j < b \longrightarrow \neg P\ (C!j))))$›
unfolding *find-in-list-between-def*
apply (*refine-vcg WHILEIT-rule*[**where** *R* = ‹*measure* (λ(*f*, *i*). *Suc* (*length C*) − (*i* + (*if f* = *None then 0 else 1*)))›])
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *less-Suc-eq*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

lemma *list-assn-map-list-assn*: ‹$list\text{-}assn\ g\ (map\ f\ x)\ xi = list\text{-}assn\ (\lambda a\ c.\ g\ (f\ a)\ c)\ x\ xi$›
apply (*induction x arbitrary*: *xi*)
subgoal by *auto*
subgoal for *a x xi*
by (*cases xi*) *auto*
done

lemma *hfref-imp2*: $(\bigwedge x\ y.\ S\ x\ y \Longrightarrow_t S'\ x\ y) \Longrightarrow [P]_a\ RR \to S \subseteq [P]_a\ RR \to S'$
apply *clarsimp*
apply (*erule hfref-cons*)
apply (*simp-all add*: *hrp-imp-def*)
done

lemma *hr-comp-mono-entails*: ‹$B \subseteq C \Longrightarrow hr\text{-}comp\ a\ B\ x\ y \Longrightarrow_A hr\text{-}comp\ a\ C\ x\ y$›
unfolding *hr-comp-def entails-def*
by *auto*

lemma *hfref-imp-mono-result*:
$B \subseteq C \Longrightarrow [P]_a\ RR \to hr\text{-}comp\ a\ B \subseteq [P]_a\ RR \to hr\text{-}comp\ a\ C$
unfolding *hfref-def hn-refine-def*
apply *clarify*
subgoal for *aa b c aaa*
apply (*rule cons-post-rule*[*of* - -
‹$\lambda r.\ snd\ RR\ aaa\ c * (\exists_A x.\ hr\text{-}comp\ a\ B\ x\ r * \uparrow (RETURN\ x \le b\ aaa)) * true$›])
apply (*solves auto*)
using *hr-comp-mono-entails*[*of B C a*]
apply (*auto intro*!: *ent-ex-preI*)
apply (*rule-tac x=xa* **in** *ent-ex-postI*)
apply (*auto intro*!: *ent-star-mono ac-simps*)
done
done

lemma *hfref-imp-mono-result2*:
$(\bigwedge x.\ P\ L\ x \Longrightarrow B\ L \subseteq C\ L) \Longrightarrow [P\ L]_a\ RR \to hr\text{-}comp\ a\ (B\ L) \subseteq [P\ L]_a\ RR \to hr\text{-}comp\ a\ (C\ L)$
unfolding *hfref-def hn-refine-def*

apply *clarify*
subgoal for *aa b c aaa*
apply (*rule cons-post-rule*[*of* - -
$\langle \lambda r.\ snd\ RR\ aaa\ c * (\exists_A x.\ hr\text{-}comp\ a\ (B\ L)\ x\ r * \uparrow (RETURN\ x \le b\ aaa)) * true \rangle$])
apply (*solves auto*)
using *hr-comp-mono-entails*[*of* $\langle B\ L\rangle$ $\langle C\ L\rangle$ *a*]
apply (*auto intro*!: *ent-ex-preI*)
apply (*rule-tac x*=*xa* **in** *ent-ex-postI*)
apply (*auto intro*!: *ent-star-mono ac-simps*)
done
done

lemma *ex-assn-up-eq2*: $\langle (\exists_A ba.\ f\ ba * \uparrow (ba = c)) = (f\ c)\rangle$
by (*simp add*: *ex-assn-def*)

lemma *ex-assn-pair-split*: $\langle (\exists_A b.\ P\ b) = (\exists_A a\ b.\ P\ (a, b))\rangle$
by (*subst ex-assn-def*, *subst* (*1*) *ex-assn-def*, *auto*)+

lemma *ex-assn-swap*: $\langle (\exists_A a\ b.\ P\ a\ b) = (\exists_A b\ a.\ P\ a\ b)\rangle$
by (*meson ent-ex-postI ent-ex-preI ent-iffI ent-refl*)

lemma *ent-ex-up-swap*: $\langle (\exists_A aa.\ \uparrow (P\ aa)) = (\uparrow(\exists aa.\ P\ aa))\rangle$
by (*smt ent-ex-postI ent-ex-preI ent-iffI ent-pure-pre-iff ent-refl mult.left-neutral*)

lemma *ex-assn-def-pure-eq-middle3*:
$\langle (\exists_A ba\ b\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb) * P\ b\ ba\ bb) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * P\ b\ (h\ b\ bb)\ bb)\rangle$
$\langle (\exists_A b\ ba\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb) * P\ b\ ba\ bb) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * P\ b\ (h\ b\ bb)\ bb)\rangle$
$\langle (\exists_A b\ bb\ ba.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb) * P\ b\ ba\ bb) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * P\ b\ (h\ b\ bb)\ bb)\rangle$
$\langle (\exists_A ba\ b\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb \wedge Q\ b\ ba\ bb)) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * \uparrow(Q\ b\ (h\ b\ bb)\ bb))\rangle$
$\langle (\exists_A b\ ba\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb \wedge Q\ b\ ba\ bb)) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * \uparrow(Q\ b\ (h\ b\ bb)\ bb))\rangle$
$\langle (\exists_A b\ bb\ ba.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb \wedge Q\ b\ ba\ bb)) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * \uparrow(Q\ b\ (h\ b\ bb)\ bb))\rangle$
by (*subst ex-assn-def*, *subst* (*3*) *ex-assn-def*, *auto*)+

lemma *ex-assn-def-pure-eq-middle2*:
$\langle (\exists_A ba\ b.\ f\ b\ ba * \uparrow (ba = h\ b) * P\ b\ ba) = (\exists_A b\ .\ f\ b\ (h\ b) * P\ b\ (h\ b))\rangle$
$\langle (\exists_A b\ ba.\ f\ b\ ba * \uparrow (ba = h\ b) * P\ b\ ba) = (\exists_A b\ .\ f\ b\ (h\ b) * P\ b\ (h\ b))\rangle$
$\langle (\exists_A b\ ba.\ f\ b\ ba * \uparrow (ba = h\ b \wedge Q\ b\ ba)) = (\exists_A b.\ f\ b\ (h\ b) * \uparrow(Q\ b\ (h\ b)))\rangle$
$\langle (\exists_A\ ba\ b.\ f\ b\ ba * \uparrow (ba = h\ b \wedge Q\ b\ ba)) = (\exists_A b.\ f\ b\ (h\ b) * \uparrow(Q\ b\ (h\ b)))\rangle$
by (*subst ex-assn-def*, *subst* (*2*) *ex-assn-def*, *auto*)+

lemma *ex-assn-skip-first2*:
$\langle (\exists_A ba\ bb.\ f\ bb * \uparrow(P\ ba\ bb)) = (\exists_A bb.\ f\ bb * \uparrow(\exists ba.\ P\ ba\ bb))\rangle$
$\langle (\exists_A bb\ ba.\ f\ bb * \uparrow(P\ ba\ bb)) = (\exists_A bb.\ f\ bb * \uparrow(\exists ba.\ P\ ba\ bb))\rangle$
apply (*subst ex-assn-swap*)
by (*subst ex-assn-def*, *subst* (*2*) *ex-assn-def*, *auto*)+

lemma *fr-refl'*: $\langle A \Longrightarrow_A B \Longrightarrow C * A \Longrightarrow_A C * B\rangle$
unfolding *assn-times-comm*[*of C*]
by (*rule Automation.fr-refl*)

lemma *hrp-comp-Id2*[*simp*]: $\langle hrp\text{-}comp\ A\ Id = A\rangle$
unfolding *hrp-comp-def* **by** *auto*

lemma *hn-ctxt-prod-assn-prod*:
$\langle hn\text{-}ctxt\ (R *a\ S)\ (a, b)\ (a', b') = hn\text{-}ctxt\ R\ a\ a' * hn\text{-}ctxt\ S\ b\ b'\rangle$

unfolding *hn-ctxt-def*
by *auto*

lemma *hfref-weaken-change-pre*:
assumes $(f,h) \in hfref\ P\ R\ S$
assumes $\bigwedge x.\ P\ x \Longrightarrow (fst\ R\ x,\ snd\ R\ x) = (fst\ R'\ x,\ snd\ R'\ x)$
assumes $\bigwedge y\ x.\ S\ y\ x \Longrightarrow_t S'\ y\ x$
shows $(f,h) \in hfref\ P\ R'\ S'$
proof −
have ‹$(f,h) \in hfref\ P\ R'\ S'$›
using *assms*
by (*auto simp*: *hfref-def*)
then show *?thesis*
using *hfref-imp2*[*of S S′ P R′*] *assms*(*3*) **by** *auto*
qed

lemma *norm-RETURN-o*[*to-hnr-post*]:
$\bigwedge f.\ (RETURN\ oooo\ f)\$x\$y\$z\$a = (RETURN\$(f\$x\$y\$z\$a))$
$\bigwedge f.\ (RETURN\ ooooo\ f)\$x\$y\$z\$a\$b = (RETURN\$(f\$x\$y\$z\$a\$b))$
$\bigwedge f.\ (RETURN\ oooooo\ f)\$x\$y\$z\$a\$b\$c = (RETURN\$(f\$x\$y\$z\$a\$b\$c))$
$\bigwedge f.\ (RETURN\ ooooooo\ f)\$x\$y\$z\$a\$b\$c\$d = (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d))$
$\bigwedge f.\ (RETURN\ oooooooo\ f)\$x\$y\$z\$a\$b\$c\$d\$e = (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e))$
$\bigwedge f.\ (RETURN\ ooooooooo\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g = (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g))$
$\bigwedge f.\ (RETURN\ oooooooooo\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h))$
$\bigwedge f.\ (RETURN\ \circ_{11}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i))$
$\bigwedge f.\ (RETURN\ \circ_{12}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j))$
$\bigwedge f.\ (RETURN\ \circ_{13}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l))$
$\bigwedge f.\ (RETURN\ \circ_{14}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m))$
$\bigwedge f.\ (RETURN\ \circ_{15}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n))$
$\bigwedge f.\ (RETURN\ \circ_{16}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p= (RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p))$
$\bigwedge f.\ (RETURN\ \circ_{17}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r=$
$(RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r))$
$\bigwedge f.\ (RETURN\ \circ_{18}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s=$
$(RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s))$
$\bigwedge f.\ (RETURN\ \circ_{19}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t=$
$(RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t))$
$\bigwedge f.\ (RETURN\ \circ_{20}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t\$u=$
$(RETURN\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t\$u))$
by *auto*

lemma *norm-return-o*[*to-hnr-post*]:
$\bigwedge f.\ (return\ oooo\ f)\$x\$y\$z\$a = (return\$(f\$x\$y\$z\$a))$
$\bigwedge f.\ (return\ ooooo\ f)\$x\$y\$z\$a\$b = (return\$(f\$x\$y\$z\$a\$b))$
$\bigwedge f.\ (return\ oooooo\ f)\$x\$y\$z\$a\$b\$c = (return\$(f\$x\$y\$z\$a\$b\$c))$
$\bigwedge f.\ (return\ ooooooo\ f)\$x\$y\$z\$a\$b\$c\$d = (return\$(f\$x\$y\$z\$a\$b\$c\$d))$
$\bigwedge f.\ (return\ oooooooo\ f)\$x\$y\$z\$a\$b\$c\$d\$e = (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e))$
$\bigwedge f.\ (return\ ooooooooo\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g = (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g))$
$\bigwedge f.\ (return\ oooooooooo\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h))$
$\bigwedge f.\ (return\ \circ_{11}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i))$
$\bigwedge f.\ (return\ \circ_{12}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j))$
$\bigwedge f.\ (return\ \circ_{13}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l))$
$\bigwedge f.\ (return\ \circ_{14}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m))$
$\bigwedge f.\ (return\ \circ_{15}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n))$
$\bigwedge f.\ (return\ \circ_{16}\ f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p= (return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p))$

$\bigwedge f.\ (return \circ_{17} f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r=$
$(return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r))$
$\bigwedge f.\ (return \circ_{18} f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s=$
$(return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s))$
$\bigwedge f.\ (return \circ_{19} f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t=$
$(return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t))$
$\bigwedge f.\ (return \circ_{20} f)\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t\$u=$
$(return\$(f\$x\$y\$z\$a\$b\$c\$d\$e\$g\$h\$i\$j\$l\$m\$n\$p\$r\$s\$t\$u))$
by *auto*

lemma *list-rel-update*:
fixes *R* :: ‹'*a* ⇒ '*b* :: {*heap*}⇒ *assn*›
assumes *rel*: ‹(*xs*, *ys*) ∈ ⟨*the-pure R*⟩*list-rel*› **and**
h: ‹*h* ⊨ *A* ∗ *R b bi*› **and**
p: ‹*is-pure R*›
shows ‹(*list-update xs ba bi*, *list-update ys ba b*) ∈ ⟨*the-pure R*⟩*list-rel*›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
have [*simp*]: ‹(*bi*, *b*) ∈ *the-pure R*›
using *h p* **by** (*auto simp*: *mod-star-conv R R'*)
have ‹*length xs* = *length ys*›
using *assms list-rel-imp-same-length* **by** *blast*

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

end
theory *Array-Array-List*
imports *WB-More-IICF-SML*
begin

0.1.3 Array of Array Lists

We define here array of array lists. We need arrays owning there elements. Therefore most of the rules introduced by *sep-auto* cannot lead to proofs.

fun *heap-list-all* :: ('*a* ⇒ '*b* ⇒ *assn*) ⇒ '*a list* ⇒ '*b list* ⇒ *assn* **where**
‹*heap-list-all R* [] [] = *emp*›
| ‹*heap-list-all R* (*x* # *xs*) (*y* # *ys*) = *R x y* ∗ *heap-list-all R xs ys*›
| ‹*heap-list-all R* - - = *false*›

It is often useful to speak about arrays except at one index (e.g., because it is updated).

definition *heap-list-all-nth*:: ('*a* ⇒ '*b* ⇒ *assn*) ⇒ *nat list* ⇒ '*a list* ⇒ '*b list* ⇒ *assn* **where**
‹*heap-list-all-nth R is xs ys* = *foldr* ((∗)) (*map* (λ*i*. *R* (*xs* ! *i*) (*ys* ! *i*)) *is*) *emp*›

lemma *heap-list-all-nth-emty*[*simp*]: ‹*heap-list-all-nth R* [] *xs ys* = *emp*›
unfolding *heap-list-all-nth-def* **by** *auto*

lemma *heap-list-all-nth-Cons*:
‹*heap-list-all-nth R* (*a* # *is'*) *xs ys* = *R* (*xs* ! *a*) (*ys* ! *a*) ∗ *heap-list-all-nth R is' xs ys*›
unfolding *heap-list-all-nth-def* **by** *auto*

lemma *heap-list-all-heap-list-all-nth*:

‹*length xs = length ys* ⟹ *heap-list-all R xs ys = heap-list-all-nth R* [*0*..< *length xs*] *xs ys*›
proof (*induction R xs ys rule*: *heap-list-all.induct*)
case (*2 R x xs y ys*) **note** *IH = this*
then have *IH*: ‹*heap-list-all R xs ys = heap-list-all-nth R* [*0*..<*length xs*] *xs ys*›
by *auto*
have *upt*: ‹[*0*..<*length* (*x* # *xs*)] = *0* # [*1*..<*Suc* (*length xs*)]›
by (*simp add*: *upt-rec*)
have *upt-map-Suc*: ‹[*1*..<*Suc* (*length xs*)] = *map Suc* [*0*..<*length xs*]›
by (*induction xs*) *auto*
have *map*: ‹(*map* (λ*i*. *R* ((*x* # *xs*) ! *i*) ((*y* # *ys*) ! *i*)) [*1*..<*Suc* (*length xs*)]) =
(*map* (λ*i*. *R* (*xs* ! *i*) (*ys* ! *i*)) [*0*..< (*length xs*)])›
unfolding *upt-map-Suc map-map* **by** *auto*
have *1*: ‹*heap-list-all-nth R* [*0*..<*length* (*x* # *xs*)] (*x* # *xs*) (*y* # *ys*) =
R x y ∗ *heap-list-all-nth R* [*0*..<*length xs*] *xs ys*›
unfolding *heap-list-all-nth-def upt*
by (*simp only*: *list.map foldr.simps map*) *auto*
show *?case*
using *IH* **unfolding** *1* **by** *auto*
qed *auto*

lemma *heap-list-all-nth-single*: ‹*heap-list-all-nth R* [*a*] *xs ys = R* (*xs* ! *a*) (*ys* ! *a*)›
by (*auto simp*: *heap-list-all-nth-def*)

lemma *heap-list-all-nth-mset-eq*:
assumes ‹*mset is = mset is′*›
shows ‹*heap-list-all-nth R is xs ys = heap-list-all-nth R is′ xs ys*›
using *assms*
proof (*induction is′ arbitrary*: *is*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a is′*) **note** *IH = this*(*1*) **and** *eq-is = this*(*2*)
from *eq-is* **have** ‹*a* ∈ *set is*›
by (*fastforce dest*: *mset-eq-setD*)
then obtain *ixs iys* **where**
is: ‹*is = ixs* @ *a* # *iys*›
using *eq-is* **by** (*meson split-list*)
then have *H*: ‹*heap-list-all-nth R* (*ixs* @ *iys*) *xs ys = heap-list-all-nth R is′ xs ys*›
using *IH*[*of* ‹*ixs* @ *iys*›] *eq-is* **by** *auto*
have *H′*: ‹*heap-list-all-nth R* (*ixs* @ *a* # *iys*) *xs ys = heap-list-all-nth R* (*a* # *ixs* @ *iys*) *xs ys*›
for *xs ys*
by (*induction ixs*)(*auto simp*: *heap-list-all-nth-Cons star-aci*(*3*))
show *?case*
using *H*[*symmetric*] **by** (*auto simp*: *heap-list-all-nth-Cons is H′*)
qed

lemma *heap-list-add-same-length*:
‹*h* ⊨ *heap-list-all R′ xs p* ⟹ *length p = length xs*›
by (*induction R′ xs p arbitrary*: *h rule*: *heap-list-all.induct*) (*auto elim*!: *mod-starE*)

lemma *heap-list-all-nth-Suc*:
assumes *a*: ‹*a* > *1*›
shows ‹*heap-list-all-nth R* [*Suc 0*..<*a*] (*x* # *xs*) (*y* # *ys*) =
heap-list-all-nth R [*0*..<*a*−*1*] *xs ys*›
proof −
have *upt*: ‹[*0*..< *a*] = *0* # [*1*..< *a*]›

using *a* **by** (*simp add*: *upt-rec*)
have *upt-map-Suc*: ‹[*Suc 0*..<*a*] = *map Suc* [*0*..< *a*−*1*]›
using *a* **by** (*auto simp*: *map-Suc-upt*)
have *map*: ‹(*map* (λ*i*. *R* ((*x* # *xs*) ! *i*) ((*y* # *ys*) ! *i*)) [*Suc 0*..<*a*]) =
(*map* (λ*i*. *R* (*xs* ! *i*) (*ys* ! *i*)) [*0*..<*a*−*1*])›
unfolding *upt-map-Suc map-map* **by** *auto*
show *?thesis*
unfolding *heap-list-all-nth-def* **unfolding** *map* **..**
qed

lemma *heap-list-all-nth-append*:
‹*heap-list-all-nth R* (*is* @ *is*′) *xs ys* = *heap-list-all-nth R is xs ys* ∗ *heap-list-all-nth R is*′ *xs ys*›
by (*induction is*) (*auto simp*: *heap-list-all-nth-Cons star-aci*)

lemma *heap-list-all-heap-list-all-nth-eq*:
‹*heap-list-all R xs ys* = *heap-list-all-nth R* [*0*..< *length xs*] *xs ys* ∗ ↑(*length xs* = *length ys*)›
by (*induction R xs ys rule*: *heap-list-all.induct*)
(*auto simp del*: *upt-Suc upt-Suc-append*
simp: *upt-rec*[*of 0*] *heap-list-all-nth-single star-aci*(*3*)
heap-list-all-nth-Cons heap-list-all-nth-Suc)

lemma *heap-list-all-nth-remove1*: ‹*i* ∈ *set is* ⟹
heap-list-all-nth R is xs ys = *R* (*xs* ! *i*) (*ys* ! *i*) ∗ *heap-list-all-nth R* (*remove1 i is*) *xs ys*›
using *heap-list-all-nth-mset-eq*[*of* ‹*is*› ‹*i* # *remove1 i is*›]
by (*auto simp*: *heap-list-all-nth-Cons*)

definition *arrayO-assn* :: ‹(′*a* ⇒ ′*b*::*heap* ⇒ *assn*) ⇒ ′*a list* ⇒ ′*b array* ⇒ *assn*› **where**
‹*arrayO-assn R*′ *xs axs* ≡ $\exists_A$ *p*. *array-assn id-assn p axs* ∗ *heap-list-all R*′ *xs p*›

definition *arrayO-except-assn*:: ‹(′*a* ⇒ ′*b*::*heap* ⇒ *assn*) ⇒ *nat list* ⇒ ′*a list* ⇒ ′*b array* ⇒ - ⇒ *assn*›
where
‹*arrayO-except-assn R*′ *is xs axs f* ≡
$\exists_A$ *p*. *array-assn id-assn p axs* ∗ *heap-list-all-nth R*′ (*fold remove1 is* [*0*..<*length xs*]) *xs p* ∗
↑ (*length xs* = *length p*) ∗ *f p*›

lemma *arrayO-except-assn-array0*: ‹*arrayO-except-assn R* [] *xs asx* (λ-. *emp*) = *arrayO-assn R xs asx*›
proof −
have ‹(*h* ⊨ *array-assn id-assn p asx* ∗ *heap-list-all-nth R* [*0*..<*length xs*] *xs p* ∧ *length xs* = *length p*)
=
(*h* ⊨ *array-assn id-assn p asx* ∗ *heap-list-all R xs p*)› (**is** ‹*?a* = *?b*›) **for** *h p*
proof (*rule iffI*)
assume *?a*
then show *?b*
by (*auto simp*: *heap-list-all-heap-list-all-nth*)
next
assume *?b*
then have ‹*length xs* = *length p*›
by (*auto simp*: *heap-list-add-same-length mod-star-conv*)
then show *?a*
using ‹*?b*›
by (*auto simp*: *heap-list-all-heap-list-all-nth*)
qed
then show *?thesis*
unfolding *arrayO-except-assn-def arrayO-assn-def* **by** (*auto simp*: *ex-assn-def*)
qed

lemma *arrayO-except-assn-array0-index*:
  ‹i < *length xs* ⟹ *arrayO-except-assn R* [i] *xs asx* (λp. *R* (*xs* ! i) (p ! i)) = *arrayO-assn R xs asx*›
  unfolding *arrayO-except-assn-array0*[*symmetric*] *arrayO-except-assn-def*
  using *heap-list-all-nth-remove1*[*of i* ‹[*0*..<*length xs*]› *R xs*] **by** (*auto simp*: *star-aci*(*2*,*3*))

lemma *arrayO-nth-rule*[*sep-heap-rules*]:
  assumes *i*: ‹i < *length a*›
  shows ‹ <*arrayO-assn* (*arl-assn R*) *a ai*> *Array.nth ai i* <λr. *arrayO-except-assn* (*arl-assn R*) [i] *a ai*
  ($\lambda r'$. *arl-assn R* (*a* ! i) r ∗ ↑(r = r' ! i))>›
proof −
  have *i-le*: ‹i < *Array.length h ai*› **if** ‹(*h*, *as*) ⊨ *arrayO-assn* (*arl-assn R*) *a ai*› **for** *h as*
    using *that i* **unfolding** *arrayO-assn-def array-assn-def is-array-def*
    by (*auto simp*: *run.simps tap-def arrayO-assn-def*
        mod-star-conv array-assn-def is-array-def
        Abs-assn-inverse heap-list-add-same-length length-def snga-assn-def)
  have *A*: ‹*Array.get h ai* ! i = p ! i› **if** ‹(*h*, *as*) ⊨
      array-assn id-assn p ai ∗
      heap-list-all-nth (*arl-assn R*) (*remove1 i* [*0*..<*length p*]) *a p* ∗
      arl-assn R (*a* ! i) (p ! i)›
    for *as p h*
    using *that*
    by (*auto simp*: *mod-star-conv array-assn-def is-array-def Array.get-def snga-assn-def*
        Abs-assn-inverse)
  show *?thesis*
    unfolding *hoare-triple-def Let-def*
    apply (*clarify*, *intro allI impI conjI*)
    using *assms A*
      apply (*auto simp*: *hoare-triple-def Let-def i-le execute-simps relH-def in-range.simps*
        arrayO-except-assn-array0-index[*of i*, *symmetric*]
        elim!: *run-elims*
        intro!: *norm-pre-ex-rule*)
    apply (*auto simp*: *arrayO-except-assn-def*)
    done
qed

definition *length-a* :: ‹$'a$::*heap array* ⇒ *nat Heap*› **where**
  ‹*length-a xs* = *Array.len xs*›

lemma *length-a-rule*[*sep-heap-rules*]:
  ‹<*arrayO-assn R x xi*> *length-a xi* <λr. *arrayO-assn R x xi* ∗ ↑(r = *length x*)>$_t$›
  by (*sep-auto simp*: *arrayO-assn-def length-a-def array-assn-def is-array-def mod-star-conv*
      dest: *heap-list-add-same-length*)

lemma *length-a-hnr*[*sepref-fr-rules*]:
  ‹(*length-a*, *RETURN o op-list-length*) ∈ (*arrayO-assn R*)^k $\rightarrow_a$ *nat-assn*›
  by *sepref-to-hoare sep-auto*

lemma *le-length-ll-nemptyD*: ‹b < *length-ll a ba* ⟹ *a* ! *ba* ≠ []›
  by (*auto simp*: *length-ll-def*)

definition *length-aa* :: ‹($'a$::*heap array-list*) *array* ⇒ *nat* ⇒ *nat Heap*› **where**
  ‹*length-aa xs i* = *do* {
     x ← *Array.nth xs i*;
    arl-length x}›

lemma *length-aa-rule*[*sep-heap-rules*]:
‹b < *length xs* ⟹ <*arrayO-assn* (*arl-assn R*) *xs a*> *length-aa a b*
<λ*r*. *arrayO-assn* (*arl-assn R*) *xs a* ∗ ↑ (*r* = *length-ll xs b*)>$_t$›
unfolding *length-aa-def*
apply *sep-auto*
apply (*sep-auto simp*: *arrayO-except-assn-def arl-length-def arl-assn-def*
eq-commute[*of* ‹(-, -)›] *hr-comp-def length-ll-def*)
apply (*sep-auto simp*: *arrayO-except-assn-def arl-length-def arl-assn-def*
eq-commute[*of* ‹(-, -)›] *is-array-list-def hr-comp-def length-ll-def list-rel-def*
dest: *list-all2-lengthD*)[]
unfolding *arrayO-assn-def*[*symmetric*] *arl-assn-def*[*symmetric*]
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of b*])
apply *simp*
unfolding *arrayO-except-assn-def arl-assn-def hr-comp-def*
apply *sep-auto*
done

lemma *length-aa-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-aa*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈
[λ(*xs*, *i*). *i* < *length xs*]$_a$ (*arrayO-assn* (*arl-assn R*))^k ∗$_a$ *nat-assn*^{k} → *nat-assn*›
by *sepref-to-hoare sep-auto*

definition *nth-aa* **where**
‹*nth-aa xs i j* = *do* {
x ← *Array.nth xs i*;
y ← *arl-get x j*;
return y}›

lemma *models-heap-list-all-models-nth*:
‹(*h*, *as*) ⊨ *heap-list-all R a b* ⟹ *i* < *length a* ⟹ ∃ *as*′. (*h*, *as*′) ⊨ *R* (*a*!*i*) (*b*!*i*)›
by (*induction R a b arbitrary*: *as i rule*: *heap-list-all.induct*)
(*auto simp*: *mod-star-conv nth-Cons elim*!: *less-SucE split*: *nat.splits*)

definition *nth-ll* :: ′*a list list* ⇒ *nat* ⇒ *nat* ⇒ ′*a* **where**
‹*nth-ll l i j* = *l* ! *i* ! *j*›

lemma *nth-aa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 nth-aa*, *uncurry2* (*RETURN* ∘∘∘ *nth-ll*)) ∈
[λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-ll l i*]$_a$
(*arrayO-assn* (*arl-assn R*))^k ∗$_a$ *nat-assn*^{k} ∗$_a$ *nat-assn*^{k} → *R*›
proof −
obtain *R*′ **where** *R*: ‹*the-pure R* = *R*′› **and** *R*′: ‹*R* = *pure R*′›
using *p* **by** *fastforce*
have *H*: ‹*list-all2* (λ*x x*′. (*x*, *x*′) ∈ *the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *R*′))) *bc* (*a* ! *ba*) ⟹
b < *length* (*a* ! *ba*) ⟹
(*bc* ! *b*, *a* ! *ba* ! *b*) ∈ *R*′› **for** *bc a ba b*
by (*auto simp add*: *ent-refl-true list-all2-conv-all-nth is-pure-alt-def pure-app-eq*[*symmetric*])
show *?thesis*
apply *sepref-to-hoare*
apply (*subst* (*2*) *arrayO-except-assn-array0-index*[*symmetric*])
apply (*solves* ‹*auto*›)[]
apply (*sep-auto simp*: *nth-aa-def nth-ll-def length-ll-def*)
apply (*sep-auto simp*: *arrayO-except-assn-def arrayO-assn-def arl-assn-def hr-comp-def list-rel-def*
list-all2-lengthD
star-aci(*3*) *R R*′ *pure-def H*)

done
qed

definition *append-el-aa* :: $('a::\{default,heap\}$ *array-list*) *array* $\Rightarrow$
nat $\Rightarrow$ $'a \Rightarrow$ $('a$ *array-list*) *array Heap***where**
append-el-aa $\equiv$ $\lambda a\ i\ x.$ *do* {
$j \leftarrow$ *Array.nth a i*;
$a' \leftarrow$ *arl-append j x*;
Array.upd i a' a
}

lemma *sep-auto-is-stupid*:
fixes R :: ‹$'a \Rightarrow 'b::\{heap,default\} \Rightarrow assn$›
assumes p: ‹*is-pure R*›
shows
‹$<\exists_A p.\ R1\ p * R2\ p * arl\text{-}assn\ R\ l'\ aa * R\ x\ x' * R4\ p>$
$arl\text{-}append\ aa\ x' <\lambda r.\ (\exists_A p.\ arl\text{-}assn\ R\ (l'\ @\ [x])\ r * R1\ p * R2\ p * R\ x\ x' * R4\ p * true) >$›
proof −
obtain R' **where** R: ‹*the-pure* $R = R'$› **and** R': ‹$R = pure\ R'$›
using p **by** *fastforce*
have *bbi*: ‹$(x', x) \in$ *the-pure R*› **if**
‹$(aa, bb) \models is\text{-}array\text{-}list\ (ba\ @\ [x'])\ (a, baa) * R1\ p * R2\ p * pure\ R'\ x\ x' * R4\ p * true$›
for *aa bb a ba baa p*
using *that* **by** (*auto simp*: *mod-star-conv R R'*)
show *?thesis*
unfolding *arl-assn-def hr-comp-def*
by (*sep-auto simp*: *list-rel-def R R' intro*!: *list-all2-appendI dest*!: *bbi*)
qed

declare *arrayO-nth-rule*[*sep-heap-rules*]

lemma *heap-list-all-nth-cong*:
assumes
‹$\forall i \in set\ is.\ xs\ !\ i = xs'\ !\ i$› **and**
‹$\forall i \in set\ is.\ ys\ !\ i = ys'\ !\ i$›
shows ‹*heap-list-all-nth R is xs ys* = *heap-list-all-nth R is xs' ys'*›
using *assms* **by** (*induction* ‹*is*›) (*auto simp*: *heap-list-all-nth-Cons*)

lemma *append-aa-hnr*[*sepref-fr-rules*]:
fixes R :: ‹$'a \Rightarrow 'b :: \{heap,\ default\} \Rightarrow assn$›
assumes p: ‹*is-pure R*›
shows
‹(*uncurry2 append-el-aa*, *uncurry2* (*RETURN* ∘∘∘ *append-ll*)) ∈
$[\lambda((l,i),x).\ i < length\ l]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a R^k \rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
proof −
obtain R' **where** R: ‹*the-pure* $R = R'$› **and** R': ‹$R = pure\ R'$›
using p **by** *fastforce*
have [*simp*]: ‹$(\exists_A x.\ arrayO\text{-}assn\ (arl\text{-}assn\ R)\ a\ ai * R\ x\ r * true * \uparrow (x = a\ !\ ba\ !\ b)) =$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R)\ a\ ai * R\ (a\ !\ ba\ !\ b)\ r * true)$› **for** *a ai ba b r*
by (*auto simp*: *ex-assn-def*)
show *?thesis* — TODO tune proof
apply *sepref-to-hoare*
apply (*sep-auto simp*: *append-el-aa-def*)
apply (*simp add*: *arrayO-except-assn-def*)
apply (*rule sep-auto-is-stupid*[*OF p*])

apply (*sep-auto simp*: *array-assn-def is-array-def append-ll-def*)
apply (*simp add*: *arrayO-except-assn-array0*[*symmetric*] *arrayO-except-assn-def*)
apply (*subst-tac* (*2*) $i = ba$ **in** *heap-list-all-nth-remove1*)
apply (*solves* ‹*simp*›)
apply (*simp add*: *array-assn-def is-array-def*)
apply (*rule-tac* x=‹$p[ba := (ab, bc)]$› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)xs'=a **and** ys'=p **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)[*2*]
apply (*auto simp*: *star-aci*)
done
qed

definition *update-aa* :: (*'a*::{*heap*} *array-list*) *array* ⇒ *nat* ⇒ *nat* ⇒ *'a* ⇒ (*'a array-list*) *array Heap*
where
‹*update-aa a i j y* = *do* {
$x \leftarrow$ *Array.nth a i*;
$a' \leftarrow$ *arl-set x j y*;
Array.upd i a' a
}› — is the Array.upd really needed?

definition *update-ll* :: *'a list list* ⇒ *nat* ⇒ *nat* ⇒ *'a* ⇒ *'a list list* **where**
‹*update-ll xs i j y* = $xs[i := (xs\ !\ i)[j := y]]$›

declare *nth-rule*[*sep-heap-rules del*]
declare *arrayO-nth-rule*[*sep-heap-rules*]

TODO: is it possible to be more precise and not drop the $\uparrow ((aa,\ bc) = r'\ !\ bb)$

lemma *arrayO-except-assn-arl-set*[*sep-heap-rules*]:
fixes R :: ‹*'a* ⇒ *'b* :: {*heap*}⇒ *assn*›
assumes *p*: ‹*is-pure R*› **and** ‹$bb <$ *length a*› **and**
‹$ba <$ *length-ll a bb*›
shows ‹
<*arrayO-except-assn* (*arl-assn R*) [*bb*] *a ai* ($\lambda r'$. *arl-assn R* (*a* ! *bb*) (*aa*, *bc*) ∗
↑ ((*aa*, *bc*) = *r'* ! *bb*)) ∗ *R b bi*>
arl-set (*aa*, *bc*) *ba bi*
<λ(*aa*, *bc*). *arrayO-except-assn* (*arl-assn R*) [*bb*] *a ai*
($\lambda r'$. *arl-assn R* ((*a* ! *bb*)[*ba* := *b*]) (*aa*, *bc*)) ∗ *R b bi* ∗ *true*>›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
show *?thesis*
using *assms*
apply (*sep-auto simp*: *arrayO-except-assn-def arl-assn-def hr-comp-def list-rel-imp-same-length*
list-rel-update length-ll-def)
done
qed

lemma *update-aa-rule*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› **and** ‹$bb <$ *length a*› **and** ‹$ba <$ *length-ll a bb*›
shows ‹<*R b bi* ∗ *arrayO-assn* (*arl-assn R*) *a ai*> *update-aa ai bb ba bi*
<λ*r*. *R b bi* ∗ ($\exists_A x$. *arrayO-assn* (*arl-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
using *assms*
apply (*sep-auto simp add*: *update-aa-def update-ll-def p*)
apply (*sep-auto simp add*: *update-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def*)
apply (*subst-tac i*=*bb* **in** *arrayO-except-assn-array0-index*[*symmetric*])
apply (*solves* ‹*simp*›)

apply (*subst arrayO-except-assn-def*)
apply (*auto simp add*: *update-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def*)

apply (*rule-tac x*=‹*p*[*bb* := (*aa*, *bc*)]› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs*′=*a* **and** *ys*′=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)
apply (*solves* ‹*auto*›)
apply (*auto simp*: *star-aci*)
done

lemma *update-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa*, *uncurry3* (*RETURN oooo update-ll*)) ∈ [λ(((*l*,*i*), *j*), *x*). *i* < *length l* ∧ *j* < *length-ll l i*]$_a$ (*arrayO-assn* (*arl-assn R*))^d $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ R^k → (*arrayO-assn* (*arl-assn R*))›
by *sepref-to-hoare* (*sep-auto simp*: *assms*)

definition *set-butlast-ll* **where**
‹*set-butlast-ll xs i* = *xs*[*i* := *butlast* (*xs* ! *i*)]›

definition *set-butlast-aa* :: (′*a*::{*heap*} *array-list*) *array* ⇒ *nat* ⇒ (′*a array-list*) *array Heap* **where**
‹*set-butlast-aa a i* = *do* {
x ← *Array.nth a i*;
a′ ← *arl-butlast x*;
Array.upd i a′ *a*
}› — Replace the *i*-th element by the itself except the last element.

lemma *list-rel-butlast*:
assumes *rel*: ‹(*xs*, *ys*) ∈ ⟨*R*⟩*list-rel*›
shows ‹(*butlast xs*, *butlast ys*) ∈ ⟨*R*⟩*list-rel*›
proof −
have ‹*length xs* = *length ys*›
using *assms list-rel-imp-same-length* **by** *blast*
then show *?thesis*
using *rel*
by (*induction xs ys rule*: *list-induct2*) (*auto split*: *nat.splits*)
qed

lemma *arrayO-except-assn-arl-butlast*:
assumes ‹*b* < *length a*› **and**
‹*a* ! *b* ≠ []›
shows
‹<*arrayO-except-assn* (*arl-assn R*) [*b*] *a ai* (λ*r*′. *arl-assn R* (*a* ! *b*) (*aa*, *ba*) ∗
↑ ((*aa*, *ba*) = *r*′ ! *b*))>
arl-butlast (*aa*, *ba*)
<λ(*aa*, *ba*). *arrayO-except-assn* (*arl-assn R*) [*b*] *a ai* (λ*r*′. *arl-assn R* (*butlast* (*a* ! *b*)) (*aa*, *ba*)∗ *true*)>›
proof −
show *?thesis*
using *assms*
apply (*subst* (*1*) *arrayO-except-assn-def*)
apply (*sep-auto simp*: *arl-assn-def hr-comp-def list-rel-imp-same-length*
list-rel-update
intro: *list-rel-butlast*)
apply (*subst* (*1*) *arrayO-except-assn-def*)
apply (*rule-tac x*=‹*p*› **in** *ent-ex-postI*)

apply (*sep-auto intro*: *list-rel-butlast*)
done
qed

lemma *set-butlast-aa-rule*[*sep-heap-rules*]:
assumes ‹*is-pure R*› **and**
‹$b <$ *length a*› **and**
‹*a* ! $b \neq []$›
shows ‹<*arrayO-assn* (*arl-assn R*) *a ai*> *set-butlast-aa ai b*
<$\lambda r.$ ($\exists_A x.$ *arrayO-assn* (*arl-assn R*) *x r* $*$ $\uparrow$ ($x =$ *set-butlast-ll a b*))>$_t$›
proof −
note *arrayO-except-assn-arl-butlast*[*sep-heap-rules*]
note *arl-butlast-rule*[*sep-heap-rules del*]
have ‹$\bigwedge b$ *bi*.
$b <$ *length a* $\Longrightarrow$
a ! $b \neq []$ $\Longrightarrow$
a $::_i$ *TYPE*('*a list list*) $\Longrightarrow$
b $::_i$ *TYPE*(*nat*) $\Longrightarrow$
nofail (*RETURN* (*set-butlast-ll a b*)) $\Longrightarrow$
<$\uparrow$ ((*bi*, *b*) $\in$ *nat-rel*) $*$
arrayO-assn (*arl-assn R*) *a*
ai> *set-butlast-aa ai*
bi <$\lambda r.$ $\uparrow$ ((*bi*, *b*) $\in$ *nat-rel*) $*$
true $*$
($\exists_A x.$
arrayO-assn (*arl-assn R*) *x r* $*$
$\uparrow$ (*RETURN x* $\leq$ *RETURN* (*set-butlast-ll a b*)))>$_t$›
apply (*sep-auto simp add*: *set-butlast-aa-def set-butlast-ll-def assms*)

apply (*sep-auto simp add*: *set-butlast-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def*)
apply (*subst-tac i=b* **in** *arrayO-except-assn-array0-index*[*symmetric*])
apply (*solves* ‹*simp*›)
apply (*subst arrayO-except-assn-def*)
apply (*auto simp add*: *set-butlast-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def*)

apply (*rule-tac x*=‹*p*[*b* := (*aa*, *ba*)]› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)
apply (*solves* ‹*auto*›)
apply (*solves* ‹*auto*›)
done
then show *?thesis*
using *assms* **by** *sep-auto*
qed

lemma *set-butlast-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry set-butlast-aa*, *uncurry* (*RETURN oo set-butlast-ll*)) $\in$
[$\lambda(l,i).$ $i <$ *length l* $\wedge$ *l* ! $i \neq []$]$_a$ (*arrayO-assn* (*arl-assn R*))^d $*_a$ *nat-assn*^{k} $\rightarrow$ (*arrayO-assn* (*arl-assn R*))›
using *assms* **by** *sepref-to-hoare sep-auto*

definition *last-aa* :: ('*a*::*heap array-list*) *array* $\Rightarrow$ *nat* $\Rightarrow$ '*a Heap* **where**
‹*last-aa xs i* = *do* {
$x \leftarrow$ *Array.nth xs i*;

arl-last x
}›

definition *last-ll* :: *'a list list* ⇒ *nat* ⇒ *'a* **where**
‹*last-ll xs i* = *last* (*xs* ! *i*)›

lemma *last-aa-rule*[*sep-heap-rules*]:
assumes
p: ‹*is-pure R*› **and**
‹*b* < *length a*› **and**
‹*a* ! *b* ≠ []›
shows ‹
<*arrayO-assn* (*arl-assn R*) *a ai*>
last-aa ai b
<λr. *arrayO-assn* (*arl-assn R*) *a ai* ∗ ($\exists_A x$. *R x r* ∗ ↑ (*x* = *last-ll a b*))>$_t$›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
note *arrayO-except-assn-arl-butlast*[*sep-heap-rules*]
note *arl-butlast-rule*[*sep-heap-rules del*]
have ‹$\bigwedge b$.
b < *length a* ⟹
a ! *b* ≠ [] ⟹
<*arrayO-assn* (*arl-assn R*) *a ai*>
last-aa ai b
<λr. *arrayO-assn* (*arl-assn R*) *a ai* ∗ ($\exists_A x$. *R x r* ∗ ↑ (*x* = *last-ll a b*))>$_t$›
apply (*sep-auto simp add*: *last-aa-def last-ll-def assms*)

apply (*sep-auto simp add*: *last-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def arl-assn-def*)
apply (*subst-tac i*=*b* **in** *arrayO-except-assn-array0-index*[*symmetric*])
apply (*solves* ‹*simp*›)
apply (*subst arrayO-except-assn-def*)
apply (*auto simp add*: *last-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def*)

apply (*rule-tac x*=‹*p*› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)
apply (*solves* ‹*auto*›)

apply (*rule-tac x*=‹*bb*› **in** *ent-ex-postI*)
unfolding *R* **unfolding** *R'*
apply (*sep-auto simp*: *pure-def param-last*)
done
from *this*[*of b*] **show** *?thesis*
using *assms* **unfolding** *R'* **by** *blast*
qed

lemma *last-aa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows ‹(*uncurry last-aa*, *uncurry* (*RETURN oo last-ll*)) ∈
[λ(*l*,*i*). *i* < *length l* ∧ *l* ! *i* ≠ []]$_a$ (*arrayO-assn* (*arl-assn R*))^k $*_a$ *nat-assn*^{k} → *R*›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
note *arrayO-except-assn-arl-butlast*[*sep-heap-rules*]

```
  note arl-butlast-rule[sep-heap-rules del]
  show ?thesis
    using assms by sepref-to-hoare sep-auto
qed

definition nth-a :: ‹('a::heap array-list) array ⇒ nat ⇒ ('a array-list) Heap› where
 ‹nth-a xs i = do {
     x ← Array.nth xs i;
     arl-copy x}›

lemma nth-a-hnr[sepref-fr-rules]:
  ‹(uncurry nth-a, uncurry (RETURN oo op-list-get)) ∈
     [λ(xs, i). i < length xs]a (arrayO-assn (arl-assn R))^k *a nat-assn^k → arl-assn R›
  unfolding nth-a-def
  apply sepref-to-hoare
  subgoal for b b' xs a — TODO proof
    apply sep-auto
    apply (subst arrayO-except-assn-array0-index[symmetric, of b])
     apply simp
    apply (sep-auto simp: arrayO-except-assn-def arl-length-def arl-assn-def
        eq-commute[of ‹(-, -)›] hr-comp-def length-ll-def)
    done
  done

 definition swap-aa :: ('a::heap array-list) array ⇒ nat ⇒ nat ⇒ nat ⇒ ('a array-list) array Heap
where
  ‹swap-aa xs k i j = do {
    xi ← nth-aa xs k i;
    xj ← nth-aa xs k j;
    xs ← update-aa xs k i xj;
    xs ← update-aa xs k j xi;
    return xs
  }›

definition swap-ll where
  ‹swap-ll xs k i j = list-update xs k (swap (xs!k) i j)›

lemma nth-aa-heap[sep-heap-rules]:
  assumes p: ‹is-pure R› and ‹b < length aa› and ‹ba < length-ll aa b›
  shows ‹
   <arrayO-assn (arl-assn R) aa a>
   nth-aa a b ba
   <λr. ∃Ax. arrayO-assn (arl-assn R) aa a *
              (R x r *
               ↑ (x = nth-ll aa b ba)) *
              true>›
proof -
  have ‹<arrayO-assn (arl-assn R) aa a *
        nat-assn b b *
        nat-assn ba ba>
       nth-aa a b ba
       <λr. ∃Ax. arrayO-assn (arl-assn R) aa a *
                 nat-assn b b *
                 nat-assn ba ba *
                 R x r *
                 true *
```

↑ (*x* = *nth-ll aa b ba*)>›
using *p assms nth-aa-hnr*[*of R*] **unfolding** *hfref-def hn-refine-def*
by *auto*
then show *?thesis*
unfolding *hoare-triple-def*
by (*auto simp*: *Let-def pure-def*)
qed

lemma *update-aa-rule-pure*:
assumes *p*: ‹*is-pure R*› **and** ‹*b* < *length aa*› **and** ‹*ba* < *length-ll aa b*› **and**
b: ‹(*bb*, *be*) ∈ *the-pure R*›
shows ‹
<*arrayO-assn* (*arl-assn R*) *aa a*>
update-aa a b ba bb
<λ*r*. $\exists_A$*x*. *invalid-assn* (*arrayO-assn* (*arl-assn R*)) *aa a* ∗ *arrayO-assn* (*arl-assn R*) *x r* ∗
true ∗
↑ (*x* = *update-ll aa b ba be*)>›
proof −
obtain *R′* **where** *R′*: ‹*R′* = *the-pure R*› **and** *RR′*: ‹*R* = *pure R′*›
using *p* **by** *fastforce*
have *bb*: ‹*pure R′ be bb* = ↑((*bb*, *be*) ∈ *R′*)›
by (*auto simp*: *pure-def*)
have ‹ <*arrayO-assn* (*arl-assn R*) *aa a* ∗ *nat-assn b b* ∗ *nat-assn ba ba* ∗ *R be bb*>
update-aa a b ba bb
<λ*r*. $\exists_A$*x*. *invalid-assn* (*arrayO-assn* (*arl-assn R*)) *aa a* ∗ *nat-assn b b* ∗ *nat-assn ba ba* ∗
R be bb ∗
arrayO-assn (*arl-assn R*) *x r* ∗
true ∗
↑ (*x* = *update-ll aa b ba be*)>›
using *p assms update-aa-hnr*[*of R*] **unfolding** *hfref-def hn-refine-def*
by *auto*
then show *?thesis*
using *b* **unfolding** *R′*[*symmetric*] **unfolding** *hoare-triple-def RR′ bb*
by (*auto simp*: *Let-def pure-def*)
qed

lemma *length-update-ll*[*simp*]: ‹*length* (*update-ll a bb b c*) = *length a*›
unfolding *update-ll-def* **by** *auto*

lemma *length-ll-update-ll*:
‹*bb* < *length a* ⟹ *length-ll* (*update-ll a bb b c*) *bb* = *length-ll a bb*›
unfolding *length-ll-def update-ll-def* **by** *auto*

lemma *swap-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 swap-aa*, *uncurry3* (*RETURN oooo swap-ll*)) ∈
[λ(((*xs*, *k*), *i*), *j*). *k* < *length xs* ∧ *i* < *length-ll xs k* ∧ *j* < *length-ll xs k*]$_a$
(*arrayO-assn* (*arl-assn R*))^d $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} → (*arrayO-assn* (*arl-assn R*))›
proof −
note *update-aa-rule-pure*[*sep-heap-rules*]
obtain *R′* **where** *R′*: ‹*R′* = *the-pure R*› **and** *RR′*: ‹*R* = *pure R′*›
using *assms* **by** *fastforce*
have [*simp*]: ‹*the-pure* (λ*a b*. ↑ ((*b*, *a*) ∈ *R′*)) = *R′*›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
using *assms* **unfolding** *R′*[*symmetric*] **unfolding** *RR′*

apply *sepref-to-hoare*
apply (*sep-auto simp*: *swap-aa-def swap-ll-def arrayO-except-assn-def length-ll-update-ll*)
by (*sep-auto simp*: *update-ll-def swap-def nth-ll-def list-update-swap*)
qed

It is not possible to do a direct initialisation: there is no element that can be put everywhere.

definition *arrayO-ara-empty-sz* **where**
⟨*arrayO-ara-empty-sz n* =
(*let xs* = *fold* (λ- *xs*. [] # *xs*) [*0*..<*n*] [] *in*
op-list-copy xs)
⟩

lemma *heap-list-all-list-assn*: ⟨*heap-list-all R x y* = *list-assn R x y*⟩
by (*induction R x y rule*: *heap-list-all.induct*) *auto*

lemma *of-list-op-list-copy-arrayO*[*sepref-fr-rules*]:
⟨(*Array.of-list*, *RETURN* ∘ *op-list-copy*) ∈ (*list-assn* (*arl-assn R*))^d $\rightarrow_a$ *arrayO-assn* (*arl-assn R*)⟩
apply *sepref-to-hoare*
apply (*sep-auto simp*: *arrayO-assn-def array-assn-def*)
apply (*rule-tac ?psi*=⟨*xa* $\mapsto_a$ *xi* ∗ *list-assn* (*arl-assn R*) *x xi* $\Longrightarrow_A$ *is-array xi xa* ∗ *heap-list-all* (*arl-assn R*) *x xi* ∗ *true*⟩ **in** *asm-rl*)
by (*sep-auto simp*: *heap-list-all-list-assn is-array-def*)

sepref-definition
arrayO-ara-empty-sz-code
is *RETURN o arrayO-ara-empty-sz*
:: ⟨*nat-assn*^{k} $\rightarrow_a$ *arrayO-assn* (*arl-assn* (*R*::′*a* ⇒ ′*b*::{*heap*, *default*} ⇒ *assn*))⟩
unfolding *arrayO-ara-empty-sz-def op-list-empty-def*[*symmetric*]
apply (*rewrite at* ⟨(#) ⌑⟩ *op-arl-empty-def*[*symmetric*])
apply (*rewrite at* ⟨*fold* - - ⌑⟩ *op-HOL-list-empty-def*[*symmetric*])
supply [[*goals-limit* = *1*]]
by *sepref*

definition *init-lrl* :: ⟨*nat* ⇒ ′*a list list*⟩ **where**
⟨*init-lrl n* = *replicate n* []⟩

lemma *arrayO-ara-empty-sz-init-lrl*: ⟨*arrayO-ara-empty-sz n* = *init-lrl n*⟩
by (*induction n*) (*auto simp*: *arrayO-ara-empty-sz-def init-lrl-def*)

lemma *arrayO-raa-empty-sz-init-lrl*[*sepref-fr-rules*]:
⟨(*arrayO-ara-empty-sz-code*, *RETURN o init-lrl*) ∈
nat-assn^k $\rightarrow_a$ *arrayO-assn* (*arl-assn R*)⟩
using *arrayO-ara-empty-sz-code.refine* **unfolding** *arrayO-ara-empty-sz-init-lrl* **.**

definition (**in** −) *shorten-take-ll* **where**
⟨*shorten-take-ll L j W* = *W*[*L* := *take j* (*W* ! *L*)]⟩

definition (**in** −) *shorten-take-aa* **where**
⟨*shorten-take-aa L j W* = *do* {
(*a*, *n*) ← *Array.nth W L*;
Array.upd L (*a*, *j*) *W*
}⟩

lemma *Array-upd-arrayO-except-assn*[*sep-heap-rules*]:
  assumes
    ‹$ba \leq length\ (b\ !\ a)$› **and**
    ‹$a < length\ b$›
  shows ‹<*arrayO-except-assn* (*arl-assn R*) [*a*] *b bi*
        ($\lambda r'$. *arl-assn R* (*b* ! *a*) (*aaa*, *n*) $*$ ↑ ((*aaa*, *n*) = r' ! *a*))>
      Array.upd a (*aaa*, *ba*) *bi*
      <λr. $\exists_A x$. *arrayO-assn* (*arl-assn R*) *x r* $*$ *true* $*$
          ↑ (*x* = *b*[*a* := *take ba* (*b* ! *a*)])>›
proof −
  have [*simp*]: ‹*ba* ≤ *length l′*›
    if
      ‹*ba* ≤ *length* (*b* ! *a*)› **and**
      aa: ‹(*take n l′*, *b* ! *a*) ∈ ⟨*the-pure R*⟩*list-rel*›
    for *l′* :: ‹′*b list*›
  proof −
    show *?thesis*
      using *list-rel-imp-same-length*[*OF aa*] *that*
      by *auto*
  qed
  have [*simp*]: ‹(*take ba l′*, *take ba* (*b* ! *a*)) ∈ ⟨*the-pure R*⟩*list-rel*›
    if
      ‹*ba* ≤ *length* (*b* ! *a*)› **and**
      ‹*n* ≤ *length l′*› **and**
      take: ‹(*take n l′*, *b* ! *a*) ∈ ⟨*the-pure R*⟩*list-rel*›
    for *l′* :: ‹′*b list*›
  proof −
    have [*simp*]: ‹*n* = *length* (*b* ! *a*)›
      using *list-rel-imp-same-length*[*OF take*] *that* **by** *auto*
    have *1*: ‹*take ba l′* = *take ba* (*take n l′*)›
      using *that* **by** (*auto simp*: *min-def*)
    show *?thesis*
      using *take*
      unfolding *1*
      by (*rule list-rel-take*)
  qed

  have [*simp*]: ‹*heap-list-all-nth* (*arl-assn R*) (*remove1 a* [*0*..<*length p*])
        (*b*[*a* := *take ba* (*b* ! *a*)]) (*p*[*a* := (*aaa*, *ba*)]) =
      heap-list-all-nth (*arl-assn R*) (*remove1 a* [*0*..<*length p*]) *b p*›
    for *p* :: ‹(′*b array* × *nat*) *list*› **and** *l′* :: ‹′*b list*›
  proof −
    show *?thesis*
      by (*rule heap-list-all-nth-cong*) *auto*
  qed

  show *?thesis*
    using *assms*
    unfolding *arrayO-except-assn-def*
    apply (*subst* (*2*) *arl-assn-def*)
    apply (*subst is-array-list-def*[*abs-def*])
    apply (*subst hr-comp-def*[*abs-def*])
    apply (*subst array-assn-def*)
    apply (*subst is-array-def*[*abs-def*])
    apply (*subst hr-comp-def*[*abs-def*])

apply *sep-auto*
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of a*])
apply (*solves simp*)
unfolding *arrayO-except-assn-def array-assn-def is-array-def*
apply (*subst* (*3*) *arl-assn-def*)
apply (*subst is-array-list-def*[*abs-def*])
apply (*subst* (*2*) *hr-comp-def*[*abs-def*])
apply (*subst ex-assn-move-out*)+
apply (*rule-tac x*=‹*p*[*a* := (*aaa*, *ba*)]› **in** *ent-ex-postI*)
apply (*rule-tac x*=‹*take ba l*′› **in** *ent-ex-postI*)
by (*sep-auto simp*:)
qed

lemma *shorten-take-aa-hnr*[*sepref-fr-rules*]:
‹(*uncurry2 shorten-take-aa*, *uncurry2* (*RETURN ooo shorten-take-ll*)) ∈
$[\lambda((L, j), W).\ j \le length\ (W\ !\ L) \wedge L < length\ W]_a$
$nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d \rightarrow arrayO\text{-}assn\ (arl\text{-}assn\ R)$›
unfolding *shorten-take-aa-def shorten-take-ll-def*
by *sepref-to-hoare sep-auto*

end
theory *Array-List-Array*
imports *Array-Array-List*
begin

0.1.4 Array of Array Lists

There is a major difference compared to ′*a array-list array*: ′*a array-list* is not of sort default. This means that function like *arl-append* cannot be used here.

type-synonym ′*a arrayO-raa* = ‹′*a array array-list*›
type-synonym ′*a list-rll* = ‹′*a list list*›

definition *arlO-assn* :: ‹(′*a* ⇒ ′*b*::*heap* ⇒ *assn*) ⇒ ′*a list* ⇒ ′*b array-list* ⇒ *assn*› **where**
‹*arlO-assn R*′ *xs axs* ≡ $\exists_A$*p*. *arl-assn id-assn p axs* ∗ *heap-list-all R*′ *xs p*›

definition *arlO-assn-except* :: ‹(′*a* ⇒ ′*b*::*heap* ⇒ *assn*) ⇒ *nat list* ⇒ ′*a list* ⇒ ′*b array-list* ⇒ - ⇒ *assn*›
where
‹*arlO-assn-except R*′ *is xs axs f* ≡
$\exists_A$ *p*. *arl-assn id-assn p axs* ∗ *heap-list-all-nth R*′ (*fold remove1 is* [*0*..<*length xs*]) *xs p* ∗
↑ (*length xs* = *length p*) ∗ *f p*›

lemma *arlO-assn-except-array0*: ‹*arlO-assn-except R* [] *xs asx* (λ-. *emp*) = *arlO-assn R xs asx*›
proof −
have ‹(*h* ⊨ *arl-assn id-assn p asx* ∗ *heap-list-all-nth R* [*0*..<*length xs*] *xs p* ∧ *length xs* = *length p*) =
(*h* ⊨ *arl-assn id-assn p asx* ∗ *heap-list-all R xs p*)› (**is** ‹*?a* = *?b*›) **for** *h p*
proof (*rule iffI*)
assume *?a*
then show *?b*
by (*auto simp*: *heap-list-all-heap-list-all-nth*)
next
assume *?b*
then have ‹*length xs* = *length p*›
by (*auto simp*: *heap-list-add-same-length mod-star-conv*)
then show *?a*
using ‹*?b*›

```
      by (auto simp: heap-list-all-heap-list-all-nth)
    qed
  then show ?thesis
    unfolding arlO-assn-except-def arlO-assn-def by (auto simp: ex-assn-def)
qed

lemma arlO-assn-except-array0-index:
  ‹i < length xs ⟹ arlO-assn-except R [i] xs asx (λp. R (xs ! i) (p ! i)) = arlO-assn R xs asx›
  unfolding arlO-assn-except-array0[symmetric] arlO-assn-except-def
  using heap-list-all-nth-remove1[of i ‹[0..<length xs]› R xs] by (auto simp: star-aci(2,3))

lemma arrayO-raa-nth-rule[sep-heap-rules]:
  assumes i: ‹i < length a›
  shows ‹ <arlO-assn (array-assn R) a ai> arl-get ai i <λr. arlO-assn-except (array-assn R) [i] a ai
   (λr'. array-assn R (a ! i) r ∗ ↑(r = r' ! i))>›
proof −
  obtain t n where ai: ‹ai = (t, n)› by (cases ai)
  have i-le: ‹i < Array.length h t› if ‹(h, as) ⊨ arlO-assn (array-assn R) a ai› for h as
    using ai that i unfolding arlO-assn-def array-assn-def is-array-def arl-assn-def is-array-list-def
    by (auto simp: run.simps tap-def arlO-assn-def
        mod-star-conv array-assn-def is-array-def
        Abs-assn-inverse heap-list-add-same-length length-def snga-assn-def
        dest: heap-list-add-same-length)
  show ?thesis
    unfolding hoare-triple-def Let-def
  proof (clarify, intro allI impI conjI)
    fix h as σ r
    assume
      a: ‹(h, as) ⊨ arlO-assn (array-assn R) a ai› and
      r: ‹run (arl-get ai i) (Some h) σ r›
    have [simp]: ‹length a = n›
      using a ai
      by (auto simp: arlO-assn-def mod-star-conv arl-assn-def is-array-list-def
          dest: heap-list-add-same-length)
    obtain p where
      p: ‹(h, as) ⊨ arl-assn id-assn p (t, n) ∗
          heap-list-all-nth (array-assn R) (remove1 i [0..<length p]) a p ∗
          array-assn R (a ! i) (p ! i)›
      using assms a ai
      by (auto simp: hoare-triple-def Let-def execute-simps relH-def in-range.simps
          arlO-assn-except-array0-index[of i, symmetric] arl-get-def
          arlO-assn-except-array0-index arlO-assn-except-def
          elim!: run-elims
          intro!: norm-pre-ex-rule)
    then have ‹(Array.get h t ! i) = p ! i›
      using ai i i-le unfolding arlO-assn-except-array0-index
      apply (auto simp: mod-star-conv array-assn-def is-array-def snga-assn-def
          Abs-assn-inverse arl-assn-def)
      unfolding is-array-list-def is-array-def hr-comp-def list-rel-def
      apply (auto simp: mod-star-conv array-assn-def is-array-def snga-assn-def
          Abs-assn-inverse arl-assn-def from-nat-def
          intro!: nth-take[symmetric])
      done
    moreover have ‹length p = n›
      using p ai by (auto simp: arl-assn-def is-array-list-def)
```

ultimately show ‹(*the-state* σ, *new-addrs h as* (*the-state* σ)) $\models$
arlO-assn-except (*array-assn R*) [*i*] *a ai* ($\lambda r'$. *array-assn R* (*a* ! *i*) *r* $*$ $\uparrow$ ($r = r'$! *i*))›
using *assms ai i-le r p*
by (*fastforce simp*: *hoare-triple-def Let-def execute-simps relH-def in-range.simps*
arlO-assn-except-array0-index[*of i*, *symmetric*] *arl-get-def*
arlO-assn-except-array0-index arlO-assn-except-def
elim!: *run-elims*
intro!: *norm-pre-ex-rule*)
qed ((*solves* ‹*use assms ai i-le in* ‹*auto simp*: *hoare-triple-def Let-def execute-simps relH-def*
in-range.simps arlO-assn-except-array0-index[*of i*, *symmetric*] *arl-get-def*
elim!: *run-elims*
intro!: *norm-pre-ex-rule*››)+)[*3*]
qed

definition *length-ra* :: ‹'*a*::*heap arrayO-raa* ⇒ *nat Heap*› **where**
‹*length-ra xs* = *arl-length xs*›

lemma *length-ra-rule*[*sep-heap-rules*]:
‹<*arlO-assn R x xi*> *length-ra xi* <λr. *arlO-assn R x xi* $*$ $\uparrow$(r = *length x*)>$_t$›
by (*sep-auto simp*: *arlO-assn-def length-ra-def mod-star-conv arl-assn-def*
dest: *heap-list-add-same-length*)

lemma *length-ra-hnr*[*sepref-fr-rules*]:
‹(*length-ra*, *RETURN o op-list-length*) ∈ (*arlO-assn R*)^k $\rightarrow_a$ *nat-assn*›
by *sepref-to-hoare sep-auto*

definition *length-rll* :: ‹'*a list-rll* ⇒ *nat* ⇒ *nat*› **where**
‹*length-rll l i* = *length* (*l*!*i*)›

lemma *le-length-rll-nemptyD*: ‹*b* < *length-rll a ba* ⟹ *a* ! *ba* ≠ []›
by (*auto simp*: *length-rll-def*)

definition *length-raa* :: ‹'*a*::*heap arrayO-raa* ⇒ *nat* ⇒ *nat Heap*› **where**
‹*length-raa xs i* = *do* {
x ← *arl-get xs i*;
Array.len x}›

lemma *length-raa-rule*[*sep-heap-rules*]:
‹*b* < *length xs* ⟹ <*arlO-assn* (*array-assn R*) *xs a*> *length-raa a b*
<λr. *arlO-assn* (*array-assn R*) *xs a* $*$ $\uparrow$ (r = *length-rll xs b*)>$_t$›
unfolding *length-raa-def*
apply (*cases a*)
apply *sep-auto*
apply (*sep-auto simp*: *arlO-assn-except-def arl-length-def array-assn-def*
eq-commute[*of* ‹(-, -)›] *is-array-def hr-comp-def length-rll-def*
dest: *list-all2-lengthD*)
apply (*sep-auto simp*: *arlO-assn-except-def arl-length-def arl-assn-def*
eq-commute[*of* ‹(-, -)›] *is-array-list-def hr-comp-def length-rll-def list-rel-def*
dest: *list-all2-lengthD*)[]
unfolding *arlO-assn-def*[*symmetric*] *arl-assn-def*[*symmetric*]
apply (*subst arlO-assn-except-array0-index*[*symmetric*, *of b*])
apply *simp*
unfolding *arlO-assn-except-def arl-assn-def hr-comp-def is-array-def*
apply *sep-auto*
done

lemma *length-raa-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-raa*, *uncurry* (*RETURN* ∘∘ *length-rll*)) ∈
  [λ(*xs*, *i*). *i* < *length xs*]$_a$ (*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} → *nat-assn*›
 by *sepref-to-hoare sep-auto*

definition *nth-raa* :: ‹′*a*::*heap arrayO-raa* ⇒ *nat* ⇒ *nat* ⇒ ′*a Heap*› **where**
 ‹*nth-raa xs i j* = *do* {
   x ← *arl-get xs i*;
   y ← *Array.nth x j*;
   return y}›

lemma *nth-raa-hnr*[*sepref-fr-rules*]:
 assumes *p*: ‹*is-pure R*›
 shows
  ‹(*uncurry2 nth-raa*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
   [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
   (*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} → *R*›
proof −
 obtain *R′* **where** *R*: ‹*the-pure R* = *R′*› **and** *R′*: ‹*R* = *pure R′*›
  using *p* **by** *fastforce*
 have *H*: ‹*list-all2* (λ*x x′*. (*x*, *x′*) ∈ *the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *R′*))) *bc* (*a* ! *ba*) ⟹
   b < *length* (*a* ! *ba*) ⟹
   (*bc* ! *b*, *a* ! *ba* ! *b*) ∈ *R′*› **for** *bc a ba b*
  by (*auto simp add*: *ent-refl-true list-all2-conv-all-nth is-pure-alt-def pure-app-eq*[*symmetric*])
 show *?thesis*
  supply *nth-rule*[*sep-heap-rules*]
  apply *sepref-to-hoare*
  apply (*subst* (*2*) *arlO-assn-except-array0-index*[*symmetric*])
  apply (*solves* ‹*auto*›)[]
  apply (*sep-auto simp*: *nth-raa-def nth-rll-def length-rll-def*)
  apply (*sep-auto simp*: *arlO-assn-except-def arlO-assn-def arl-assn-def hr-comp-def list-rel-def*
   list-all2-lengthD array-assn-def is-array-def hr-comp-def[*abs-def*]
   star-aci(*3*) *R R′ pure-def H*)
  done
qed

definition *update-raa* :: (′*a*::{*heap*,*default*}) *arrayO-raa* ⇒ *nat* ⇒ *nat* ⇒ ′*a* ⇒ ′*a arrayO-raa Heap*
where
 ‹*update-raa a i j y* = *do* {
   x ← *arl-get a i*;
   a′ ← *Array.upd j y x*;
   arl-set a i a′
  }› — is the Array.upd really needed?

definition *update-rll* :: ′*a list-rll* ⇒ *nat* ⇒ *nat* ⇒ ′*a* ⇒ ′*a list list* **where**
 ‹*update-rll xs i j y* = *xs*[*i*:= (*xs* ! *i*)[*j* := *y*]]›

declare *nth-rule*[*sep-heap-rules del*]
declare *arrayO-raa-nth-rule*[*sep-heap-rules*]

TODO: is it possible to be more precise and not drop the ↑ ((*aa*, *bc*) = *r′* ! *bb*)

lemma *arlO-assn-except-arl-set*[*sep-heap-rules*]:
 fixes *R* :: ‹′*a* ⇒ ′*b* :: {*heap*} ⇒ *assn*›
 assumes *p*: ‹*is-pure R*› **and** ‹*bb* < *length a*› **and**
  ‹*ba* < *length-rll a bb*›
 shows ‹
   <*arlO-assn-except* (*array-assn R*) [*bb*] *a ai* (λ*r′*. *array-assn R* (*a* ! *bb*) *aa* ∗

$\uparrow (aa = r' ! bb)) * R\ b\ bi$>
Array.upd ba bi aa
<$\lambda aa.$ *arlO-assn-except* (*array-assn R*) [*bb*] *a ai*
($\lambda r'$. *array-assn R* ((*a* ! *bb*)[*ba* := *b*]) *aa*) $*$ *R b bi* $*$ *true*>›
proof −
obtain R' **where** *R*: ‹*the-pure R* = R'› **and** R': ‹*R* = *pure* R'›
using *p* **by** *fastforce*
show *?thesis*
using *assms*
by (*cases ai*)
(*sep-auto simp*: *arlO-assn-except-def arl-assn-def hr-comp-def list-rel-imp-same-length list-rel-update length-rll-def array-assn-def is-array-def*)
qed

lemma *update-raa-rule*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› **and** ‹*bb* < *length a*› **and** ‹*ba* < *length-rll a bb*›
shows ‹<*R b bi* $*$ *arlO-assn* (*array-assn R*) *a ai*> *update-raa ai bb ba bi*
<λr. *R b bi* $*$ ($\exists_A x$. *arlO-assn* (*array-assn R*) *x r* $* \uparrow$ (*x* = *update-rll a bb ba b*))>$_t$›
using *assms*
apply (*sep-auto simp add*: *update-raa-def update-rll-def p*)
apply (*sep-auto simp add*: *update-raa-def arlO-assn-except-def array-assn-def is-array-def hr-comp-def arl-assn-def*)
apply (*subst-tac i*=*bb* **in** *arlO-assn-except-array0-index*[*symmetric*])
apply (*solves* ‹*simp*›)
apply (*subst arlO-assn-except-def*)
apply (*auto simp add*: *update-raa-def arlO-assn-except-def array-assn-def is-array-def hr-comp-def*)

apply (*rule-tac x*=‹*p*[*bb* := *xa*]› **in** *ent-ex-postI*)
apply (*rule-tac x*=‹*bc*› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)
apply (*solves* ‹*auto*›)
by (*sep-auto simp*: *arl-assn-def*)

lemma *update-raa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-raa*, *uncurry3* (*RETURN oooo update-rll*)) $\in$
$[\lambda(((l,i), j), x).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$ $(arlO\text{-}assn\ (array\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k *_a R^k \rightarrow (arlO\text{-}assn\ (array\text{-}assn\ R))$›
by *sepref-to-hoare* (*sep-auto simp*: *assms*)

definition *swap-aa* :: ('*a*::{*heap*,*default*}) *arrayO-raa* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*a arrayO-raa Heap*
where
‹*swap-aa xs k i j* = *do* {
xi ← *nth-raa xs k i*;
xj ← *nth-raa xs k j*;
xs ← *update-raa xs k i xj*;
xs ← *update-raa xs k j xi*;
return xs
}›

definition *swap-ll* **where**
‹*swap-ll xs k i j* = *list-update xs k* (*swap* (*xs*!*k*) *i j*)›

lemma *nth-raa-heap*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› **and** ‹*b* < *length aa*› **and** ‹*ba* < *length-rll aa b*›

```
  shows ‹
   <arlO-assn (array-assn R) aa a>
   nth-raa a b ba
   <λr. ∃_A x. arlO-assn (array-assn R) aa a ∗
              (R x r ∗
               ↑ (x = nth-rll aa b ba)) ∗
              true>›
proof –
  have ‹<arlO-assn (array-assn R) aa a ∗
        nat-assn b b ∗
        nat-assn ba ba>
       nth-raa a b ba
       <λr. ∃_A x. arlO-assn (array-assn R) aa a ∗
                  nat-assn b b ∗
                  nat-assn ba ba ∗
                  R x r ∗
                  true ∗
                  ↑ (x = nth-rll aa b ba)>›
    using p assms nth-raa-hnr[of R] unfolding hfref-def hn-refine-def
    by (cases a) auto
  then show ?thesis
    unfolding hoare-triple-def
    by (auto simp: Let-def pure-def)
qed

lemma update-raa-rule-pure:
  assumes p: ‹is-pure R› and ‹b < length aa› and ‹ba < length-rll aa b› and
    b: ‹(bb, be) ∈ the-pure R›
  shows ‹
   <arlO-assn (array-assn R) aa a>
          update-raa a b ba bb
          <λr. ∃_A x. invalid-assn (arlO-assn (array-assn R)) aa a ∗ arlO-assn (array-assn R) x r ∗
                      true ∗
                      ↑ (x = update-rll aa b ba be)>›
proof –
  obtain R′ where R′: ‹R′ = the-pure R› and RR′: ‹R = pure R′›
    using p by fastforce
  have bb: ‹pure R′ be bb = ↑((bb, be) ∈ R′)›
    by (auto simp: pure-def)
  have ‹ <arlO-assn (array-assn R) aa a ∗ nat-assn b b ∗ nat-assn ba ba ∗ R be bb>
          update-raa a b ba bb
          <λr. ∃_A x. invalid-assn (arlO-assn (array-assn R)) aa a ∗ nat-assn b b ∗ nat-assn ba ba ∗
                      R be bb ∗
                      arlO-assn (array-assn R) x r ∗
                      true ∗
                      ↑ (x = update-rll aa b ba be)>›
    using p assms update-raa-hnr[of R] unfolding hfref-def hn-refine-def
    by (cases a) auto
  then show ?thesis
    using b unfolding R′[symmetric] unfolding hoare-triple-def RR′ bb
    by (auto simp: Let-def pure-def)
qed

lemma length-update-rll[simp]: ‹length (update-rll a bb b c) = length a›
  unfolding update-rll-def by auto
```

lemma *length-rll-update-rll*:
‹$bb < length\ a \Longrightarrow length\text{-}rll\ (update\text{-}rll\ a\ bb\ b\ c)\ bb = length\text{-}rll\ a\ bb$›
unfolding *length-rll-def update-rll-def* **by** *auto*

lemma *swap-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹$(uncurry3\ swap\text{-}aa,\ uncurry3\ (RETURN\ oooo\ swap\text{-}ll)) \in$
$[\lambda(((xs,\ k),\ i),\ j).\ k < length\ xs \wedge i < length\text{-}rll\ xs\ k \wedge j < length\text{-}rll\ xs\ k]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow (arlO\text{-}assn\ (array\text{-}assn\ R))$›
proof −
note *update-raa-rule-pure*[*sep-heap-rules*]
obtain R' **where** R': ‹$R' = the\text{-}pure\ R$› **and** RR': ‹$R = pure\ R'$›
using *assms* **by** *fastforce*
have [*simp*]: ‹$the\text{-}pure\ (\lambda a\ b.\ \uparrow ((b,\ a) \in R')) = R'$›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
using *assms* **unfolding** R'[*symmetric*] **unfolding** RR'
apply *sepref-to-hoare*
apply (*sep-auto simp*: *swap-aa-def swap-ll-def arlO-assn-except-def*
length-rll-update-rll)
by (*sep-auto simp*: *update-rll-def swap-def nth-rll-def list-update-swap*)
qed

definition *update-ra* :: ‹$'a\ arrayO\text{-}raa \Rightarrow nat \Rightarrow 'a\ array \Rightarrow 'a\ arrayO\text{-}raa\ Heap$› **where**
‹*update-ra xs n x = arl-set xs n x*›

lemma *update-ra-list-update-rules*[*sep-heap-rules*]:
assumes ‹$n < length\ l$›
shows ‹$<R\ y\ x * arlO\text{-}assn\ R\ l\ xs>\ update\text{-}ra\ xs\ n\ x\ <arlO\text{-}assn\ R\ (l[n{:=}y])>_t$›
proof −
have H: ‹$heap\text{-}list\text{-}all\ R\ l\ p = heap\text{-}list\text{-}all\ R\ l\ p * \uparrow (n < length\ p)$› **for** p
using *assms* **by** (*simp add*: *ent-iffI heap-list-add-same-length*)
have [*simp*]: ‹$heap\text{-}list\text{-}all\text{-}nth\ R\ (remove1\ n\ [0..<length\ p])\ (l[n := y])\ (p[n := x]) =$
$heap\text{-}list\text{-}all\text{-}nth\ R\ (remove1\ n\ [0..<length\ p])\ (l)\ (p)$› **for** p
by (*rule heap-list-all-nth-cong*) *auto*
show *?thesis*
using *assms*
apply (*cases xs*)
supply *arl-set-rule*[*sep-heap-rules del*]
apply (*sep-auto simp*: *arlO-assn-def update-ra-def Let-def arl-assn-def*
dest!: *heap-list-add-same-length*
elim!: *run-elims*)
apply (*subst H*)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst heap-list-all-nth-remove1* [**where** $i = n$])
apply (*solves* ‹*simp*›)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst* (*2*) *heap-list-all-nth-remove1* [**where** $i = n$])
apply (*solves* ‹*simp*›)
supply *arl-set-rule*[*sep-heap-rules*]
apply (*sep-auto* (*plain*))
apply (*subgoal-tac* ‹$length\ (l[n := y]) = length\ (p[n := x])$›)
apply *assumption*
apply *auto*[]
apply *sep-auto*

done
qed
lemma *ex-assn-up-eq*: ‹$(\exists_A x.\ P\ x * \uparrow(x = a) * Q) = (P\ a * Q)$›
by (*smt ex-one-point-gen mod-pure-star-dist mod-starE mult.right-neutral pure-true*)
lemma *update-ra-list-update*[*sepref-fr-rules*]:
‹(*uncurry2 update-ra*, *uncurry2* (*RETURN ooo list-update*)) ∈
$[\lambda((xs,\ n),\ \text{-}).\ n < \mathit{length}\ xs]_a\ (\mathit{arlO\text{-}assn}\ R)^d *_a \mathit{nat\text{-}assn}^k *_a R^d \rightarrow (\mathit{arlO\text{-}assn}\ R)$›
proof −
have [*simp*]: ‹$(\exists_A x.\ \mathit{arlO\text{-}assn}\ R\ x\ r * \mathit{true} * \uparrow (x = \mathit{list\text{-}update}\ a\ ba\ b)) =$
$\mathit{arlO\text{-}assn}\ R\ (a[ba := b])\ r * \mathit{true}$›
for *a ba b r*
apply (*subst assn-aci*(*10*))
apply (*subst ex-assn-up-eq*)
..
show *?thesis*
by *sepref-to-hoare sep-auto*
qed
term *arl-append*
definition *arrayO-raa-append* **where**
arrayO-raa-append ≡ λ(*a*,*n*) *x*. *do* {
len ← *Array.len a*;
if n<*len then do* {
a ← *Array.upd n x a*;
return (*a*,*n*+*1*)
} *else do* {
let newcap = *2* ∗ *len*;
default ← *Array.new 0 default*;
a ← *array-grow a newcap default*;
a ← *Array.upd n x a*;
return (*a*,*n*+*1*)
}
}

lemma *heap-list-all-append-Nil*:
‹$y \neq [] \Longrightarrow \mathit{heap\text{-}list\text{-}all}\ R\ (va\ @\ y)\ [] = \mathit{false}$›
by (*cases va*; *cases y*) *auto*

lemma *heap-list-all-Nil-append*:
‹$y \neq [] \Longrightarrow \mathit{heap\text{-}list\text{-}all}\ R\ []\ (va\ @\ y) = \mathit{false}$›
by (*cases va*; *cases y*) *auto*

lemma *heap-list-all-append*: ‹$\mathit{heap\text{-}list\text{-}all}\ R\ (l\ @\ [y])\ (l'\ @\ [x])$
$= \mathit{heap\text{-}list\text{-}all}\ R\ (l)\ (l') * R\ y\ x$›
by (*induction R l l′ rule*: *heap-list-all.induct*)
(*auto simp*: *ac-simps heap-list-all-Nil-append heap-list-all-append-Nil*)
term *arrayO-raa*
lemma *arrayO-raa-append-rule*[*sep-heap-rules*]:
‹<*arlO-assn R l a* ∗ *R y x*> *arrayO-raa-append a x* <λ*a*. *arlO-assn R* (*l*@[*y*]) *a* >$_t$›
proof −
have *1*: ‹$\mathit{arl\text{-}assn}\ \mathit{id\text{-}assn}\ p\ a * \mathit{heap\text{-}list\text{-}all}\ R\ l\ p =$
$\mathit{arl\text{-}assn}\ \mathit{id\text{-}assn}\ p\ a * \mathit{heap\text{-}list\text{-}all}\ R\ l\ p * \uparrow (\mathit{length}\ l = \mathit{length}\ p)$› **for** *p*
by (*smt ent-iffI ent-pure-post-iff entailsI heap-list-add-same-length mult.right-neutral*
pure-false pure-true star-false-right)

show *?thesis*
unfolding *arrayO-raa-append-def arrayO-raa-append-def arlO-assn-def*

length-ra-def arl-length-def hr-comp-def
apply (*subst 1*)
unfolding *arl-assn-def is-array-list-def hr-comp-def*
apply (*cases a*)
apply *sep-auto*
apply (*rule-tac psi*=‹*Suc* (*length l*) ≤ *length* (*l'*[*length l* := *x*])› **in** *asm-rl*)
apply *simp*
apply *simp*
apply (*sep-auto simp*: *take-update-last heap-list-all-append*)
apply (*sep-auto* (*plain*))
apply *sep-auto*
apply (*sep-auto* (*plain*))
apply *sep-auto*
apply (*sep-auto* (*plain*))
apply *sep-auto*
apply (*rule-tac psi* = ‹*Suc* (*length p*) ≤ *length* ((*p* @ *replicate* (*length p*) *xa*)[*length p* := *x*])›
in *asm-rl*)
apply *sep-auto*
apply *sep-auto*
apply (*sep-auto simp*: *heap-list-all-append*)
done
qed

lemma *arrayO-raa-append-op-list-append*[*sepref-fr-rules*]:
‹(*uncurry arrayO-raa-append*, *uncurry* (*RETURN oo op-list-append*)) ∈
(*arlO-assn R*)^d $*_a$ R^d $\rightarrow_a$ *arlO-assn R*›
apply *sepref-to-hoare*
apply (*subst mult.commute*)
apply (*subst mult.assoc*)
by (*sep-auto simp*: *ex-assn-up-eq*)

definition *array-of-arl* :: ‹*'a list* ⇒ *'a list*› **where**
‹*array-of-arl xs* = *xs*›

definition *array-of-arl-raa* :: *'a::heap array-list* ⇒ *'a array Heap* **where**
‹*array-of-arl-raa* = (λ(*a*, *n*). *array-shrink a n*)›

lemma *array-of-arl*[*sepref-fr-rules*]:
‹(*array-of-arl-raa*, *RETURN o array-of-arl*) ∈ (*arl-assn R*)^d $\rightarrow_a$ (*array-assn R*)›
by *sepref-to-hoare*
(*sep-auto simp*: *array-of-arl-raa-def arl-assn-def is-array-list-def hr-comp-def*
array-assn-def is-array-def array-of-arl-def)

definition *arrayO-raa-empty* ≡ *do* {
a ← *Array.new initial-capacity default*;
return (*a*,*0*)
}

lemma *arrayO-raa-empty-rule*[*sep-heap-rules*]: < *emp* > *arrayO-raa-empty* <λ*r*. *arlO-assn R* [] *r*>
by (*sep-auto simp*: *arrayO-raa-empty-def is-array-list-def initial-capacity-def*
arlO-assn-def arl-assn-def)

definition *arrayO-raa-empty-sz* **where**
arrayO-raa-empty-sz init-cap ≡ *do* {
default ← *Array.new 0 default*;
a ← *Array.new* (*max init-cap minimum-capacity*) *default*;

return (*a*,*0*)
}

lemma *arl-empty-sz-array-rule*[*sep-heap-rules*]: < *emp* > *arrayO-raa-empty-sz N* <λ*r*. *arlO-assn R* [] *r*>$_t$
proof −
have [*simp*]: ‹(*xa* $\mapsto_a$ *replicate* (*max N 16*) *x*) $*$ *x* $\mapsto_a$ [] = (*xa* $\mapsto_a$ (*x* # *replicate* (*max N 16* − *1*) *x*)) $*$ *x* $\mapsto_a$ []›
for *xa x*
by (*cases N*) (*sep-auto simp*: *arrayO-raa-empty-sz-def is-array-list-def minimum-capacity-def max-def*)+
show *?thesis*
by (*sep-auto simp*: *arrayO-raa-empty-sz-def is-array-list-def minimum-capacity-def arlO-assn-def arl-assn-def*)
qed

definition *nth-rl* :: ‹'*a*::*heap arrayO-raa* ⇒ *nat* ⇒ '*a array Heap*› **where**
‹*nth-rl xs n* = *do* {*x* ← *arl-get xs n*; *array-copy x*}›

lemma *nth-rl-op-list-get*:
‹(*uncurry nth-rl*, *uncurry* (*RETURN oo op-list-get*)) ∈
[λ(*xs*, *n*). *n* < *length xs*]$_a$ (*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} → *array-assn R*›
apply *sepref-to-hoare*
unfolding *arlO-assn-def heap-list-all-heap-list-all-nth-eq*
apply (*subst-tac i*=*b* **in** *heap-list-all-nth-remove1*)
apply (*solves* ‹*simp*›)
apply (*subst-tac* (*2*) *i*=*b* **in** *heap-list-all-nth-remove1*)
apply (*solves* ‹*simp*›)
by (*sep-auto simp*: *nth-rl-def arlO-assn-def heap-list-all-heap-list-all-nth-eq array-assn-def hr-comp-def*[*abs-def*] *is-array-def arl-assn-def*)

definition *arl-of-array* :: '*a list list* ⇒ '*a list list* **where**
‹*arl-of-array xs* = *xs*›

definition *arl-of-array-raa* :: '*a*::*heap array* ⇒ ('*a array-list*) *Heap* **where**
‹*arl-of-array-raa xs* = *do* {
n ← *Array.len xs*;
return (*xs*, *n*)
}›

lemma *arl-of-array-raa*: ‹(*arl-of-array-raa*, *RETURN o arl-of-array*) ∈
[λ*xs*. *xs* ≠ []]$_a$ (*array-assn R*)^d → (*arl-assn R*)›
by *sepref-to-hoare* (*sep-auto simp*: *arl-of-array-raa-def arl-assn-def is-array-list-def hr-comp-def array-assn-def is-array-def arl-of-array-def*)

end
theory *WB-Word*
imports *HOL*−*Word.Word Native-Word.Uint64 Native-Word.Uint32 WB-More-Refinement HOL*−*Imperative-HOL.He*
Collections.HashCode Bits-Natural
begin

lemma *less-upper-bintrunc-id*: ‹*n* < *2* ^*b* ⟹ *n* ≥ *0* ⟹ *bintrunc b n* = *n*›
unfolding *uint32-of-nat-def*
by (*simp add*: *no-bintr-alt1*)

definition *word-nat-rel* :: ('*a* :: *len0 Word.word* × *nat*) *set* **where**
‹*word-nat-rel* = *br unat* (λ-. *True*)›

lemma *bintrunc-eq-bits-eqI*: ‹ (⋀*n*. (*n* < *r* ∧ *bin-nth c n*) = (*n* < *r* ∧ *bin-nth a n*)) ⟹
bintrunc r (*a*) = *bintrunc r c*›
proof (*induction r arbitrary*: *a c*)
case *0*
then show *?case* **by** (*simp-all flip*: *bin-nth.Z*)
next
case (*Suc r a c*) **note** *IH* = *this*(*1*) **and** *eq* = *this*(*2*)
have *1*: ‹(*n* < *r* ∧ *bin-nth* (*bin-rest a*) *n*) = (*n* < *r* ∧ *bin-nth* (*bin-rest c*) *n*)› **for** *n*
using *eq*[*of* ‹*Suc n*›] *eq*[*of 1*] **by** (*clarsimp simp flip*: *bin-nth.Z*)
show *?case*
using *IH*[*OF 1*] *eq*[*of 0*] **by** (*simp-all flip*: *bin-nth.Z*)
qed

lemma *and-eq-bits-eqI*: ‹(⋀*n*. *c* !! *n* = (*a* !! *n* ∧ *b* !! *n*))⟹ *a AND b* = *c*› **for** *a b c* :: ‹*- word*›
by *transfer*
(*rule bintrunc-eq-bits-eqI*, *auto simp add*: *bin-nth-ops*)

lemma *pow2-mono-word-less*:
‹*m* < *LENGTH*(*'a*) ⟹ *n* < *LENGTH*(*'a*) ⟹ *m* < *n* ⟹ (*2* :: *'a* :: *len word*) ^*m* < *2* ^ *n*›
proof (*induction n arbitrary*: *m*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n m*) **note** *IH* = *this*(*1*) **and** *le* = *this*(*2*−)
have [*simp*]: ‹*nat* (*bintrunc LENGTH*(*'a*) (*2*::*int*)) = *2*›
by (*metis add-lessD1 le*(*2*) *plus-1-eq-Suc power-one-right uint-bintrunc unat-def unat-p2*)
have *1*: ‹*unat* ((*2* :: *'a word*) ^ *n*) ≤ (*2* :: *nat*) ^ *n*›
by (*metis Suc.prems*(*2*) *eq-imp-le le-SucI linorder-not-less unat-p2*)
have *2*: ‹*unat* ((*2* :: *'a word*)) ≤ (*2* :: *nat*)›
by (*metis le-unat-uoi nat-le-linear of-nat-numeral*)
have ‹*unat* (*2* :: *'a word*) ∗ *unat* ((*2* :: *'a word*) ^ *n*) ≤ (*2* :: *nat*) ^ *Suc n*›
using *mult-le-mono*[*OF 2 1*] **by** *auto*
also have ‹(*2* :: *nat*) ^ *Suc n* < (*2* :: *nat*) ^ *LENGTH*(*'a*)›
using *le*(*2*) **by** (*metis unat-lt2p unat-p2*)
finally have ‹*unat* (*2* :: *'a word*) ∗ *unat* ((*2* :: *'a word*) ^ *n*) < *2* ^ *LENGTH*(*'a*)›
.
then have [*simp*]: ‹*unat* (*2* ∗ (*2* :: *'a word*) ^ *n*) = *unat* (*2* :: *'a word*) ∗ *unat* ((*2* :: *'a word*) ^ *n*)›
using *unat-mult-lem*[*of* ‹*2* :: *'a word*› ‹(*2* :: *'a word*) ^ *n*›]
by *auto*
have [*simp*]: ‹(*0*::*nat*) < *unat* ((*2*::*'a word*) ^ *n*)›
by (*simp add*: *Suc-lessD le*(*2*) *unat-p2*)

show *?case*
using *IH*(*1*)[*of m*] *le*(*2*−)
by (*auto simp*: *less-Suc-eq word-less-nat-alt*
simp del: *unat-lt2p*)
qed

lemma *pow2-mono-word-le*:
‹*m* < *LENGTH*(*'a*) ⟹ *n* < *LENGTH*(*'a*) ⟹ *m* ≤ *n* ⟹ (*2* :: *'a* :: *len word*) ^*m* ≤ *2* ^ *n*›
using *pow2-mono-word-less*[*of m n*, **where** *'a* = *'a*]
by (*cases* ‹*m* = *n*›) *auto*

definition *uint32-max* :: *nat* **where**
⟨*uint32-max* = *2* ^*32* − *1*⟩

lemma *unat-le-uint32-max-no-bit-set*:
fixes *n* :: ⟨*'a*::*len word*⟩
assumes *less*: ⟨*unat n* ≤ *uint32-max*⟩ **and**
n: ⟨*n* !! *na*⟩ **and**
32: ⟨*32* < *LENGTH*(*'a*)⟩
shows ⟨*na* < *32*⟩
proof (*rule ccontr*)
assume *H*: ⟨¬ *?thesis*⟩
have *na-le*: ⟨*na* < *LENGTH*(*'a*)⟩
using *test-bit-bin*[*THEN iffD1*, *OF n*]
by *auto*
have ⟨(*2* :: *nat*) ^ *32* < (*2* :: *nat*) ^ *LENGTH*(*'a*)⟩
using *32 power-strict-increasing-iff rel-simps*(*49*) *semiring-norm*(*76*) **by** *blast*
then have [*simp*]: ⟨(*4294967296*::*nat*) *mod* (*2*::*nat*) ^ *LENGTH*(*'a*) = (*4294967296*::*nat*)⟩
by (*auto simp*: *word-le-nat-alt unat-numeral uint32-max-def mod-less*
simp del: *unat-bintrunc*)
have ⟨(*2* :: *'a word*) ^ *na* ≥ *2* ^ *32*⟩
using *pow2-mono-word-le*[*OF 32 na-le*] *H* **by** *auto*
also have ⟨*n* ≥ (*2* :: *'a word*) ^ *na*⟩
using *assms*
unfolding *uint32-max-def*
by (*auto dest*!: *bang-is-le*)
finally have ⟨*unat n* > *uint32-max*⟩
supply [[*show-sorts*]]
unfolding *word-le-nat-alt*
by (*auto simp*: *word-le-nat-alt unat-numeral uint32-max-def*
simp del: *unat-bintrunc*)

then show *False*
using *less* **by** *auto*
qed

definition *uint32-max'* **where**
[*simp*, *symmetric*, *code*]: ⟨*uint32-max'* = *uint32-max*⟩

lemma [*code*]: ⟨*uint32-max'* = *4294967295*⟩
by (*auto simp*: *uint32-max-def*)

This lemma is very trivial but maps an *64 word* to its list counterpart. This especially allows to combine two numbers together via ther bit representation (which should be faster than enumerating all numbers).

lemma *ex-rbl-word64*:
⟨∃ *a64 a63 a62 a61 a60 a59 a58 a57 a56 a55 a54 a53 a52 a51 a50 a49 a48 a47 a46 a45 a44 a43 a42 a41*
a40 a39 a38 a37 a36 a35 a34 a33 a32 a31 a30 a29 a28 a27 a26 a25 a24 a23 a22 a21 a20 a19 a18 a17
a16 a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1.
to-bl (*n* :: *64 word*) =
[*a64*, *a63*, *a62*, *a61*, *a60*, *a59*, *a58*, *a57*, *a56*, *a55*, *a54*, *a53*, *a52*, *a51*, *a50*, *a49*, *a48*, *a47*, *a46*, *a45*, *a44*, *a43*, *a42*, *a41*, *a40*, *a39*, *a38*, *a37*, *a36*, *a35*, *a34*, *a33*, *a32*, *a31*, *a30*, *a29*, *a28*, *a27*, *a26*, *a25*, *a24*, *a23*, *a22*, *a21*, *a20*, *a19*, *a18*, *a17*, *a16*, *a15*, *a14*, *a13*, *a12*, *a11*, *a10*, *a9*, *a8*, *a7*, *a6*, *a5*, *a4*, *a3*, *a2*, *a1*]⟩ (**is** *?A*) **and**

ex-rbl-word64-le-uint32-max:
‹*unat n* ≤ *uint32-max* ⟹ ∃ *a31 a30 a29 a28 a27 a26 a25 a24 a23 a22 a21 a20 a19 a18 a17 a16 a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a32*.
to-bl (*n* :: *64 word*) =
[*False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*,
False, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*,
False, *False*, *False*, *False*, *False*, *False*,
a32, *a31*, *a30*, *a29*, *a28*, *a27*, *a26*, *a25*, *a24*, *a23*, *a22*, *a21*, *a20*, *a19*, *a18*, *a17*, *a16*, *a15*,
a14, *a13*, *a12*, *a11*, *a10*, *a9*, *a8*, *a7*, *a6*, *a5*, *a4*, *a3*, *a2*, *a1*]› (**is** ‹- ⟹ *?B*›) **and**
ex-rbl-word64-ge-uint32-max:
‹*n AND* (*2^32* − *1*) = *0* ⟹ ∃ *a64 a63 a62 a61 a60 a59 a58 a57 a56 a55 a54 a53 a52 a51 a50 a49 a48*
a47 a46 a45 a44 a43 a42 a41 a40 a39 a38 a37 a36 a35 a34 a33.
to-bl (*n* :: *64 word*) =
[*a64*, *a63*, *a62*, *a61*, *a60*, *a59*, *a58*, *a57*, *a56*, *a55*, *a54*, *a53*, *a52*, *a51*, *a50*, *a49*, *a48*, *a47*,
a46, *a45*, *a44*, *a43*, *a42*, *a41*, *a40*, *a39*, *a38*, *a37*, *a36*, *a35*, *a34*, *a33*,
False, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*,
False, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*,
False, *False*, *False*, *False*, *False*, *False*]› (**is** ‹- ⟹ *?C*›)
proof −
have [*simp*]: *n* > *0* ⟹ *length xs* = *n* ⟷
(∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n* − *1*) **for** *ys n xs*
by (*cases xs*) *auto*
show *H*: *?A*
using *word-bl-Rep′*[*of n*]
by (*auto simp del*: *word-bl-Rep′*)

show *?B* **if** ‹*unat n* ≤ *uint32-max*›
proof −
have *H′*: ‹*m* ≥ *32* ⟹ ¬*n* !! *m*› **for** *m*
using *unat-le-uint32-max-no-bit-set*[*of n m*, *OF that*] **by** *auto*
show *?thesis* **using** *that H′*[*of 64*] *H′*[*of 63*] *H′*[*of 62*] *H′*[*of 61*] *H′*[*of 60*] *H′*[*of 59*] *H′*[*of 58*]
H′[*of 57*] *H′*[*of 56*] *H′*[*of 55*] *H′*[*of 54*] *H′*[*of 53*] *H′*[*of 52*] *H′*[*of 51*] *H′*[*of 50*] *H′*[*of 49*]
H′[*of 48*] *H′*[*of 47*] *H′*[*of 46*] *H′*[*of 45*] *H′*[*of 44*] *H′*[*of 43*] *H′*[*of 42*] *H′*[*of 41*] *H′*[*of 40*]
H′[*of 39*] *H′*[*of 38*] *H′*[*of 37*] *H′*[*of 36*] *H′*[*of 35*] *H′*[*of 34*] *H′*[*of 33*] *H′*[*of 32*]
H′[*of 31*]
using *H* **unfolding** *unat-def*
by (*clarsimp simp add*: *test-bit-bl word-size*)
qed
show *?C* **if** ‹*n AND* (*2^32* − *1*) = *0*›
proof −
note *H′* = *test-bit-bl*[*of* ‹*n AND* (*2^32* − *1*)› *m* **for** *m*, *unfolded word-size*, *simplified*]
have [*simp*]: ‹(*n AND 4294967295*) !! *m* = *False*› **for** *m*
using *that* **by** *auto*
show *?thesis*
using *H H′*[*of 0*]
H′[*of 32*] *H′*[*of 31*] *H′*[*of 30*] *H′*[*of 29*] *H′*[*of 28*] *H′*[*of 27*] *H′*[*of 26*] *H′*[*of 25*] *H′*[*of 24*]
H′[*of 23*] *H′*[*of 22*] *H′*[*of 21*] *H′*[*of 20*] *H′*[*of 19*] *H′*[*of 18*] *H′*[*of 17*] *H′*[*of 16*] *H′*[*of 15*]
H′[*of 14*] *H′*[*of 13*] *H′*[*of 12*] *H′*[*of 11*] *H′*[*of 10*] *H′*[*of 9*] *H′*[*of 8*] *H′*[*of 7*] *H′*[*of 6*]
H′[*of 5*] *H′*[*of 4*] *H′*[*of 3*] *H′*[*of 2*] *H′*[*of 1*]
unfolding *unat-def word-size that*
by (*clarsimp simp add*: *word-size bl-word-and word-add-rbl*)
qed
qed

32-bits

lemma *word-nat-of-uint32-Rep-inject*[*simp*]: ‹*nat-of-uint32 ai* = *nat-of-uint32 bi* ⟷ *ai* = *bi*›
  by *transfer simp*

lemma *nat-of-uint32-012*[*simp*]: ‹*nat-of-uint32 0* = *0*› ‹*nat-of-uint32 2* = *2*› ‹*nat-of-uint32 1* = *1*›
  by (*transfer*, *auto*)+

lemma *nat-of-uint32-3*: ‹*nat-of-uint32 3* = *3*›
  by (*transfer*, *auto*)+

lemma *nat-of-uint32-Suc03-iff*:
 ‹*nat-of-uint32 a* = *Suc 0* ⟷ *a* = *1*›
   ‹*nat-of-uint32 a* = *3* ⟷ *a* = *3*›
   using *word-nat-of-uint32-Rep-inject nat-of-uint32-3* **by** *fastforce*+

lemma   *nat-of-uint32-013-neq*:
  (*1*::*uint32*) ≠ (*0* :: *uint32*) (*0*::*uint32*) ≠ (*1* :: *uint32*)
  (*3*::*uint32*) ≠ (*0* :: *uint32*)
  (*3*::*uint32*) ≠ (*1* :: *uint32*)
  (*0*::*uint32*) ≠ (*3* :: *uint32*)
  (*1*::*uint32*) ≠ (*3* :: *uint32*)
  by (*auto dest*: *arg-cong*[*of - - nat-of-uint32*] *simp*: *nat-of-uint32-3*)

definition *uint32-nat-rel* :: (*uint32* × *nat*) *set* **where**
  ‹*uint32-nat-rel* = *br nat-of-uint32* (λ-. *True*)›

lemma *unat-shiftr*: ‹*unat* (*xi* >> *n*) = *unat xi div* (*2*^*n*)›
proof −
  have [*simp*]: ‹*nat* (*2* ∗ *2* ^ *n*) = *2* ∗ *2* ^ *n*› **for** *n* :: *nat*
    by (*metis nat-numeral nat-power-eq power-Suc rel-simps*(*27*))
  show *?thesis*
    unfolding *unat-def*
    by (*induction n arbitrary*: *xi*) (*auto simp*: *shiftr-div-2n nat-div-distrib*)
qed

instantiation *uint32* :: *default*
begin
definition *default-uint32* :: *uint32* **where**
  ‹*default-uint32* = *0*›
instance
  ..
end

instance *uint32* :: *heap*
  by *standard* (*auto simp*: *inj-def exI*[*of - nat-of-uint32*])

instance *uint32* :: *semiring-numeral*
  by *standard*

instantiation *uint32* :: *hashable*
begin
definition *hashcode-uint32* :: ‹*uint32* ⇒ *uint32*› **where**
  ‹*hashcode-uint32 n* = *n*›

definition *def-hashmap-size-uint32* :: ‹*uint32 itself* ⇒ *nat*› **where**
‹*def-hashmap-size-uint32* = (λ-. *16*)›
— same as *nat*
instance
by *standard* (*simp add*: *def-hashmap-size-uint32-def*)
end

abbreviation *uint32-rel* :: ‹(*uint32* × *uint32*) *set*› **where**
‹*uint32-rel* ≡ *Id*›

lemma *nat-bin-trunc-ao*:
‹*nat* (*bintrunc n a*) *AND nat* (*bintrunc n b*) = *nat* (*bintrunc n* (*a AND b*))›
‹*nat* (*bintrunc n a*) *OR nat* (*bintrunc n b*) = *nat* (*bintrunc n* (*a OR b*))›
unfolding *bitAND-nat-def bitOR-nat-def*
by (*auto simp add*: *bin-trunc-ao bintr-ge0*)

lemma *nat-of-uint32-ao*:
‹*nat-of-uint32 n AND nat-of-uint32 m* = *nat-of-uint32* (*n AND m*)›
‹*nat-of-uint32 n OR nat-of-uint32 m* = *nat-of-uint32* (*n OR m*)›
subgoal apply (*transfer*, *unfold unat-def*, *transfer*, *unfold nat-bin-trunc-ao*) **..**
subgoal apply (*transfer*, *unfold unat-def*, *transfer*, *unfold nat-bin-trunc-ao*) **..**
done

lemma *nat-of-uint32-mod-2*:
‹*nat-of-uint32 L mod 2* = *nat-of-uint32* (*L mod 2*)›
by *transfer* (*auto simp*: *uint-mod unat-def nat-mod-distrib*)

lemma *bitAND-1-mod-2-uint32*: ‹*bitAND L 1* = *L mod 2*› **for** *L* :: *uint32*
proof −
have *H*: ‹*unat L mod 2* = *1* ∨ *unat L mod 2* = *0*› **for** *L*
by *auto*

show *?thesis*
apply (*subst word-nat-of-uint32-Rep-inject*[*symmetric*])
apply (*subst nat-of-uint32-ao*[*symmetric*])
apply (*subst nat-of-uint32-012*)
unfolding *bitAND-1-mod-2*
by (*rule nat-of-uint32-mod-2*)
qed

lemma *nat-uint-XOR*: ‹*nat* (*uint* (*a XOR b*)) = *nat* (*uint a*) *XOR nat* (*uint b*)›
if *len*: ‹*LENGTH*(′*a*) > *0*›
for *a b* :: ‹′*a* ::*len0 Word.word*›
proof −
have *1*: ‹*uint* ((*word-of-int*:: *int* ⇒ ′*a Word.word*)(*uint a*)) = *uint a*›
by (*subst* (*2*) *word-of-int-uint*[*of a*, *symmetric*]) (*rule refl*)
have *H*: ‹*nat* (*bintrunc n* (*a XOR b*)) = *nat* (*bintrunc n a XOR bintrunc n b*)›
if ‹*n*> *0*› **for** *n* **and** *a* :: *int* **and** *b* :: *int*
using *that*
proof (*induction n arbitrary*: *a b*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*) **note** *IH* = *this*(*1*) **and** *Suc* = *this*(*2*)
then show *?case*
proof (*cases n*)

case (*Suc m*)
moreover have
‹*nat* (*bintrunc m* (*bin-rest* (*bin-rest a*) *XOR bin-rest* (*bin-rest b*)) *BIT*
((*bin-last* (*bin-rest a*) ∨ *bin-last* (*bin-rest b*)) ∧
(*bin-last* (*bin-rest a*) ⟶ ¬ *bin-last* (*bin-rest b*))) *BIT*
((*bin-last a* ∨ *bin-last b*) ∧ (*bin-last a* ⟶ ¬ *bin-last b*))) =
nat ((*bintrunc m* (*bin-rest* (*bin-rest a*)) *XOR bintrunc m* (*bin-rest* (*bin-rest b*))) *BIT*
((*bin-last* (*bin-rest a*) ∨ *bin-last* (*bin-rest b*)) ∧
(*bin-last* (*bin-rest a*) ⟶ ¬ *bin-last* (*bin-rest b*))) *BIT*
((*bin-last a* ∨ *bin-last b*) ∧ (*bin-last a* ⟶ ¬ *bin-last b*)))›
(**is** ‹*nat* (*?n1 BIT ?b*) = *nat* (*?n2 BIT ?b*)›)
proof −
have *a1*: *nat ?n1* = *nat ?n2*
using *IH Suc* **by** *auto*
have *f2*: *0* ≤ *?n2*
by (*simp add*: *bintr-ge0*)
have *0* ≤ *?n1*
using *bintr-ge0* **by** *auto*
then have *?n2* = *?n1*
using *f2 a1* **by** *presburger*
then show *?thesis* **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
qed *simp*
qed
have ‹*nat* (*bintrunc LENGTH*('*a*) (*a XOR b*)) = *nat* (*bintrunc LENGTH*('*a*) *a XOR bintrunc LENGTH*('*a*) *b*)› **for** *a b*
using *len H*[*of* ‹*LENGTH*('*a*)› *a b*] **by** *auto*
then have ‹*nat* (*uint* (*a XOR b*)) = *nat* (*uint a XOR uint b*)›
by *transfer*
then show *?thesis*
unfolding *bitXOR-nat-def* **by** *auto*
qed

lemma *nat-of-uint32-XOR*: ‹*nat-of-uint32* (*a XOR b*) = *nat-of-uint32 a XOR nat-of-uint32 b*›
by *transfer* (*auto simp*: *unat-def nat-uint-XOR*)

lemma *nat-of-uint32-0-iff*: ‹*nat-of-uint32 xi* = *0* ⟷ *xi* = *0*› **for** *xi*
by *transfer* (*auto simp*: *unat-def uint-0-iff*)

lemma *nat-0-AND*: ‹*0 AND n* = *0*› **for** *n* :: *nat*
unfolding *bitAND-nat-def* **by** *auto*

lemma *uint32-0-AND*: ‹*0 AND n* = *0*› **for** *n* :: *uint32*
by *transfer auto*

definition *uint32-safe-minus* **where**
‹*uint32-safe-minus m n* = (*if m* < *n then 0 else m* − *n*)›

lemma *nat-of-uint32-le-minus*: ‹*ai* ≤ *bi* ⟹ *0* = *nat-of-uint32 ai* − *nat-of-uint32 bi*›
by *transfer* (*auto simp*: *unat-def word-le-def*)

lemma *nat-of-uint32-notle-minus*:
‹¬ *ai* < *bi* ⟹
nat-of-uint32 (*ai* − *bi*) = *nat-of-uint32 ai* − *nat-of-uint32 bi*›

apply *transfer*
unfolding *unat-def*
by (*subst uint-sub-lem*[*THEN iffD1*])
(*auto simp*: *unat-def uint-nonnegative nat-diff-distrib word-le-def*[*symmetric*] *intro*: *leI*)

lemma *nat-of-uint32-uint32-of-nat-id*: ‹$n \le$ *uint32-max* $\Longrightarrow$ *nat-of-uint32* (*uint32-of-nat n*) = *n*›
unfolding *uint32-of-nat-def uint32-max-def*
apply *simp*
apply *transfer*
apply (*auto simp*: *unat-def*)
apply *transfer*
by (*auto simp*: *less-upper-bintrunc-id*)

lemma *uint32-less-than-0*[*iff*]: ‹(*a*::*uint32*) $\le$ *0* $\longleftrightarrow$ *a* = *0*›
by *transfer auto*

lemma *nat-of-uint32-less-iff*: ‹*nat-of-uint32 a* < *nat-of-uint32 b* $\longleftrightarrow$ *a* < *b*›
apply *transfer*
apply (*auto simp*: *unat-def word-less-def*)
apply *transfer*
by (*smt bintr-ge0*)

lemma *nat-of-uint32-le-iff*: ‹*nat-of-uint32 a* $\le$ *nat-of-uint32 b* $\longleftrightarrow$ *a* $\le$ *b*›
apply *transfer*
by (*auto simp*: *unat-def word-less-def nat-le-iff word-le-def*)

lemma *nat-of-uint32-max*:
‹*nat-of-uint32* (*max ai bi*) = *max* (*nat-of-uint32 ai*) (*nat-of-uint32 bi*)›
by (*auto simp*: *max-def nat-of-uint32-le-iff split*: *if-splits*)

lemma *mult-mod-mod-mult*:
‹*b* < *n div a* $\Longrightarrow$ *a* > *0* $\Longrightarrow$ *b* > *0* $\Longrightarrow$ *a* ∗ *b mod n* = *a* ∗ (*b mod n*)› **for** *a b n* :: *int*
apply (*subst int-mod-eq′*)
subgoal using *not-le zdiv-mono1* **by** *fastforce*
subgoal using *not-le zdiv-mono1* **by** *fastforce*
subgoal
apply (*subst int-mod-eq′*)
subgoal by *auto*
subgoal by (*metis* (*full-types*) *le-cases not-le order-trans pos-imp-zdiv-nonneg-iff zdiv-le-dividend*)
subgoal by *auto*
done
done

lemma *nat-of-uint32-distrib-mult2*:
assumes ‹*nat-of-uint32 xi* $\le$ *uint32-max div 2*›
shows ‹*nat-of-uint32* (*2* ∗ *xi*) = *2* ∗ *nat-of-uint32 xi*›
proof −
have *H*: ‹$\bigwedge$*xi*::*32 Word.word. nat* (*uint xi*) < (*2147483648*::*nat*) $\Longrightarrow$
nat (*uint xi mod* (*4294967296*::*int*)) = *nat* (*uint xi*)›
proof −
fix *xia* :: *32 Word.word*
assume *a1*: *nat* (*uint xia*) < *2147483648*
have *f2*: $\bigwedge$*n.* (*numeral n*::*nat*) $\le$ *numeral* (*num.Bit0 n*)
by (*metis* (*no-types*) *add-0-right add-mono-thms-linordered-semiring*(*1*)
dual-order.order-iff-strict numeral-Bit0 rel-simps(*51*))

have *unat xia* ≤ *4294967296*
using *a1* **by** (*metis* (*no-types*) *add-0-right add-mono-thms-linordered-semiring*(*1*) *dual-order.order-iff-strict nat-int numeral-Bit0 rel-simps*(*51*) *uint-nat*)
then show *nat* (*uint xia mod 4294967296*) = *nat* (*uint xia*)
using *f2 a1* **by** *auto*
qed
have [*simp*]: ‹*xi* ≠ (*0*::*32 Word.word*) ⟹ (*0*::*int*) < *uint xi*› **for** *xi*
by (*metis* (*full-types*) *uint-eq-0 word-gt-0 word-less-def*)
show *?thesis*
using *assms* **unfolding** *uint32-max-def*
apply (*case-tac* ‹*xi* = *0*›)
subgoal by *auto*
subgoal by *transfer* (*auto simp*: *unat-def uint-word-ariths nat-mult-distrib mult-mod-mod-mult H*)
done
qed

lemma *nat-of-uint32-distrib-mult2-plus1*:
assumes ‹*nat-of-uint32 xi* ≤ *uint32-max div 2*›
shows ‹*nat-of-uint32* (*2* ∗ *xi* + *1*) = *2* ∗ *nat-of-uint32 xi* + *1*›
proof −
have *mod-is-id*: ‹⋀*xi*::*32 Word.word. nat* (*uint xi*) < (*2147483648*::*nat*) ⟹ (*uint xi mod* (*4294967296*::*int*)) = *uint xi*›
by (*subst zmod-trival-iff*) *auto*
have [*simp*]: ‹*xi* ≠ (*0*::*32 Word.word*) ⟹ (*0*::*int*) < *uint xi*› **for** *xi*
by (*metis* (*full-types*) *uint-eq-0 word-gt-0 word-less-def*)
show *?thesis*
using *assms* **by** *transfer* (*auto simp*: *unat-def uint-word-ariths nat-mult-distrib mult-mod-mod-mult mod-is-id nat-mod-distrib nat-add-distrib uint32-max-def*)
qed

lemma *nat-of-uint32-add*:
‹*nat-of-uint32 ai* + *nat-of-uint32 bi* ≤ *uint32-max* ⟹ *nat-of-uint32* (*ai* + *bi*) = *nat-of-uint32 ai* + *nat-of-uint32 bi*›
by *transfer* (*auto simp*: *unat-def uint-plus-if′ nat-add-distrib uint32-max-def*)

definition *zero-uint32-nat* **where**
[*simp*]: ‹*zero-uint32-nat* = (*0* :: *nat*)›

definition *one-uint32-nat* **where**
[*simp*]: ‹*one-uint32-nat* = (*1* :: *nat*)›

definition *two-uint32-nat* **where** [*simp*]: ‹*two-uint32-nat* = (*2* :: *nat*)›

definition *two-uint32* **where**
[*simp*]: ‹*two-uint32* = (*2* :: *uint32*)›

definition *fast-minus* :: ‹′*a*::{*minus*} ⇒ ′*a* ⇒ ′*a*› **where**
[*simp*]: ‹*fast-minus m n* = *m* − *n*›

definition *fast-minus-code* :: ‹′*a*::{*minus,ord*} ⇒ ′*a* ⇒ ′*a*› **where**
[*simp*]: ‹*fast-minus-code m n* = (*SOME p.* (*p* = *m* − *n* ∧ *m* ≥ *n*))›

definition *fast-minus-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*› **where**
[*simp, code del*]: ‹*fast-minus-nat* = *fast-minus-code*›

definition *fast-minus-nat'* :: ‹*nat* ⇒ *nat* ⇒ *nat*› **where**
  [*simp*, *code del*]: ‹*fast-minus-nat'* = *fast-minus-code*›

lemma [*code*]: ‹*fast-minus-nat* = *fast-minus-nat'*›
  unfolding *fast-minus-nat-def fast-minus-nat'-def* **..**

lemma *word-of-int-int-unat*[*simp*]: ‹*word-of-int* (*int* (*unat x*)) = *x*›
  unfolding *unat-def*
  apply *transfer*
  by (*simp add*: *bintr-ge0*)

lemma *uint32-of-nat-nat-of-uint32*[*simp*]: ‹*uint32-of-nat* (*nat-of-uint32 x*) = *x*›
  unfolding *uint32-of-nat-def*
  by *transfer auto*

definition *sum-mod-uint32-max* **where**
  ‹*sum-mod-uint32-max a b* = (*a* + *b*) *mod* (*uint32-max* + *1*)›

lemma *nat-of-uint32-plus*:
  ‹*nat-of-uint32* (*a* + *b*) = (*nat-of-uint32 a* + *nat-of-uint32 b*) *mod* (*uint32-max* + *1*)›
  by *transfer* (*auto simp*: *unat-word-ariths uint32-max-def*)

definition *one-uint32* **where**
  ‹*one-uint32* = (*1*::*uint32*)›

This lemma is meant to be used to simplify expressions like *nat-of-uint32 5* and therefore we add the bound explicitely instead of keeping *uint32-max*. Remark the types are non trivial here: we convert a *uint32* to a *nat*, even if the experession *numeral n* looks the same.

lemma *nat-of-uint32-numeral*[*simp*]:
  ‹*numeral n* ≤ ((*2* ^*32* − *1*)::*nat*) ⟹ *nat-of-uint32* (*numeral n*) = *numeral n*›
proof (*induction n*)
 case *One*
  then show *?case* **by** *auto*
next
  case (*Bit0 n*) **note** *IH* = *this*(*1*)[*unfolded uint32-max-def*[*symmetric*]] **and** *le* = *this*(*2*)
  define *m* :: *nat* **where** ‹*m* ≡ *numeral n*›
  have *n-le*: ‹*numeral n* ≤ *uint32-max*›
    using *le*
    by (*subst* (*asm*) *numeral.numeral-Bit0*) (*auto simp*: *m-def*[*symmetric*] *uint32-max-def*)
  have *n-le-div2*: ‹*nat-of-uint32* (*numeral n*) ≤ *uint32-max div 2*›
    apply (*subst IH*[*OF n-le*])
    using *le* **by** (*subst* (*asm*) *numeral.numeral-Bit0*) (*auto simp*: *m-def*[*symmetric*] *uint32-max-def*)

  have ‹*nat-of-uint32* (*numeral* (*num.Bit0 n*)) = *nat-of-uint32* (*2* ∗ *numeral n*)›
    by (*subst numeral.numeral-Bit0*)
      (*metis comm-monoid-mult-class.mult-1 distrib-right-numeral one-add-one*)
  also have ‹... = *2* ∗ *nat-of-uint32* (*numeral n*)›
    by (*subst nat-of-uint32-distrib-mult2*[*OF n-le-div2*]) (*rule refl*)
  also have ‹... = *2* ∗ *numeral n*›
    by (*subst IH*[*OF n-le*]) (*rule refl*)
  also have ‹... = *numeral* (*num.Bit0 n*)›
    by (*subst* (*2*) *numeral.numeral-Bit0*, *subst mult-2*)
      (*rule refl*)
  finally show *?case* **by** *simp*

next
  case (*Bit1 n*) **note** *IH* = *this*(*1*)[*unfolded uint32-max-def*[*symmetric*]] **and** *le* = *this*(*2*)

  define *m* :: *nat* **where** ‹*m* ≡ *numeral n*›
  have *n-le*: ‹*numeral n* ≤ *uint32-max*›
    using *le*
    by (*subst* (*asm*) *numeral.numeral-Bit1*) (*auto simp*: *m-def*[*symmetric*] *uint32-max-def*)
  have *n-le-div2*: ‹*nat-of-uint32* (*numeral n*) ≤ *uint32-max div 2*›
    apply (*subst IH*[*OF n-le*])
    using *le* **by** (*subst* (*asm*) *numeral.numeral-Bit1*) (*auto simp*: *m-def*[*symmetric*] *uint32-max-def*)

  have ‹*nat-of-uint32* (*numeral* (*num.Bit1 n*)) = *nat-of-uint32* (*2* ∗ *numeral n* + *1*)›
    by (*subst numeral.numeral-Bit1*)
      (*metis comm-monoid-mult-class.mult-1 distrib-right-numeral one-add-one*)
  also have ‹. . . = *2* ∗ *nat-of-uint32* (*numeral n*) + *1*›
    by (*subst nat-of-uint32-distrib-mult2-plus1*[*OF n-le-div2*]) (*rule refl*)
  also have ‹. . . = *2* ∗ *numeral n* + *1*›
    by (*subst IH*[*OF n-le*]) (*rule refl*)
  also have ‹. . . = *numeral* (*num.Bit1 n*)›
    by (*subst numeral.numeral-Bit1*) *linarith*
  finally show *?case* **by** *simp*
qed

lemma *nat-of-uint32-mod-232*:
  shows ‹*nat-of-uint32 xi* = *nat-of-uint32 xi mod 2^32*›
proof −
  show *?thesis*
    unfolding *uint32-max-def*
    subgoal apply *transfer*
      subgoal for *xi*
      by (*use word-unat.norm-Rep*[*of xi*] **in**
        ‹*auto simp*: *uint-word-ariths nat-mult-distrib mult-mod-mod-mult*
          simp del: *word-unat.norm-Rep*›)
    done
  done
qed

lemma *transfer-pow-uint32*:
  ‹*Transfer.Rel* (*rel-fun cr-uint32* (*rel-fun* (=) *cr-uint32*)) ((^)) ((^))›
proof −
  have [*simp*]: ‹*Rep-uint32 y* ^ *x* = *Rep-uint32* (*y* ^ *x*)› **for** *y* :: *uint32* **and** *x* :: *nat*
    by (*induction x*)
      (*auto simp*: *one-uint32.rep-eq times-uint32.rep-eq*)
  show *?thesis*
    by (*auto simp*: *Transfer.Rel-def rel-fun-def cr-uint32-def*)
qed

lemma *uint32-mod-232-eq*:
  fixes *xi* :: *uint32*
  shows ‹*xi* = *xi mod 2^32*›
proof −
  have *H*: ‹*nat-of-uint32* (*xi mod 2* ^ *32*) = *nat-of-uint32 xi*›
    apply *transfer*
    prefer *2*
      apply (*rule transfer-pow-uint32*)
    subgoal for *xi*

```
      using uint-word-ariths(1)[of xi 0]
      supply [[show-types]]
      apply auto
      apply (rule word-uint-eq-iff[THEN iffD2])
      apply (subst uint-mod-alt)
      by auto
    done

  show ?thesis
    by (rule word-nat-of-uint32-Rep-inject[THEN iffD1, OF H[symmetric]])
qed

lemma nat-of-uint32-numeral-mod-232:
  ‹nat-of-uint32 (numeral n) = numeral n mod 2^32›
  apply transfer
  apply (subst unat-numeral)
  by auto

lemma int-of-uint32-alt-def: ‹int-of-uint32 n = int (nat-of-uint32 n)›
   by (simp add: int-of-uint32.rep-eq nat-of-uint32.rep-eq unat-def)

lemma int-of-uint32-numeral[simp]:
  ‹numeral n ≤ ((2 ^ 32 − 1)::nat) ⟹ int-of-uint32 (numeral n) = numeral n›
  by (subst int-of-uint32-alt-def) simp

lemma nat-of-uint32-numeral-iff[simp]:
  ‹numeral n ≤ ((2 ^ 32 − 1)::nat) ⟹ nat-of-uint32 a = numeral n ⟷ a = numeral n›
  apply (rule iffI)
  prefer 2 apply (solves simp)
  using word-nat-of-uint32-Rep-inject by fastforce



lemma nat-of-uint32-mult-le:
   ‹nat-of-uint32 ai ∗ nat-of-uint32 bi ≤ uint32-max ⟹
       nat-of-uint32 (ai ∗ bi) = nat-of-uint32 ai ∗ nat-of-uint32 bi›
  apply transfer
  by (auto simp: unat-word-ariths uint32-max-def)

lemma nat-and-numerals [simp]:
  (numeral (Num.Bit0 x) :: nat) AND (numeral (Num.Bit0 y) :: nat) = (2 :: nat) ∗ (numeral x AND
numeral y)
  numeral (Num.Bit0 x) AND numeral (Num.Bit1 y) = (2 :: nat) ∗ (numeral x AND numeral y)
  numeral (Num.Bit1 x) AND numeral (Num.Bit0 y) = (2 :: nat) ∗ (numeral x AND numeral y)
  numeral (Num.Bit1 x) AND numeral (Num.Bit1 y) = (2 :: nat) ∗ (numeral x AND numeral y)+1
  (1::nat) AND numeral (Num.Bit0 y) = 0
  (1::nat) AND numeral (Num.Bit1 y) = 1
  numeral (Num.Bit0 x) AND (1::nat) = 0
  numeral (Num.Bit1 x) AND (1::nat) = 1
  (Suc 0::nat) AND numeral (Num.Bit0 y) = 0
  (Suc 0::nat) AND numeral (Num.Bit1 y) = 1
  numeral (Num.Bit0 x) AND (Suc 0::nat) = 0
  numeral (Num.Bit1 x) AND (Suc 0::nat) = 1
  Suc 0 AND Suc 0 = 1
  supply [[show-types]]
  by (auto simp: bitAND-nat-def Bit-def nat-add-distrib)
```

lemma *nat-of-uint32-div*:
‹*nat-of-uint32* (*a div b*) = *nat-of-uint32 a div nat-of-uint32 b*›
by *transfer* (*auto simp*: *unat-div*)

64-bits

definition *uint64-nat-rel* :: (*uint64* × *nat*) *set* **where**
‹*uint64-nat-rel* = *br nat-of-uint64* (λ-. *True*)›

abbreviation *uint64-rel* :: ‹(*uint64* × *uint64*) *set*› **where**
‹*uint64-rel* ≡ *Id*›

lemma *word-nat-of-uint64-Rep-inject*[*simp*]: ‹*nat-of-uint64 ai* = *nat-of-uint64 bi* ⟷ *ai* = *bi*›
by *transfer simp*

instantiation *uint64* :: *default*
begin
definition *default-uint64* :: *uint64* **where**
‹*default-uint64* = *0*›
instance
..
end

instance *uint64* :: *heap*
by *standard* (*auto simp*: *inj-def exI*[*of* - *nat-of-uint64*])

instance *uint64* :: *semiring-numeral*
by *standard*

lemma *nat-of-uint64-012*[*simp*]: ‹*nat-of-uint64 0* = *0*› ‹*nat-of-uint64 2* = *2*› ‹*nat-of-uint64 1* = *1*›
by (*transfer*, *auto*)+

definition *zero-uint64-nat* **where**
[*simp*]: ‹*zero-uint64-nat* = (*0* :: *nat*)›

definition *uint64-max* :: *nat* **where**
‹*uint64-max* = *2* ^*64* − *1*›

definition *uint64-max′* **where**
[*simp*, *symmetric*, *code*]: ‹*uint64-max′* = *uint64-max*›

lemma [*code*]: ‹*uint64-max′* = *18446744073709551615*›
by (*auto simp*: *uint64-max-def*)

lemma *nat-of-uint64-uint64-of-nat-id*: ‹*n* ≤ *uint64-max* ⟹ *nat-of-uint64* (*uint64-of-nat n*) = *n*›
unfolding *uint64-of-nat-def uint64-max-def*
apply *simp*
apply *transfer*
apply (*auto simp*: *unat-def*)
apply *transfer*
by (*auto simp*: *less-upper-bintrunc-id*)

lemma *nat-of-uint64-add*:
‹*nat-of-uint64 ai* + *nat-of-uint64 bi* ≤ *uint64-max* ⟹
nat-of-uint64 (*ai* + *bi*) = *nat-of-uint64 ai* + *nat-of-uint64 bi*›

by *transfer* (*auto simp*: *unat-def uint-plus-if′ nat-add-distrib uint64-max-def*)

definition *one-uint64-nat* **where**
[*simp*]: ‹*one-uint64-nat* = (*1* :: *nat*)›

lemma *uint64-less-than-0*[*iff*]: ‹(*a*::*uint64*) ≤ *0* ⟷ *a* = *0*›
by *transfer auto*

lemma *nat-of-uint64-less-iff*: ‹*nat-of-uint64 a* < *nat-of-uint64 b* ⟷ *a* < *b*›
apply *transfer*
apply (*auto simp*: *unat-def word-less-def*)
apply *transfer*
by (*smt bintr-ge0*)

lemma *nat-of-uint64-distrib-mult2*:
assumes ‹*nat-of-uint64 xi* ≤ *uint64-max div 2*›
shows ‹*nat-of-uint64* (*2* ∗ *xi*) = *2* ∗ *nat-of-uint64 xi*›
proof −
show *?thesis*
using *assms* **unfolding** *uint64-max-def*
apply (*case-tac* ‹*xi* = *0*›)
subgoal by *auto*
subgoal by *transfer* (*auto simp*: *unat-def uint-word-ariths nat-mult-distrib mult-mod-mod-mult*)
done
qed

lemma (**in** −)*nat-of-uint64-distrib-mult2-plus1*:
assumes ‹*nat-of-uint64 xi* ≤ *uint64-max div 2*›
shows ‹*nat-of-uint64* (*2* ∗ *xi* + *1*) = *2* ∗ *nat-of-uint64 xi* + *1*›
proof −
show *?thesis*
using *assms* **by** *transfer* (*auto simp*: *unat-def uint-word-ariths nat-mult-distrib mult-mod-mod-mult nat-mod-distrib nat-add-distrib uint64-max-def*)
qed

lemma *nat-of-uint64-numeral*[*simp*]:
‹*numeral n* ≤ ((*2* ^ *64* − *1*)::*nat*) ⟹ *nat-of-uint64* (*numeral n*) = *numeral n*›
proof (*induction n*)
case *One*
then show *?case* **by** *auto*
next
case (*Bit0 n*) **note** *IH* = *this*(*1*)[*unfolded uint64-max-def*[*symmetric*]] **and** *le* = *this*(*2*)
define *m* :: *nat* **where** ‹*m* ≡ *numeral n*›
have *n-le*: ‹*numeral n* ≤ *uint64-max*›
using *le*
by (*subst* (*asm*) *numeral.numeral-Bit0*) (*auto simp*: *m-def*[*symmetric*] *uint64-max-def*)
have *n-le-div2*: ‹*nat-of-uint64* (*numeral n*) ≤ *uint64-max div 2*›
apply (*subst IH*[*OF n-le*])
using *le* **by** (*subst* (*asm*) *numeral.numeral-Bit0*) (*auto simp*: *m-def*[*symmetric*] *uint64-max-def*)

have ‹*nat-of-uint64* (*numeral* (*num.Bit0 n*)) = *nat-of-uint64* (*2* ∗ *numeral n*)›
by (*subst numeral.numeral-Bit0*)
(*metis comm-monoid-mult-class.mult-1 distrib-right-numeral one-add-one*)
also have ‹. . . = *2* ∗ *nat-of-uint64* (*numeral n*)›

by (*subst nat-of-uint64-distrib-mult2*[*OF n-le-div2*]) (*rule refl*)
also have ‹. . . = *2* ∗ *numeral n*›
by (*subst IH*[*OF n-le*]) (*rule refl*)
also have ‹. . . = *numeral* (*num.Bit0 n*)›
by (*subst* (*2*) *numeral.numeral-Bit0*, *subst mult-2*)
(*rule refl*)
finally show *?case* by *simp*
next
case (*Bit1 n*) note *IH* = *this*(*1*)[*unfolded uint64-max-def*[*symmetric*]] and *le* = *this*(*2*)

define *m* :: *nat* where ‹*m* ≡ *numeral n*›
have *n-le*: ‹*numeral n* ≤ *uint64-max*›
using *le*
by (*subst* (*asm*) *numeral.numeral-Bit1*) (*auto simp*: *m-def*[*symmetric*] *uint64-max-def*)
have *n-le-div2*: ‹*nat-of-uint64* (*numeral n*) ≤ *uint64-max div 2*›
apply (*subst IH*[*OF n-le*])
using *le* by (*subst* (*asm*) *numeral.numeral-Bit1*) (*auto simp*: *m-def*[*symmetric*] *uint64-max-def*)

have ‹*nat-of-uint64* (*numeral* (*num.Bit1 n*)) = *nat-of-uint64* (*2* ∗ *numeral n* + *1*)›
by (*subst numeral.numeral-Bit1*)
(*metis comm-monoid-mult-class.mult-1 distrib-right-numeral one-add-one*)

also have ‹. . . = *2* ∗ *nat-of-uint64* (*numeral n*) + *1*›
by (*subst nat-of-uint64-distrib-mult2-plus1*[*OF n-le-div2*]) (*rule refl*)
also have ‹. . . = *2* ∗ *numeral n* + *1*›
by (*subst IH*[*OF n-le*]) (*rule refl*)
also have ‹. . . = *numeral* (*num.Bit1 n*)›
by (*subst numeral.numeral-Bit1*) *linarith*
finally show *?case* by *simp*
qed

lemma *int-of-uint64-alt-def*: ‹*int-of-uint64 n* = *int* (*nat-of-uint64 n*)›
by (*simp add*: *int-of-uint64.rep-eq nat-of-uint64.rep-eq unat-def*)

lemma *int-of-uint64-numeral*[*simp*]:
‹*numeral n* ≤ ((*2* ^ *64* − *1*)::*nat*) ⟹ *int-of-uint64* (*numeral n*) = *numeral n*›
by (*subst int-of-uint64-alt-def*) *simp*

lemma *nat-of-uint64-numeral-iff*[*simp*]:
‹*numeral n* ≤ ((*2* ^ *64* − *1*)::*nat*) ⟹ *nat-of-uint64 a* = *numeral n* ⟷ *a* = *numeral n*›
apply (*rule iffI*)
prefer *2* apply (*solves simp*)
using *word-nat-of-uint64-Rep-inject* by *fastforce*

lemma *numeral-uint64-eq-iff*[*simp*]:
‹*numeral m* ≤ (*2*^*64*−*1* :: *nat*) ⟹ *numeral n* ≤ (*2*^*64*−*1* :: *nat*) ⟹ ((*numeral m* :: *uint64*) = *numeral n*) ⟷ *numeral m* = (*numeral n* :: *nat*)›
by (*subst word-nat-of-uint64-Rep-inject*[*symmetric*])
(*auto simp*: *uint64-max-def*)

lemma *numeral-uint64-eq0-iff*[*simp*]:
‹*numeral n* ≤ (*2*^*64*−*1* :: *nat*) ⟹ ((*0* :: *uint64*) = *numeral n*) ⟷ *0* = (*numeral n* :: *nat*)›
by (*subst word-nat-of-uint64-Rep-inject*[*symmetric*])
(*auto simp*: *uint64-max-def*)

lemma *transfer-pow-uint64*: ‹*Transfer.Rel* (*rel-fun cr-uint64* (*rel-fun* (=) *cr-uint64*)) (^) (^)›
  apply (*auto simp*: *Transfer.Rel-def rel-fun-def cr-uint64-def*)
  subgoal for *x y*
    by (*induction y*)
      (*auto simp*: *one-uint64.rep-eq times-uint64.rep-eq*)
  done

lemma *shiftl-t2n-uint64*: ‹*n* << *m* = *n* ∗ *2* ^ *m*› **for** *n* :: *uint64*
  apply *transfer*
  prefer *2* **apply** (*rule transfer-pow-uint64*)
  by (*auto simp*: *shiftl-t2n*)

lemma *mod2-bin-last*: ‹*a mod 2* = *0* ⟷ ¬*bin-last a*›
  by (*auto simp*: *bin-last-def*)

lemma *bitXOR-1-if-mod-2-int*: ‹*bitOR L 1* = (*if L mod 2* = *0 then L* + *1 else L*)› **for** *L* :: *int*
  apply (*rule bin-rl-eqI*)
  unfolding *bin-rest-OR bin-last-OR*
   apply (*auto simp*: *bin-rest-def bin-last-def*)
  done

lemma *bitOR-1-if-mod-2-nat*:
  ‹*bitOR L 1* = (*if L mod 2* = *0 then L* + *1 else L*)›
  ‹*bitOR L* (*Suc 0*) = (*if L mod 2* = *0 then L* + *1 else L*)› **for** *L* :: *nat*
proof −
  have *H*: ‹*bitOR L 1* = *L* + (*if bin-last* (*int L*) *then 0 else 1*)›
    unfolding *bitOR-nat-def*
    apply (*auto simp*: *bitOR-nat-def bin-last-def*
        bitXOR-1-if-mod-2-int)
    done
  show ‹*bitOR L 1* = (*if L mod 2* = *0 then L* + *1 else L*)›
    unfolding *H*
    apply (*auto simp*: *bitOR-nat-def bin-last-def*)
    apply *presburger*+
    done
  then show ‹*bitOR L* (*Suc 0*) = (*if L mod 2* = *0 then L* + *1 else L*)›
    by *simp*
qed

lemma *uint64-max-uint-def*: ‹*unat* (−*1* :: *64 Word.word*) = *uint64-max*›
proof −
  have ‹*unat* (−*1* :: *64 Word.word*) = *unat* (− *Numeral1* :: *64 Word.word*)›
    unfolding *numeral.numeral-One* **..**
  also have ‹. . . = *uint64-max*›
    unfolding *unat-bintrunc-neg*
    apply (*simp add*: *uint64-max-def*)
    apply (*subst numeral-eq-Suc*; *subst bintrunc.Suc*; *simp*)+
    done
  finally show *?thesis* **.**
qed

lemma *nat-of-uint64-le-uint64-max*: ‹*nat-of-uint64 x* ≤ *uint64-max*›
  apply *transfer*

subgoal for *x*
using *word-le-nat-alt*[*of x* ‹− *1*›]
unfolding *uint64-max-def* [*symmetric*] *uint64-max-uint-def*
by *auto*
done

lemma *bitOR-1-if-mod-2-uint64*: ‹*bitOR L 1* = (*if L mod 2* = *0 then L* + *1 else L*)› **for** *L* :: *uint64*
proof −
have *H*: ‹*bitOR L 1* = *a* ⟷ *bitOR* (*nat-of-uint64 L*) *1* = *nat-of-uint64 a*› **for** *a*
apply *transfer*
apply (*rule iffI*)
subgoal for *L a*
by (*auto simp*: *unat-def uint-or bitOR-nat-def*)
subgoal for *L a*
by (*auto simp*: *unat-def uint-or bitOR-nat-def eq-nat-nat-iff*
word-or-def)
done
have *K*: ‹*L mod 2* = *0* ⟷ *nat-of-uint64 L mod 2* = *0*›
apply *transfer*
subgoal for *L*
using *unat-mod*[*of L 2*]
by (*auto simp*: *unat-eq-0*)
done
have *L*: ‹*nat-of-uint64* (*if L mod 2* = *0 then L* + *1 else L*) =
(*if nat-of-uint64 L mod 2* = *0 then nat-of-uint64 L* + *1 else nat-of-uint64 L*)›
using *nat-of-uint64-le-uint64-max*[*of L*]
by (*auto simp*: *K nat-of-uint64-add uint64-max-def*)

show *?thesis*
apply (*subst H*)
unfolding *bitOR-1-if-mod-2-nat*[*symmetric*] *L* **..**
qed

lemma *nat-of-uint64-plus*:
‹*nat-of-uint64* (*a* + *b*) = (*nat-of-uint64 a* + *nat-of-uint64 b*) *mod* (*uint64-max* + *1*)›
by *transfer* (*auto simp*: *unat-word-ariths uint64-max-def*)

lemma *nat-and*:
‹*ai*≥ *0* ⟹ *bi* ≥ *0* ⟹ *nat* (*ai AND bi*) = *nat ai AND nat bi*›
by (*auto simp*: *bitAND-nat-def*)

lemma *nat-of-uint64-and*:
‹*nat-of-uint64 ai* ≤ *uint64-max* ⟹ *nat-of-uint64 bi* ≤ *uint64-max* ⟹
nat-of-uint64 (*ai AND bi*) = *nat-of-uint64 ai AND nat-of-uint64 bi*›
unfolding *uint64-max-def*
by *transfer* (*auto simp*: *unat-def uint-and nat-and*)

definition *two-uint64-nat* :: *nat* **where**
[*simp*]: ‹*two-uint64-nat* = *2*›

lemma *nat-or*:
‹*ai*≥ *0* ⟹ *bi* ≥ *0* ⟹ *nat* (*ai OR bi*) = *nat ai OR nat bi*›
by (*auto simp*: *bitOR-nat-def*)

lemma *nat-of-uint64-or*:

⟨*nat-of-uint64 ai* ≤ *uint64-max* ⟹ *nat-of-uint64 bi* ≤ *uint64-max* ⟹
nat-of-uint64 (*ai OR bi*) = *nat-of-uint64 ai OR nat-of-uint64 bi*⟩
unfolding *uint64-max-def*
by *transfer* (*auto simp*: *unat-def uint-or nat-or*)

lemma *Suc-0-le-uint64-max*: ⟨*Suc 0* ≤ *uint64-max*⟩
by (*auto simp*: *uint64-max-def*)

lemma *nat-of-uint64-le-iff*: ⟨*nat-of-uint64 a* ≤ *nat-of-uint64 b* ⟷ *a* ≤ *b*⟩
apply *transfer*
by (*auto simp*: *unat-def word-less-def nat-le-iff word-le-def*)

lemma *nat-of-uint64-notle-minus*:
⟨¬ *ai* < *bi* ⟹
nat-of-uint64 (*ai* − *bi*) = *nat-of-uint64 ai* − *nat-of-uint64 bi*⟩
apply *transfer*
unfolding *unat-def*
by (*subst uint-sub-lem*[*THEN iffD1*])
(*auto simp*: *unat-def uint-nonnegative nat-diff-distrib word-le-def*[*symmetric*] *intro*: *leI*)

lemma *le-uint32-max-le-uint64-max*: ⟨*a* ≤ *uint32-max* + *2* ⟹ *a* ≤ *uint64-max*⟩
by (*auto simp*: *uint32-max-def uint64-max-def*)

lemma *nat-of-uint64-ge-minus*:
⟨*ai* ≥ *bi* ⟹
nat-of-uint64 (*ai* − *bi*) = *nat-of-uint64 ai* − *nat-of-uint64 bi*⟩
apply *transfer*
unfolding *unat-def*
by (*subst uint-sub-lem*[*THEN iffD1*])
(*auto simp*: *unat-def uint-nonnegative nat-diff-distrib word-le-def*[*symmetric*] *intro*: *leI*)

definition *sum-mod-uint64-max* **where**
⟨*sum-mod-uint64-max a b* = (*a* + *b*) *mod* (*uint64-max* + *1*)⟩

definition *uint32-max-uint32* :: *uint32* **where**
⟨*uint32-max-uint32* = − *1*⟩

lemma *nat-of-uint32-uint32-max-uint32*[*simp*]:
⟨*nat-of-uint32* (*uint32-max-uint32*) = *uint32-max*⟩
proof −
have ⟨*unat* (*Rep-uint32* (−*1*) :: *32 Word.word*) = *unat* (− *Numeral1* :: *32 Word.word*)⟩
unfolding *numeral.numeral-One uminus-uint32.rep-eq one-uint32.rep-eq* **..**
also have ⟨. . . = *uint32-max*⟩
unfolding *unat-bintrunc-neg*
apply (*simp add*: *uint32-max-def*)
apply (*subst numeral-eq-Suc*; *subst bintrunc.Suc*; *simp*)+
done
finally show *?thesis* **by** (*simp add*: *nat-of-uint32-def uint32-max-uint32-def*)
qed

lemma *sum-mod-uint64-max-le-uint64-max*[*simp*]: ⟨*sum-mod-uint64-max a b* ≤ *uint64-max*⟩
unfolding *sum-mod-uint64-max-def*
by *auto*

definition *uint64-of-uint32* **where**
‹*uint64-of-uint32 n* = *uint64-of-nat* (*nat-of-uint32 n*)›

export-code *uint64-of-uint32* **in** *SML*

We do not want to follow the definition in the generated code (that would be crazy).

definition *uint64-of-uint32'* **where**
[*symmetric*, *code*]: ‹*uint64-of-uint32'* = *uint64-of-uint32*›

code-printing constant *uint64-of-uint32'* ⇀
(*SML*) (*Uint64.fromLarge* (*Word32.toLarge* (-)))

export-code *uint64-of-uint32* **checking** *SML-imp*

export-code *uint64-of-uint32* **in** *SML-imp*

lemma
assumes *n*[*simp*]: ‹*n* ≤ *uint32-max-uint32*›
shows ‹*nat-of-uint64* (*uint64-of-uint32 n*) = *nat-of-uint32 n*›
proof −

have *H*: ‹*nat-of-uint32 n* ≤ *uint32-max*› **if** ‹*n* ≤ *uint32-max-uint32*› **for** *n*
apply (*subst nat-of-uint32-uint32-max-uint32*[*symmetric*])
apply (*subst nat-of-uint32-le-iff*)
by (*auto simp*: *that*)
have [*simp*]: ‹*nat-of-uint32 n* ≤ *uint64-max*› **if** ‹*n* ≤ *uint32-max-uint32*› **for** *n*
using *H*[*of n*] **by** (*auto simp*: *that uint64-max-def uint32-max-def*)
show *?thesis*
apply (*auto simp*: *uint64-of-uint32-def*
nat-of-uint64-uint64-of-nat-id uint64-max-def)
by (*subst nat-of-uint64-uint64-of-nat-id*) *auto*
qed

definition *zero-uint64* **where**
‹*zero-uint64* ≡ (*0* :: *uint64*)›
definition *zero-uint32* **where**
‹*zero-uint32* ≡ (*0* :: *uint32*)›
definition *two-uint64* **where** ‹*two-uint64* = (*2* :: *uint64*)›

lemma *nat-of-uint64-ao*:
‹*nat-of-uint64 m AND nat-of-uint64 n* = *nat-of-uint64* (*m AND n*)›
‹*nat-of-uint64 m OR nat-of-uint64 n* = *nat-of-uint64* (*m OR n*)›
by (*simp-all add*: *nat-of-uint64-and nat-of-uint64-or nat-of-uint64-le-uint64-max*)

Conversions

From nat to 64 bits **definition** *uint64-of-nat-conv* :: ‹*nat* ⇒ *nat*› **where**
‹*uint64-of-nat-conv i* = *i*›

From nat to 32 bits **definition** *nat-of-uint32-spec* :: ‹*nat* ⇒ *nat*› **where**
[*simp*]: ‹*nat-of-uint32-spec n* = *n*›

From 64 to nat bits **definition** *nat-of-uint64-conv* :: ‹*nat* ⇒ *nat*› **where**
[*simp*]: ‹*nat-of-uint64-conv i* = *i*›

From 32 to nat bits **definition** *nat-of-uint32-conv* :: ‹*nat* ⇒ *nat*› **where**
[*simp*]: ‹*nat-of-uint32-conv i* = *i*›

definition *convert-to-uint32* :: ‹*nat* ⇒ *nat*› **where**
[*simp*]: ‹*convert-to-uint32* = *id*›

From 32 to 64 bits **definition** *uint64-of-uint32-conv* :: ‹*nat* ⇒ *nat*› **where**
[*simp*]: ‹*uint64-of-uint32-conv x* = *x*›

lemma *nat-of-uint32-le-uint32-max*: ‹*nat-of-uint32 n* ≤ *uint32-max*›
using *nat-of-uint32-plus*[*of n 0*]
pos-mod-bound[*of* ‹*uint32-max* + *1*› ‹*nat-of-uint32 n*›]
by *auto*

lemma *nat-of-uint32-le-uint64-max*: ‹*nat-of-uint32 n* ≤ *uint64-max*›
using *nat-of-uint32-le-uint32-max*[*of n*] **unfolding** *uint64-max-def uint32-max-def*
by *auto*

lemma *nat-of-uint64-uint64-of-uint32*: ‹*nat-of-uint64* (*uint64-of-uint32 n*) = *nat-of-uint32 n*›
unfolding *uint64-of-uint32-def*
by (*auto simp*: *nat-of-uint64-uint64-of-nat-id nat-of-uint32-le-uint64-max*)

From 64 to 32 bits **definition** *uint32-of-uint64* **where**
‹*uint32-of-uint64 n* = *uint32-of-nat* (*nat-of-uint64 n*)›

definition *uint32-of-uint64-conv* **where**
[*simp*]: ‹*uint32-of-uint64-conv n* = *n*›

lemma (**in** −) *uint64-neq0-gt*: ‹*j* ≠ (*0*::*uint64*) ⟷ *j* > *0*›
by *transfer* (*auto simp*: *word-neq-0-conv*)

lemma *uint64-gt0-ge1*: ‹*j* > *0* ⟷ *j* ≥ (*1*::*uint64*)›
apply (*subst nat-of-uint64-less-iff*[*symmetric*])
apply (*subst nat-of-uint64-le-iff*[*symmetric*])
by *auto*

definition *three-uint32* **where** ‹*three-uint32* = (*3* :: *uint32*)›

definition *nat-of-uint64-id-conv* :: ‹*uint64* ⇒ *nat*› **where**
‹*nat-of-uint64-id-conv* = *nat-of-uint64*›

definition *op-map* :: (′*b* ⇒ ′*a*) ⇒ ′*a* ⇒ ′*b list* ⇒ ′*a list nres* **where**
‹*op-map R e xs* = *do* {
let zs = *replicate* (*length xs*) *e*;
(-, *zs*) ← $WHILE_T{}^{\lambda(i,zs).\ i \le length\ xs \wedge take\ i\ zs = map\ R\ (take\ i\ xs) \wedge \quad length\ zs = length\ xs \wedge (\forall k{\ge}i.\ k < length\ x}$
(λ(*i*, *zs*). *i* < *length zs*)
(λ(*i*, *zs*). *do* {*ASSERT*(*i* < *length zs*); *RETURN* (*i*+*1*, *zs*[*i* := *R* (*xs*!*i*)])})
(*0*, *zs*);
RETURN zs
}›

lemma *op-map-map*: ‹*op-map R e xs* ≤ *RETURN* (*map R xs*)›
unfolding *op-map-def Let-def*

by (*refine-vcg WHILEIT-rule*[**where** *R*=‹*measure* (λ(*i*,-). *length xs* − *i*)›])
(*auto simp*: *last-conv-nth take-Suc-conv-app-nth list-update-append split*: *nat.splits*)

lemma *op-map-map-rel*:
‹(*op-map R e*, *RETURN o* (*map R*)) ∈ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
by (*intro frefI nres-relI*) (*auto simp*: *op-map-map*)

definition *array-nat-of-uint64-conv* :: ‹*nat list* ⇒ *nat list*› **where**
‹*array-nat-of-uint64-conv* = *id*›

definition *array-nat-of-uint64* :: *nat list* ⇒ *nat list nres* **where**
‹*array-nat-of-uint64 xs* = *op-map nat-of-uint64-conv 0 xs*›

lemma *array-nat-of-uint64-conv-alt-def*:
‹*array-nat-of-uint64-conv* = *map nat-of-uint64-conv*›
unfolding *nat-of-uint64-conv-def array-nat-of-uint64-conv-def* **by** *auto*

definition *array-uint64-of-nat-conv* :: ‹*nat list* ⇒ *nat list*› **where**
‹*array-uint64-of-nat-conv* = *id*›

definition *array-uint64-of-nat* :: *nat list* ⇒ *nat list nres* **where**
‹*array-uint64-of-nat xs* = *op-map uint64-of-nat-conv zero-uint64-nat xs*›

end
theory *WB-Word-Assn*
imports *Refine-Imperative-HOL.IICF*
WB-Word Bits-Natural
WB-More-Refinement WB-More-IICF-SML
begin

0.1.5 More Setup for Fixed Size Natural Numbers

Words

abbreviation *word-nat-assn* :: *nat* ⇒ *'a*::*len0 Word.word* ⇒ *assn* **where**
‹*word-nat-assn* ≡ *pure word-nat-rel*›

lemma *op-eq-word-nat*:
‹(*uncurry* (*return oo* ((=) :: *'a* :: *len Word.word* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈
word-nat-assn^k $*_a$ *word-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *word-nat-rel-def br-def*)

abbreviation *uint32-nat-assn* :: *nat* ⇒ *uint32* ⇒ *assn* **where**
‹*uint32-nat-assn* ≡ *pure uint32-nat-rel*›

lemma *op-eq-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* ((=) :: *uint32* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈
uint32-nat-assn^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

abbreviation *uint32-assn* :: ‹*uint32* ⇒ *uint32* ⇒ *assn*› **where**
‹*uint32-assn* ≡ *id-assn*›

lemma *op-eq-uint32*:
‹(*uncurry* (*return oo* ((=) :: *uint32* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈

uint32-assn^k $*_a$ *uint32-assn*^{k} $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemmas [*id-rules*] =
itypeI[*Pure.of 0 TYPE* (*uint32*)]
itypeI[*Pure.of 1 TYPE* (*uint32*)]

lemma *param-uint32*[*param*, *sepref-import-param*]:
(*0*, *0*::*uint32*) ∈ *Id*
(*1*, *1*::*uint32*) ∈ *Id*
by (*rule IdI*)+

lemma *param-max-uint32*[*param*,*sepref-import-param*]:
(*max*,*max*)∈*uint32-rel* → *uint32-rel* → *uint32-rel* **by** *auto*

lemma *max-uint32*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo max*), *uncurry* (*RETURN oo max*)) ∈
uint32-assn^k $*_a$ *uint32-assn*^{k} $\rightarrow_a$ *uint32-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *uint32-nat-assn-minus*:
‹(*uncurry* (*return oo uint32-safe-minus*), *uncurry* (*RETURN oo* (−))) ∈
uint32-nat-assn^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def nat-of-uint32-le-minus*
br-def uint32-safe-minus-def nat-of-uint32-notle-minus)

lemma [*safe-constraint-rules*]:
‹*CONSTRAINT IS-LEFT-UNIQUE uint32-nat-rel*›
‹*CONSTRAINT IS-RIGHT-UNIQUE uint32-nat-rel*›
by (*auto simp*: *IS-LEFT-UNIQUE-def single-valued-def uint32-nat-rel-def br-def*)

lemma *shiftr1*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* ((>>))), *uncurry* (*RETURN oo* (>>))) ∈ *uint32-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$
uint32-assn›
by *sepref-to-hoare* (*sep-auto simp*: *shiftr1-def uint32-nat-rel-def br-def*)

lemma *shiftl1*[*sepref-fr-rules*]: ‹(*return o shiftl1*, *RETURN o shiftl1*) ∈ *nat-assn*^{k} $\rightarrow_a$ *nat-assn*›
by *sepref-to-hoare sep-auto*

lemma *nat-of-uint32-rule*[*sepref-fr-rules*]:
‹(*return o nat-of-uint32*, *RETURN o nat-of-uint32*) ∈ *uint32-assn*^{k} $\rightarrow_a$ *nat-assn*›
by *sepref-to-hoare sep-auto*

lemma *max-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo max*), *uncurry* (*RETURN oo max*)) ∈ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$
uint32-nat-assn›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-max*)

lemma *array-set-hnr-u*:
‹*CONSTRAINT is-pure A* ⟹
(*uncurry2* (λ*xs i. heap-array-set xs* (*nat-of-uint32 i*)), *uncurry2* (*RETURN* ∘∘∘ *op-list-set*)) ∈
[*pre-list-set*]$_a$ (*array-assn A*)^d $*_a$ *uint32-nat-assn*^{k} $*_a$ A^k → *array-assn A*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*

hr-comp-def list-rel-pres-length list-rel-update)

lemma *array-get-hnr-u*:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry* (λ*xs i*. *Array.nth xs* (*nat-of-uint32 i*)),
uncurry (*RETURN* ∘∘ *op-list-get*)) ∈ [*pre-list-get*]$_a$ (*array-assn A*)^k $*_a$ *uint32-nat-assn*^{k} → *A*›
proof −
obtain A' **where**
A: ‹*pure* A' = *A*›
using *assms pure-the-pure* **by** *auto*
then have A': ‹*the-pure A* = A'›
by *auto*
have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ A')) = A'›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update param-nth A' *A*[*symmetric*] *ent-refl-true*
list-rel-eq-listrel listrel-iff-nth pure-def)
qed

lemma *arl-get-hnr-u*:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry* (λ*xs i*. *arl-get xs* (*nat-of-uint32 i*)), *uncurry* (*RETURN* ∘∘ *op-list-get*))
∈ [*pre-list-get*]$_a$ (*arl-assn A*)^k $*_a$ *uint32-nat-assn*^{k} → *A*›
proof −
obtain A' **where**
A: ‹*pure* A' = *A*›
using *assms pure-the-pure* **by** *auto*
then have A': ‹*the-pure A* = A'›
by *auto*
have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ A')) = A'›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update param-nth arl-assn-def
A' *A*[*symmetric*] *pure-def*)
qed

lemma *uint32-nat-assn-plus*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ [λ(*m*, *n*). *m* + *n* ≤ *uint32-max*]$_a$
uint32-nat-assn^k $*_a$ *uint32-nat-assn*^{k} → *uint32-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def nat-of-uint32-add br-def*)

lemma *uint32-nat-assn-one*:
‹(*uncurry0* (*return 1*), *uncurry0* (*RETURN 1*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *uint32-nat-assn-zero*:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN 0*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *nat-of-uint32-int32-assn*:
‹(*return o id*, *RETURN o nat-of-uint32*) ∈ *uint32-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›

by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *uint32-nat-assn-zero-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint32-nat*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *nat-assn-zero*:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN 0*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *one-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 1*), *uncurry0* (*RETURN one-uint32-nat*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *uint32-nat-assn-less*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (<)), *uncurry* (*RETURN oo* (<))) ∈
uint32-nat-assn^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def max-def*
nat-of-uint32-less-iff)

lemma *uint32-2-hnr*[*sepref-fr-rules*]: ‹(*uncurry0* (*return two-uint32*), *uncurry0* (*RETURN two-uint32-nat*))
∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def two-uint32-nat-def*)

Do NOT declare this theorem as *sepref-fr-rules* to avoid bad unexpected conversions.

lemma *le-uint32-nat-hnr*:
‹(*uncurry* (*return oo* (*λa b. nat-of-uint32 a* < *b*)), *uncurry* (*RETURN oo* (<))) ∈
uint32-nat-assn^k $*_a$ *nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *le-nat-uint32-hnr*:
‹(*uncurry* (*return oo* (*λa b. a* < *nat-of-uint32 b*)), *uncurry* (*RETURN oo* (<))) ∈
nat-assn^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

code-printing constant *fast-minus-nat'* ⇀ (*SML-imp*) (*Nat*(*integer'-of'-nat*/ (-)/ −/ *integer'-of'-nat*/ (-)))

lemma *fast-minus-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo fast-minus-nat*), *uncurry* (*RETURN oo fast-minus*)) ∈
$[\lambda(m, n).\ m \geq n]_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} → *nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-le-minus*
nat-of-uint32-notle-minus nat-of-uint32-le-iff)

definition *fast-minus-uint32* :: ‹*uint32* ⇒ *uint32* ⇒ *uint32*› **where**
[*simp*]: ‹*fast-minus-uint32* = *fast-minus*›

lemma *fast-minus-uint32*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo fast-minus-uint32*), *uncurry* (*RETURN oo fast-minus*)) ∈
$[\lambda(m, n).\ m \geq n]_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} → *uint32-nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-le-minus*
nat-of-uint32-notle-minus nat-of-uint32-le-iff)

lemma *uint32-nat-assn-0-eq*: ‹*uint32-nat-assn 0 a* = ↑ (*a* = *0*)›
 by (*auto simp*: *uint32-nat-rel-def br-def pure-def nat-of-uint32-0-iff*)

lemma *uint32-nat-assn-nat-assn-nat-of-uint32*:
 ‹*uint32-nat-assn aa a* = *nat-assn aa* (*nat-of-uint32 a*)›
 by (*auto simp*: *pure-def uint32-nat-rel-def br-def*)

lemma *sum-mod-uint32-max*: ‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo sum-mod-uint32-max*)) ∈
 $uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a$
 uint32-nat-assn›
 by *sepref-to-hoare*
 (*sep-auto simp*: *sum-mod-uint32-max-def uint32-nat-rel-def br-def nat-of-uint32-plus*)

lemma *le-uint32-nat-rel-hnr*[*sepref-fr-rules*]:
 ‹(*uncurry* (*return oo* (≤)), *uncurry* (*RETURN oo* (≤))) ∈
 $uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a bool\text{-}assn$›
 by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-le-iff*)

lemma *one-uint32-hnr*[*sepref-fr-rules*]:
 ‹(*uncurry0* (*return 1*), *uncurry0* (*RETURN one-uint32*)) ∈ $unit\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
 by *sepref-to-hoare* (*sep-auto simp*: *one-uint32-def*)

lemma *sum-uint32-assn*[*sepref-fr-rules*]:
 ‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ $uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
 by *sepref-to-hoare sep-auto*

lemma *Suc-uint32-nat-assn-hnr*:
 ‹(*return o* (λ*n*. *n* + *1*), *RETURN o Suc*) ∈ $[\lambda n.\ n < uint32\text{-}max]_a\ uint32\text{-}nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn$›
 by *sepref-to-hoare* (*sep-auto simp*: *br-def uint32-nat-rel-def nat-of-uint32-add*)

lemma *minus-uint32-assn*:
 ‹(*uncurry* (*return oo* (−)), *uncurry* (*RETURN oo* (−))) ∈ $uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
 by *sepref-to-hoare sep-auto*

lemma *bitAND-uint32-nat-assn*[*sepref-fr-rules*]:
 ‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*))) ∈
 $uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a uint32\text{-}nat\text{-}assn$›
 by *sepref-to-hoare*
 (*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-ao*)

lemma *bitAND-uint32-assn*[*sepref-fr-rules*]:
 ‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*))) ∈
 $uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
 by *sepref-to-hoare*
 (*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-ao*)

lemma *bitOR-uint32-nat-assn*[*sepref-fr-rules*]:
 ‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*))) ∈
 $uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a uint32\text{-}nat\text{-}assn$›
 by *sepref-to-hoare*
 (*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-ao*)

lemma *bitOR-uint32-assn*[*sepref-fr-rules*]:
 ‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*))) ∈

$uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-ao*)

lemma *uint32-nat-assn-mult*:
‹(*uncurry* (*return oo* ((∗))), *uncurry* (*RETURN oo* ((∗)))) ∈ $[\lambda(a, b).\ a * b \le uint32\text{-}max]_a$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-mult-le*)

lemma [*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*div*)), *uncurry* (*RETURN oo* (*div*))) ∈
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a uint32\text{-}nat\text{-}assn$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-div*)

64-bits

lemmas [*id-rules*] =
itypeI[*Pure.of 0 TYPE* (*uint64*)]
itypeI[*Pure.of 1 TYPE* (*uint64*)]

lemma *param-uint64*[*param*, *sepref-import-param*]:
(*0*, *0*::*uint64*) ∈ *Id*
(*1*, *1*::*uint64*) ∈ *Id*
by (*rule IdI*)+

abbreviation *uint64-nat-assn* :: *nat* ⇒ *uint64* ⇒ *assn* **where**
‹*uint64-nat-assn* ≡ *pure uint64-nat-rel*›

abbreviation *uint64-assn* :: ‹*uint64* ⇒ *uint64* ⇒ *assn*› **where**
‹*uint64-assn* ≡ *id-assn*›

lemma *op-eq-uint64*:
‹(*uncurry* (*return oo* ((=) :: *uint64* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈
$uint64\text{-}assn^k *_a uint64\text{-}assn^k \rightarrow_a bool\text{-}assn$›
by *sepref-to-hoare sep-auto*

lemma *op-eq-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* ((=) :: *uint64* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a bool\text{-}assn$›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *uint64-nat-assn-zero-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint64-nat*)) ∈ $unit\text{-}assn^k \rightarrow_a uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *uint64-nat-assn-plus*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ $[\lambda(m, n).\ m + n \le uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def nat-of-uint64-add br-def*)

lemma *one-uint64-nat*[*sepref-fr-rules*]:

‹(uncurry0 (return 1), uncurry0 (RETURN one-uint64-nat)) ∈ unit-assn^k $\rightarrow_a$ uint64-nat-assn›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *uint64-nat-assn-less*[*sepref-fr-rules*]:
‹(uncurry (return oo (<)), uncurry (RETURN oo (<))) ∈
uint64-nat-assn^k $*_a$ uint64-nat-assn^k $\rightarrow_a$ bool-assn›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def max-def*
nat-of-uint64-less-iff)

lemma *mult-uint64*[*sepref-fr-rules*]:
‹(uncurry (return oo (∗)), uncurry (RETURN oo (∗)))
∈ uint64-assn^k $*_a$ uint64-assn^k $\rightarrow_a$ uint64-assn›
by *sepref-to-hoare sep-auto*

lemma *shiftr-uint64*[*sepref-fr-rules*]:
‹(uncurry (return oo (>>)), uncurry (RETURN oo (>>)))
∈ uint64-assn^k $*_a$ nat-assn^k $\rightarrow_a$ uint64-assn›
by *sepref-to-hoare sep-auto*

Taken from theory *Native-Word.Uint64*. We use real Word64 instead of the unbounded integer as done by default.

Remark that all this setup is taken from *Native-Word.Uint64*.

code-printing code-module *Uint64* ⇀ (*SML*) ‹(∗ *Test that words can handle numbers between 0 and 63* ∗)
val - = if 6 <= Word.wordSize then () else raise (Fail (wordSize less than 6));

structure Uint64 : sig
eqtype uint64;
val zero : uint64;
val one : uint64;
val fromInt : IntInf.int −> uint64;
val toInt : uint64 −> IntInf.int;
val toFixedInt : uint64 −> Int.int;
val toLarge : uint64 −> LargeWord.word;
val fromLarge : LargeWord.word −> uint64
val fromFixedInt : Int.int −> uint64
val plus : uint64 −> uint64 −> uint64;
val minus : uint64 −> uint64 −> uint64;
val times : uint64 −> uint64 −> uint64;
val divide : uint64 −> uint64 −> uint64;
val modulus : uint64 −> uint64 −> uint64;
val negate : uint64 −> uint64;
val less-eq : uint64 −> uint64 −> bool;
val less : uint64 −> uint64 −> bool;
val notb : uint64 −> uint64;
val andb : uint64 −> uint64 −> uint64;
val orb : uint64 −> uint64 −> uint64;
val xorb : uint64 −> uint64 −> uint64;
val shiftl : uint64 −> IntInf.int −> uint64;
val shiftr : uint64 −> IntInf.int −> uint64;
val shiftr-signed : uint64 −> IntInf.int −> uint64;
val set-bit : uint64 −> IntInf.int −> bool −> uint64;
val test-bit : uint64 −> IntInf.int −> bool;

```
end = struct

type uint64 = Word64.word;

val zero = (0wx0 : uint64);

val one = (0wx1 : uint64);

fun fromInt x = Word64.fromLargeInt (IntInf.toLarge x);

fun toInt x = IntInf.fromLarge (Word64.toLargeInt x);

fun toFixedInt x = Word64.toInt x;

fun fromLarge x = Word64.fromLarge x;

fun fromFixedInt x = Word64.fromInt x;

fun toLarge x = Word64.toLarge x;

fun plus x y = Word64.+(x, y);

fun minus x y = Word64.-(x, y);

fun negate x = Word64.~(x);

fun times x y = Word64.*(x, y);

fun divide x y = Word64.div(x, y);

fun modulus x y = Word64.mod(x, y);

fun less-eq x y = Word64.<=(x, y);

fun less x y = Word64.<(x, y);

fun set-bit x n b =
  let val mask = Word64.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
  in if b then Word64.orb (x, mask)
     else Word64.andb (x, Word64.notb mask)
  end

fun shiftl x n =
  Word64.<< (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr x n =
  Word64.>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr-signed x n =
  Word64.~>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun test-bit x n =
  Word64.andb (x, Word64.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))) <> Word64.fromInt 0

val notb = Word64.notb
```

fun andb x y = Word64.andb(x, y);

fun orb x y = Word64.orb(x, y);

fun xorb x y = Word64.xorb(x, y);

end (∗struct Uint64∗)
›

lemma *bitAND-uint64-max-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*)))
∈ $[\lambda(a, b).\ a \le uint64\text{-}max \wedge b \le uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-plus*
nat-of-uint64-and)

lemma *two-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint64-nat*))
∈ $unit\text{-}assn^k \to_a uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *bitOR-uint64-max-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*)))
∈ $[\lambda(a, b).\ a \le uint64\text{-}max \wedge b \le uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-plus*
nat-of-uint64-or)

lemma *fast-minus-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo fast-minus*), *uncurry* (*RETURN oo fast-minus*))
∈ $[\lambda(a, b).\ a \ge b]_a\ uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to uint64\text{-}nat\text{-}assn$›
by (*sepref-to-hoare*)
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-notle-minus*
nat-of-uint64-less-iff nat-of-uint64-le-iff)

lemma *fast-minus-uint64*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo fast-minus*), *uncurry* (*RETURN oo fast-minus*))
∈ $[\lambda(a, b).\ a \ge b]_a\ uint64\text{-}assn^k *_a uint64\text{-}assn^k \to uint64\text{-}assn$›
by (*sepref-to-hoare*)
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-notle-minus*
nat-of-uint64-less-iff nat-of-uint64-le-iff)

lemma *minus-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (−)), *uncurry* (*RETURN oo* (−))) ∈
$[\lambda(a, b).\ a \ge b]_a\ uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-ge-minus*
nat-of-uint64-le-iff)

lemma *le-uint64-nat-assn-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (≤)), *uncurry* (*RETURN oo* (≤))) ∈ $uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to_a$
bool-assn›
by *sepref-to-hoare*

(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-le-iff*)

lemma *sum-mod-uint64-max-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo sum-mod-uint64-max*))
∈ $\textit{uint64-nat-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow_a \textit{uint64-nat-assn}$›
apply *sepref-to-hoare*
apply (*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-plus*
sum-mod-uint64-max-def)
done

lemma *zero-uint64-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint64*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint64-assn}$›
by *sepref-to-hoare* (*sep-auto simp*: *zero-uint64-def*)

lemma *zero-uint32-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint32*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint32-assn}$›
by *sepref-to-hoare* (*sep-auto simp*: *zero-uint32-def*)

lemma *zero-uin64-hnr*: ‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN 0*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint64-assn}$›
by *sepref-to-hoare sep-auto*

lemma *two-uin64-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint64*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint64-assn}$›
by *sepref-to-hoare* (*sep-auto simp*: *two-uint64-def*)

lemma *two-uint32-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint32*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint32-assn}$›
by *sepref-to-hoare sep-auto*

lemma *sum-uint64-assn*:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ $\textit{uint64-assn}^k *_a \textit{uint64-assn}^k \rightarrow_a \textit{uint64-assn}$›
by (*sepref-to-hoare*) *sep-auto*

lemma *bitAND-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*))) ∈
$\textit{uint64-nat-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow_a \textit{uint64-nat-assn}$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-ao*)

lemma *bitAND-uint64-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*))) ∈
$\textit{uint64-assn}^k *_a \textit{uint64-assn}^k \rightarrow_a \textit{uint64-assn}$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-ao*)

lemma *bitOR-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*))) ∈
$\textit{uint64-nat-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow_a \textit{uint64-nat-assn}$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-ao*)

lemma *bitOR-uint64-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*))) ∈
$\textit{uint64-assn}^k *_a \textit{uint64-assn}^k \rightarrow_a \textit{uint64-assn}$›
by *sepref-to-hoare*

(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-ao*)

lemma *nat-of-uint64-mult-le*:
$\langle$*nat-of-uint64 ai* $*$ *nat-of-uint64 bi* $\leq$ *uint64-max* $\Longrightarrow$
nat-of-uint64 (*ai* $*$ *bi*) = *nat-of-uint64 ai* $*$ *nat-of-uint64 bi*$\rangle$
apply *transfer*
by (*auto simp*: *unat-word-ariths uint64-max-def*)

lemma *uint64-nat-assn-mult*:
$\langle$(*uncurry* (*return oo* ((∗))), *uncurry* (*RETURN oo* ((∗)))) $\in$ $[\lambda(a, b).\ a * b \leq uint64\text{-}max]_a$
$uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to uint64\text{-}nat\text{-}assn$$\rangle$
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-mult-le*)

lemma *uint64-max-uint64-nat-assn*:
$\langle$(*uncurry0* (*return 18446744073709551615*), *uncurry0* (*RETURN uint64-max*)) $\in$
$unit\text{-}assn^k \to_a uint64\text{-}nat\text{-}assn$$\rangle$
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def uint64-max-def*)

lemma *uint64-max-nat-assn*[*sepref-fr-rules*]:
$\langle$(*uncurry0* (*return 18446744073709551615*), *uncurry0* (*RETURN uint64-max*)) $\in$
$unit\text{-}assn^k \to_a nat\text{-}assn$$\rangle$
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def uint64-max-def*)

Conversions

From nat to 64 bits **lemma** *uint64-of-nat-conv-hnr*[*sepref-fr-rules*]:
$\langle$(*return o uint64-of-nat*, *RETURN o uint64-of-nat-conv*) $\in$
$[\lambda n.\ n \leq uint64\text{-}max]_a\ nat\text{-}assn^k \to uint64\text{-}nat\text{-}assn$$\rangle$
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def uint64-of-nat-conv-def nat-of-uint64-uint64-of-nat-id*)

From nat to 32 bits **lemma** *nat-of-uint32-spec-hnr*[*sepref-fr-rules*]:
$\langle$(*return o uint32-of-nat*, *RETURN o nat-of-uint32-spec*) $\in$
$[\lambda n.\ n \leq uint32\text{-}max]_a\ nat\text{-}assn^k \to uint32\text{-}nat\text{-}assn$$\rangle$
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-uint32-of-nat-id*)

From 64 to nat bits **lemma** *nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
$\langle$(*return o nat-of-uint64*, *RETURN o nat-of-uint64-conv*) $\in$ $uint64\text{-}nat\text{-}assn^k \to_a nat\text{-}assn$$\rangle$
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *nat-of-uint64*[*sepref-fr-rules*]:
$\langle$(*return o nat-of-uint64*, *RETURN o nat-of-uint64*) $\in$
$(uint64\text{-}assn)^k \to_a nat\text{-}assn$$\rangle$
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*
nat-of-uint64-def
split: *option.splits*)

From 32 to nat bits **lemma** *nat-of-uint32-conv-hnr*[*sepref-fr-rules*]:
$\langle$(*return o nat-of-uint32*, *RETURN o nat-of-uint32-conv*) $\in$ $uint32\text{-}nat\text{-}assn^k \to_a nat\text{-}assn$$\rangle$
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def nat-of-uint32-conv-def*)

lemma *convert-to-uint32-hnr*[*sepref-fr-rules*]:
$\langle$(*return o uint32-of-nat*, *RETURN o convert-to-uint32*)

$\in [\lambda n.\ n \le \mathit{uint32\text{-}max}]_a\ \mathit{nat\text{-}assn}^k \rightarrow \mathit{uint32\text{-}nat\text{-}assn}$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def uint32-max-def nat-of-uint32-uint32-of-nat-id*)

From 32 to 64 bits **lemma** *uint64-of-uint32-hnr*[*sepref-fr-rules*]:
‹(*return o uint64-of-uint32*, *RETURN o uint64-of-uint32*) $\in$ *uint32-assn*^{k} $\rightarrow_a$ *uint64-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *br-def*)

lemma *uint64-of-uint32-conv-hnr*[*sepref-fr-rules*]:
‹(*return o uint64-of-uint32*, *RETURN o uint64-of-uint32-conv*) $\in$
uint32-nat-assn^k $\rightarrow_a$ *uint64-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *br-def uint32-nat-rel-def uint64-nat-rel-def*
nat-of-uint32-code nat-of-uint64-uint64-of-uint32)

From 64 to 32 bits **lemma** *uint32-of-uint64-conv-hnr*[*sepref-fr-rules*]:
‹(*return o uint32-of-uint64*, *RETURN o uint32-of-uint64-conv*) $\in$
$[\lambda a.\ a \le \mathit{uint32\text{-}max}]_a\ \mathit{uint64\text{-}nat\text{-}assn}^k \rightarrow \mathit{uint32\text{-}nat\text{-}assn}$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-of-uint64-def uint32-nat-rel-def br-def nat-of-uint64-le-iff*
nat-of-uint32-uint32-of-nat-id uint64-nat-rel-def)

From nat to 32 bits **lemma** (**in** $-$) *uint32-of-nat*[*sepref-fr-rules*]:
‹(*return o uint32-of-nat*, *RETURN o uint32-of-nat*) $\in [\lambda n.\ n \le \mathit{uint32\text{-}max}]_a\ \mathit{nat\text{-}assn}^k \rightarrow \mathit{uint32\text{-}assn}$›
by *sepref-to-hoare sep-auto*

Setup for numerals The refinement framework still defaults to *nat*, making the constants like *two-uint32-nat* still useful, but they can be omitted in some cases: For example, in $(2::'a) + n$, 2 will be refined to *nat* (independently of n). However, if the expression is $n + (2::'a)$ and if n is refined to *uint32*, then everything will work as one might expect.

lemmas [*id-rules*] =
itypeI[*Pure.of numeral TYPE* (*num* $\Rightarrow$ *uint32*)]
itypeI[*Pure.of numeral TYPE* (*num* $\Rightarrow$ *uint64*)]

lemma *id-uint32-const*[*id-rules*]: (*PR-CONST* (*a*::*uint32*)) $::_i$ *TYPE*(*uint32*) **by** *simp*
lemma *id-uint64-const*[*id-rules*]: (*PR-CONST* (*a*::*uint64*)) $::_i$ *TYPE*(*uint64*) **by** *simp*

lemma *param-uint32-numeral*[*sepref-import-param*]:
‹(*numeral n*, *numeral n*) $\in$ *uint32-rel*›
by *auto*

lemma *param-uint64-numeral*[*sepref-import-param*]:
‹(*numeral n*, *numeral n*) $\in$ *uint64-rel*›
by *auto*

locale *nat-of-uint64-loc* =
fixes *n* :: *num*
assumes *le-uint64-max*: ‹*numeral n* $\le$ *uint64-max*›
begin

definition *nat-of-uint64-numeral* :: *nat* **where**
[*simp*]: ‹*nat-of-uint64-numeral* = (*numeral n*)›

definition *nat-of-uint64* :: *uint64* **where**
[*simp*]: ‹*nat-of-uint64* = (*numeral n*)›

lemma *nat-of-uint64-numeral-hnr*:
‹(*uncurry0* (*return nat-of-uint64*), *uncurry0* (*PR-CONST* (*RETURN nat-of-uint64-numeral*)))
∈ *unit-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
using *le-uint64-max*
by (*sepref-to-hoare*; *sep-auto simp*: *uint64-nat-rel-def br-def uint64-max-def*)
sepref-register *nat-of-uint64-numeral*
end

lemma (**in** −) [*sepref-fr-rules*]:
‹*CONSTRAINT* (λ*n*. *numeral n* ≤ *uint64-max*) *n* ⟹
(*uncurry0* (*return* (*nat-of-uint64-loc.nat-of-uint64 n*)),
uncurry0 (*RETURN* (*PR-CONST* (*nat-of-uint64-loc.nat-of-uint64-numeral n*))))
∈ *unit-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
using *nat-of-uint64-loc.nat-of-uint64-numeral-hnr*[*of n*]
by (*auto simp*: *nat-of-uint64-loc-def*)

lemma *uint32-max-uint32-nat-assn*:
‹(*uncurry0* (*return 4294967295*), *uncurry0* (*RETURN uint32-max*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-max-def uint32-nat-rel-def br-def*)

lemma *minus-uint64-assn*:
‹(*uncurry* (*return oo* (−)), *uncurry* (*RETURN oo* (−))) ∈ *uint64-assn*^{k} $*_a$ *uint64-assn*^{k} $\rightarrow_a$ *uint64-assn*›
by *sepref-to-hoare sep-auto*

lemma *uint32-of-nat-uint32-nat-assn*[*sepref-fr-rules*]:
‹(*return o id*, *RETURN o uint32-of-nat*) ∈ *uint32-nat-assn*^{k} $\rightarrow_a$ *uint32-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *uint32-of-nat2*[*sepref-fr-rules*]:
‹(*return o uint32-of-uint64*, *RETURN o uint32-of-nat*) ∈
[λ*n*. *n* ≤ *uint32-max*]$_a$ *uint64-nat-assn*^{k} → *uint32-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def uint64-nat-rel-def uint32-of-uint64-def*)

lemma *three-uint32-hnr*:
‹(*uncurry0* (*return 3*), *uncurry0* (*RETURN* (*three-uint32* :: *uint32*))) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def three-uint32-def*)

lemma *nat-of-uint64-id-conv-hnr*[*sepref-fr-rules*]:
‹(*return o id*, *RETURN o nat-of-uint64-id-conv*) ∈ *uint64-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *nat-of-uint64-id-conv-def uint64-nat-rel-def br-def*)

end
theory *Array-UInt*
imports *Array-List-Array WB-Word-Assn WB-More-Refinement-List*
begin

hide-const *Autoref-Fix-Rel.CONSTRAINT*

lemma *convert-fref*:
  WB-More-Refinement.fref = *Sepref-Rules.fref*
  WB-More-Refinement.freft = *Sepref-Rules.freft*
  unfolding *WB-More-Refinement.fref-def Sepref-Rules.fref-def*
  by *auto*

0.1.6 More about general arrays

This function does not resize the array: this makes sense for our purpose, but may be not in general.

definition *butlast-arl* **where**
  ‹*butlast-arl* = (λ(*xs*, *i*). (*xs*, *fast-minus i 1*))›

lemma *butlast-arl-hnr*[*sepref-fr-rules*]:
  ‹(*return o butlast-arl*, *RETURN o butlast*) ∈ $[\lambda xs.\ xs \neq []]_a$ (*arl-assn A*)^d → *arl-assn A*›
proof −
  have [*simp*]: ‹*b* ≤ *length l′* ⟹ (*take b l′*, *x*) ∈ ⟨*the-pure A*⟩*list-rel* ⟹
    (*take* (*b* − *Suc 0*) *l′*, *take* (*length x* − *Suc 0*) *x*) ∈ ⟨*the-pure A*⟩*list-rel*›
    for *b l′ x*
    using *list-rel-take*[*of* ‹*take b l′*› *x* ‹*the-pure A*› ‹*b* −*1*›]
    by (*auto simp*: *list-rel-imp-same-length*[*symmetric*]
      butlast-conv-take min-def
      simp del: *take-butlast-conv*)
  show *?thesis*
    by *sepref-to-hoare*
      (*sep-auto simp*: *butlast-arl-def arl-assn-def hr-comp-def is-array-list-def*
        butlast-conv-take
       simp del: *take-butlast-conv*)
qed

0.1.7 Setup for array accesses via unsigned integer

NB: not all code printing equation are defined here, but this is needed to use the (more efficient) array operation by avoid the conversions back and forth to infinite integer.

Getters (Array accesses)

32-bit unsigned integers **definition** *nth-aa-u* **where**
  ‹*nth-aa-u x L L′* = *nth-aa x* (*nat-of-uint32 L*) *L′*›

definition *nth-aa′* **where**
  ‹*nth-aa′ xs i j* = *do* {
    x ← *Array.nth′ xs i*;
    y ← *arl-get x j*;
    return y}›

lemma *nth-aa-u*[*code*]:
  ‹*nth-aa-u x L L′* = *nth-aa′ x* (*integer-of-uint32 L*) *L′*›
  unfolding *nth-aa-u-def nth-aa′-def nth-aa-def Array.nth′-def nat-of-uint32-code*
  by *auto*

lemma *nth-aa-uint-hnr*[*sepref-fr-rules*]:
  fixes *R* :: ‹- ⇒ - ⇒ *assn*›

assumes ‹*CONSTRAINT Sepref-Basic.is-pure R*›
shows
‹(*uncurry2 nth-aa-u*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
[λ((*x*, *L*), *L'*). *L* < *length x* ∧ *L'* < *length* (*x* ! *L*)]$_a$
(*arrayO-assn* (*arl-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *nat-assn*^{k} → *R*›
unfolding *nth-aa-u-def*
by *sepref-to-hoare*
(*use assms* **in** ‹*sep-auto simp*: *uint32-nat-rel-def br-def length-ll-def nth-ll-def nth-rll-def*›)

definition *nth-raa-u* **where**
‹*nth-raa-u x L* = *nth-raa x* (*nat-of-uint32 L*)›

lemma *nth-raa-uint-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 nth-raa-u*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
[λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
(*arlO-assn* (*array-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *nat-assn*^{k} → *R*›
unfolding *nth-raa-u-def*
supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
using *assms*
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def*)

lemma *array-replicate-custom-hnr-u*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure A* ⟹
(*uncurry* (λ*n*. *Array.new* (*nat-of-uint32 n*)), *uncurry* (*RETURN* ∘∘ *op-array-replicate*)) ∈
uint32-nat-assn^k $*_a$ *A*^{k} $\rightarrow_a$ *array-assn A*›
using *array-replicate-custom-hnr*[*of A*]
unfolding *hfref-def*
by (*sep-auto simp*: *uint32-nat-assn-nat-assn-nat-of-uint32*)

definition *nth-u* **where**
‹*nth-u xs n* = *nth xs* (*nat-of-uint32 n*)›

definition *nth-u-code* **where**
‹*nth-u-code xs n* = *Array.nth' xs* (*integer-of-uint32 n*)›

lemma *nth-u-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry nth-u-code*, *uncurry* (*RETURN oo nth-u*)) ∈
[λ(*xs*, *n*). *nat-of-uint32 n* < *length xs*]$_a$ (*array-assn A*)^k $*_a$ *uint32-assn*^{k} → *A*›
proof –
obtain *A'* **where**
A: ‹*pure A'* = *A*›
using *assms pure-the-pure* **by** *auto*
then have *A'*: ‹*the-pure A* = *A'*›
by *auto*
have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *A'*)) = *A'*›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update param-nth A' A[*symmetric*] *ent-refl-true*
list-rel-eq-listrel listrel-iff-nth pure-def nth-u-code-def nth-u-def Array.nth'-def

nat-of-uint32-code)
qed

lemma *array-get-hnr-u*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry nth-u-code*,
uncurry (*RETURN* ∘∘ *op-list-get*)) ∈ [*pre-list-get*]$_a$ (*array-assn A*)^k $*_a$ *uint32-nat-assn*^{k} → *A*›
proof −
obtain *A'* **where**
A: ‹*pure A'* = *A*›
using *assms pure-the-pure* **by** *auto*
then have *A'*: ‹*the-pure A* = *A'*›
by *auto*
have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *A'*)) = *A'*›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update param-nth A' A[*symmetric*] *ent-refl-true*
list-rel-eq-listrel listrel-iff-nth pure-def nth-u-code-def Array.nth'-def
nat-of-uint32-code)
qed

definition *arl-get'* :: *'a*::*heap array-list* ⇒ *integer* ⇒ *'a Heap* **where**
[*code del*]: *arl-get' a i* = *arl-get a* (*nat-of-integer i*)

definition *arl-get-u* :: *'a*::*heap array-list* ⇒ *uint32* ⇒ *'a Heap* **where**
arl-get-u ≡ λ*a i*. *arl-get' a* (*integer-of-uint32 i*)

lemma *arrayO-arl-get-u-rule*[*sep-heap-rules*]:
assumes *i*: ‹*i* < *length a*› **and** ‹(*i'* , *i*) ∈ *uint32-nat-rel*›
shows ‹<*arlO-assn* (*array-assn R*) *a ai*> *arl-get-u ai i'* <λ*r*. *arlO-assn-except* (*array-assn R*) [*i*] *a ai*
(λ*r'*. *array-assn R* (*a* ! *i*) *r* ∗ ↑(*r* = *r'* ! *i*))>›
using *assms*
by (*sep-auto simp*: *arl-get-u-def arl-get'-def nat-of-uint32-code*[*symmetric*]
uint32-nat-rel-def br-def)

definition *arl-get-u'* **where**
[*symmetric*, *code*]: ‹*arl-get-u'* = *arl-get-u*›

code-printing constant *arl-get-u'* ⇀ (*SML*) (*fn*/ ()/ =>/ *Array.sub*/ (*fst* (-),/ *Word32.toInt* (-)))

lemma *arl-get'-nth'*[*code*]: ‹*arl-get'* = (λ(*a*, *n*). *Array.nth' a*)›
unfolding *arl-get-def arl-get'-def Array.nth'-def*
by (*intro ext*) *auto*

lemma *arl-get-hnr-u*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry arl-get-u*, *uncurry* (*RETURN* ∘∘ *op-list-get*))
∈ [*pre-list-get*]$_a$ (*arl-assn A*)^k $*_a$ *uint32-nat-assn*^{k} → *A*›
proof −
obtain *A'* **where**
A: ‹*pure A'* = *A*›
using *assms pure-the-pure* **by** *auto*

then have A': ‹*the-pure* $A = A'$›
  by *auto*
 have [*simp*]: ‹*the-pure* ($\lambda a\ c.\ \uparrow ((c,\ a) \in A')) = A'$›
  unfolding *pure-def*[*symmetric*] **by** *auto*
 show *?thesis*
  by *sepref-to-hoare*
   (*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def hr-comp-def list-rel-pres-length list-rel-update param-nth arl-assn-def* A' A[*symmetric*] *pure-def arl-get-u-def Array.nth'-def arl-get'-def nat-of-uint32-code*[*symmetric*])
qed

definition *nth-rll-nu* **where**
 ‹*nth-rll-nu* = *nth-rll*›

definition *nth-raa-u'* **where**
 ‹*nth-raa-u'* *xs x L* = *nth-raa xs x* (*nat-of-uint32 L*)›

lemma *nth-raa-u'-uint-hnr*[*sepref-fr-rules*]:
 assumes *p*: ‹*is-pure R*›
 shows
  ‹(*uncurry2 nth-raa-u'*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
   $[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
   $(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow R$›
 unfolding *nth-raa-u-def*
 supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
 using *assms*
 by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def nth-raa-u'-def*)

lemma *nth-nat-of-uint32-nth'*: ‹*Array.nth x* (*nat-of-uint32 L*) = *Array.nth' x* (*integer-of-uint32 L*)›
 by (*auto simp*: *Array.nth'-def nat-of-uint32-code*)

lemma *nth-aa-u-code*[*code*]:
 ‹*nth-aa-u x L L'* = *nth-u-code x L* ≫= (λ*x*. *arl-get x L'* ≫= *return*)›
 unfolding *nth-aa-u-def nth-aa-def arl-get-u-def*[*symmetric*] *Array.nth'-def*[*symmetric*]
  nth-nat-of-uint32-nth' nth-u-code-def[*symmetric*] **..**

definition *nth-aa-i64-u32* **where**
 ‹*nth-aa-i64-u32 xs x L* = *nth-aa xs* (*nat-of-uint64 x*) (*nat-of-uint32 L*)›

lemma *nth-aa-i64-u32-hnr*[*sepref-fr-rules*]:
 assumes *p*: ‹*is-pure R*›
 shows
  ‹(*uncurry2 nth-aa-i64-u32*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
   $[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
   $(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow R$›
 unfolding *nth-aa-i64-u32-def*
 supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
 using *assms*
 by *sepref-to-hoare*
  (*sep-auto simp*: *uint32-nat-rel-def br-def nth-raa-u'-def uint64-nat-rel-def length-rll-def length-ll-def nth-rll-def nth-ll-def*)

definition *nth-aa-i64-u64* **where**
 ‹*nth-aa-i64-u64 xs x L* = *nth-aa xs* (*nat-of-uint64 x*) (*nat-of-uint64 L*)›

lemma *nth-aa-i64-u64-hnr*[*sepref-fr-rules*]:
  assumes *p*: ‹*is-pure R*›
  shows
    ‹(*uncurry2 nth-aa-i64-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
      [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
      (*arrayO-assn* (*arl-assn R*))^k $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
  unfolding *nth-aa-i64-u64-def*
  supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
  using *assms*
  by *sepref-to-hoare*
    (*sep-auto simp*: *br-def nth-raa-u'-def uint64-nat-rel-def*
      length-rll-def length-ll-def nth-rll-def nth-ll-def)

definition *nth-aa-i32-u64* **where**
  ‹*nth-aa-i32-u64 xs x L* = *nth-aa xs* (*nat-of-uint32 x*) (*nat-of-uint64 L*)›

lemma *nth-aa-i32-u64-hnr*[*sepref-fr-rules*]:
  assumes *p*: ‹*is-pure R*›
  shows
    ‹(*uncurry2 nth-aa-i32-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
      [λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$
      (*arrayO-assn* (*arl-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
  unfolding *nth-aa-i32-u64-def*
  supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
  using *assms*
  by *sepref-to-hoare*
    (*sep-auto simp*: *uint32-nat-rel-def br-def nth-raa-u'-def uint64-nat-rel-def*
      length-rll-def length-ll-def nth-rll-def nth-ll-def)

64-bit unsigned integers **definition** *nth-u64* **where**
  ‹*nth-u64 xs n* = *nth xs* (*nat-of-uint64 n*)›

definition *nth-u64-code* **where**
  ‹*nth-u64-code xs n* = *Array.nth' xs* (*integer-of-uint64 n*)›

lemma *nth-u64-hnr*[*sepref-fr-rules*]:
  assumes ‹*CONSTRAINT is-pure A*›
  shows ‹(*uncurry nth-u64-code*, *uncurry* (*RETURN oo nth-u64*)) ∈
    [λ(*xs*, *n*). *nat-of-uint64 n* < *length xs*]$_a$ (*array-assn A*)^k $*_a$ *uint64-assn*^{k} → *A*›
proof −
  obtain *A'* **where**
    A: ‹*pure A'* = *A*›
    using *assms pure-the-pure* **by** *auto*
  then have *A'*: ‹*the-pure A* = *A'*›
    by *auto*
  have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *A'*)) = *A'*›
    unfolding *pure-def*[*symmetric*] **by** *auto*
  show *?thesis*
    by *sepref-to-hoare*
      (*sep-auto simp*: *array-assn-def is-array-def*
        hr-comp-def list-rel-pres-length list-rel-update param-nth A' A[*symmetric*] *ent-refl-true*
        list-rel-eq-listrel listrel-iff-nth pure-def nth-u64-code-def Array.nth'-def
        nat-of-uint64-code nth-u64-def)
qed

lemma *array-get-hnr-u64*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry nth-u64-code*,
uncurry (*RETURN* ∘∘ *op-list-get*)) ∈ $[pre\text{-}list\text{-}get]_a$ $(array\text{-}assn\ A)^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow A$›
proof −
obtain A' **where**
A: ‹*pure* $A' = A$›
using *assms pure-the-pure* **by** *auto*
then have A': ‹*the-pure* $A = A'$›
by *auto*
have [*simp*]: ‹*the-pure* ($\lambda a\ c.\ \uparrow ((c, a) \in A')) = A'$›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def hr-comp-def list-rel-pres-length list-rel-update param-nth A' A*[*symmetric*] *ent-refl-true list-rel-eq-listrel listrel-iff-nth pure-def nth-u64-code-def Array.nth′-def nat-of-uint64-code*)
qed

Setters

32-bits **definition** *heap-array-set′-u* **where**
‹*heap-array-set′-u a i x* = *Array.upd′ a* (*integer-of-uint32 i*) *x*›

definition *heap-array-set-u* **where**
‹*heap-array-set-u a i x* = *heap-array-set′-u a i x* ≫ *return a*›

lemma *array-set-hnr-u*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure A* ⟹
(*uncurry2 heap-array-set-u*, *uncurry2* (*RETURN* ∘∘∘ *op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(array\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \rightarrow array\text{-}assn\ A$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def hr-comp-def list-rel-pres-length list-rel-update heap-array-set′-u-def heap-array-set-u-def Array.upd′-def nat-of-uint32-code*[*symmetric*])

definition *update-aa-u* **where**
‹*update-aa-u xs i j* = *update-aa xs* (*nat-of-uint32 i*) *j*›

lemma *Array-upd-upd′*: ‹*Array.upd i x a* = *Array.upd′ a* (*of-nat i*) *x* ≫ *return a*›
by (*auto simp*: *Array.upd′-def upd-return*)

definition *Array-upd-u* **where**
‹*Array-upd-u i x a* = *Array.upd* (*nat-of-uint32 i*) *x a*›

lemma *Array-upd-u-code*[*code*]: ‹*Array-upd-u i x a* = *heap-array-set′-u a i x* ≫ *return a*›
unfolding *Array-upd-u-def heap-array-set′-u-def*
Array.upd′-def
by (*auto simp*: *nat-of-uint32-code upd-return*)

lemma *update-aa-u-code*[*code*]:
‹*update-aa-u a i j y* = *do* {
x ← *nth-u-code a i*;

$a' \leftarrow$ *arl-set x j y*;
Array-upd-u i a' a
}›
unfolding *update-aa-u-def update-aa-def nth-nat-of-uint32-nth' nth-nat-of-uint32-nth'*
arl-get-u-def[*symmetric*] *nth-u-code-def*[*symmetric*]
heap-array-set'-u-def[*symmetric*] *Array-upd-u-def*[*symmetric*]
by *auto*

definition *arl-set'-u* **where**
‹*arl-set'-u a i x = arl-set a* (*nat-of-uint32 i*) *x*›

definition *arl-set-u* :: ‹*'a::heap array-list ⇒ uint32 ⇒ 'a ⇒ 'a array-list Heap*›**where**
‹*arl-set-u a i x = arl-set'-u a i x*›

lemma *arl-set-hnr-u*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure A* $\Longrightarrow$
(*uncurry2 arl-set-u*, *uncurry2* (*RETURN* ∘∘∘ *op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(arl\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \rightarrow arl\text{-}assn\ A$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update heap-array-set'-u-def
heap-array-set-u-def Array.upd'-def arl-set-u-def arl-set'-u-def arl-assn-def
nat-of-uint32-code[*symmetric*])

64-bits **definition** *heap-array-set'-u64* **where**
‹*heap-array-set'-u64 a i x = Array.upd' a* (*integer-of-uint64 i*) *x*›

definition *heap-array-set-u64* **where**
‹*heap-array-set-u64 a i x = heap-array-set'-u64 a i x* ≫ *return a*›

lemma *array-set-hnr-u64*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure A* $\Longrightarrow$
(*uncurry2 heap-array-set-u64*, *uncurry2* (*RETURN* ∘∘∘ *op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(array\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a A^k \rightarrow array\text{-}assn\ A$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update heap-array-set'-u64-def
heap-array-set-u64-def Array.upd'-def
nat-of-uint64-code[*symmetric*])

definition *arl-set'-u64* **where**
‹*arl-set'-u64 a i x = arl-set a* (*nat-of-uint64 i*) *x*›

definition *arl-set-u64* :: ‹*'a::heap array-list ⇒ uint64 ⇒ 'a ⇒ 'a array-list Heap*›**where**
‹*arl-set-u64 a i x = arl-set'-u64 a i x*›

lemma *arl-set-hnr-u64*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure A* $\Longrightarrow$
(*uncurry2 arl-set-u64*, *uncurry2* (*RETURN* ∘∘∘ *op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(arl\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a A^k \rightarrow arl\text{-}assn\ A$›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
hr-comp-def list-rel-pres-length list-rel-update heap-array-set'-u-def
heap-array-set-u-def Array.upd'-def arl-set-u64-def arl-set'-u64-def arl-assn-def
nat-of-uint64-code[*symmetric*])

lemma *nth-nat-of-uint64-nth'*: ‹*Array.nth x* (*nat-of-uint64 L*) = *Array.nth' x* (*integer-of-uint64 L*)›
  by (*auto simp*: *Array.nth'-def nat-of-uint64-code*)

definition *nth-raa-i-u64* **where**
  ‹*nth-raa-i-u64 x L L'* = *nth-raa x L* (*nat-of-uint64 L'*)›

lemma *nth-raa-i-uint64-hnr*[*sepref-fr-rules*]:
  assumes *p*: ‹*is-pure R*›
  shows
    ‹(*uncurry2 nth-raa-i-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
      $[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
      $(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
  unfolding *nth-raa-i-u64-def*
  supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
  using *assms*
  by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

definition *arl-get-u64* :: *'a::heap array-list* ⇒ *uint64* ⇒ *'a Heap* **where**
  arl-get-u64 ≡ λ*a i*. *arl-get' a* (*integer-of-uint64 i*)

lemma *arl-get-hnr-u64*[*sepref-fr-rules*]:
  assumes ‹*CONSTRAINT is-pure A*›
  shows ‹(*uncurry arl-get-u64*, *uncurry* (*RETURN* ∘∘ *op-list-get*))
    $\in [pre\text{-}list\text{-}get]_a\ (arl\text{-}assn\ A)^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow A$›
proof −
  obtain *A'* **where**
    A: ‹*pure A'* = *A*›
    using *assms pure-the-pure* **by** *auto*
  then have *A'*: ‹*the-pure A* = *A'*›
    by *auto*
  have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *A'*)) = *A'*›
    unfolding *pure-def*[*symmetric*] **by** *auto*
  show *?thesis*
    by *sepref-to-hoare*
      (*sep-auto simp*: *uint64-nat-rel-def br-def ex-assn-up-eq2 array-assn-def is-array-def*
        hr-comp-def list-rel-pres-length list-rel-update param-nth arl-assn-def
        A' A[*symmetric*] *pure-def arl-get-u64-def Array.nth'-def arl-get'-def*
        nat-of-uint64-code[*symmetric*])
qed

definition *nth-raa-u64'* **where**
  ‹*nth-raa-u64' xs x L* = *nth-raa xs x* (*nat-of-uint64 L*)›

lemma *nth-raa-u64'-uint-hnr*[*sepref-fr-rules*]:
  assumes *p*: ‹*is-pure R*›
  shows
    ‹(*uncurry2 nth-raa-u64'*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
      $[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
      $(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
  supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
  using *assms*
  by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def nth-raa-u64'-def*)

definition *nth-raa-u64* **where**
⟨*nth-raa-u64 x L* = *nth-raa x* (*nat-of-uint64 L*)⟩

lemma *nth-raa-uint64-hnr*[*sepref-fr-rules*]:
assumes *p*: ⟨*is-pure R*⟩
shows
⟨(*uncurry2 nth-raa-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \to R$⟩
unfolding *nth-raa-u64-def*
supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
using *assms*
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

definition *nth-raa-u64-u64* **where**
⟨*nth-raa-u64-u64 x L L′* = *nth-raa x* (*nat-of-uint64 L*) (*nat-of-uint64 L′*)⟩

lemma *nth-raa-uint64-uint64-hnr*[*sepref-fr-rules*]:
assumes *p*: ⟨*is-pure R*⟩
shows
⟨(*uncurry2 nth-raa-u64-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to R$⟩
unfolding *nth-raa-u64-u64-def*
supply *nth-aa-hnr*[*to-hnr*, *sep-heap-rules*]
using *assms*
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *heap-array-set-u64-upd*:
⟨*heap-array-set-u64 x j xi* = *Array.upd* (*nat-of-uint64 j*) *xi x* ⤜ (λ*xa. return x*) ⟩
by (*auto simp*: *heap-array-set-u64-def heap-array-set′-u64-def*
Array.upd′-def nat-of-uint64-code[*symmetric*])

Append (32 bit integers only)

definition *append-el-aa-u′* :: (′*a*::{*default*,*heap*} *array-list*) *array* ⇒
uint32 ⇒ ′*a* ⇒ (′*a array-list*) *array Heap***where**
append-el-aa-u′ ≡ λ*a i x*.
Array.nth′ a (*integer-of-uint32 i*) ⤜
(λ*j*. *arl-append j x* ⤜
(λ*a′*. *Array.upd′ a* (*integer-of-uint32 i*) *a′* ⤜ (λ-. *return a*)))

lemma *append-el-aa-append-el-aa-u′*:
⟨*append-el-aa xs* (*nat-of-uint32 i*) *j* = *append-el-aa-u′ xs i j*⟩
unfolding *append-el-aa-def append-el-aa-u′-def Array.nth′-def nat-of-uint32-code Array.upd′-def*
by (*auto simp add*: *upd′-def upd-return max-def*)

lemma *append-aa-hnr-u*:
fixes *R* :: ⟨′*a* ⇒ ′*b* :: {*heap*, *default*} ⇒ *assn*⟩
assumes *p*: ⟨*is-pure R*⟩
shows

‹(*uncurry2* (λ*xs i. append-el-aa xs* (*nat-of-uint32 i*)), *uncurry2* (*RETURN* ∘∘∘ (λ*xs i. append-ll xs* (*nat-of-uint32 i*)))) ∈
[λ((*l*,*i*),*x*). *nat-of-uint32 i* < *length l*]$_a$ (*arrayO-assn* (*arl-assn R*))^d $*_a$ *uint32-assn*^{k} $*_a$ R^k → (*arrayO-assn* (*arl-assn R*))›

proof −
obtain R' **where** *R*: ‹*the-pure R* = R'› **and** R': ‹*R* = *pure* R'›
using *p* **by** *fastforce*
have [*simp*]: ‹($\exists_A$*x*. *arrayO-assn* (*arl-assn R*) *a ai* ∗ *R x r* ∗ *true* ∗ ↑ (*x* = *a* ! *ba* ! *b*)) =
(*arrayO-assn* (*arl-assn R*) *a ai* ∗ *R* (*a* ! *ba* ! *b*) *r* ∗ *true*)› **for** *a ai ba b r*
by (*auto simp*: *ex-assn-def*)
show *?thesis* — TODO tune proof
apply *sepref-to-hoare*
apply (*sep-auto simp*: *append-el-aa-def uint32-nat-rel-def br-def*)
apply (*simp add*: *arrayO-except-assn-def*)
apply (*rule sep-auto-is-stupid*[*OF p*])
apply (*sep-auto simp*: *array-assn-def is-array-def append-ll-def*)
apply (*simp add*: *arrayO-except-assn-array0*[*symmetric*] *arrayO-except-assn-def*)
apply (*subst-tac* (*2*) *i* = ‹*nat-of-uint32 ba*› **in** *heap-list-all-nth-remove1*)
apply (*solves* ‹*simp*›)
apply (*simp add*: *array-assn-def is-array-def*)
apply (*rule-tac x*=‹*p*[*nat-of-uint32 ba* := (*ab*, *bc*)]› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)[*2*]
apply (*auto simp*: *star-aci*)
done
qed

lemma *append-el-aa-hnr'*[*sepref-fr-rules*]:
shows ‹(*uncurry2 append-el-aa-u'*, *uncurry2* (*RETURN ooo append-ll*))
∈ [λ((*W*,*L*), *j*). *L* < *length W*]$_a$
(*arrayO-assn* (*arl-assn nat-assn*))^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *nat-assn*^{k} → (*arrayO-assn* (*arl-assn nat-assn*))›
(**is** ‹*?a* ∈ [*?pre*]$_a$ *?init* → *?post*›)
using *append-aa-hnr-u*[*of nat-assn*, *simplified*] **unfolding** *hfref-def uint32-nat-rel-def br-def pure-def hn-refine-def append-el-aa-append-el-aa-u'*
by *auto*

lemma *append-el-aa-uint32-hnr'*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows ‹(*uncurry2 append-el-aa-u'*, *uncurry2* (*RETURN ooo append-ll*))
∈ [λ((*W*,*L*), *j*). *L* < *length W*]$_a$
(*arrayO-assn* (*arl-assn R*))^d $*_a$ *uint32-nat-assn*^{k} $*_a$ R^k →
(*arrayO-assn* (*arl-assn R*))›
(**is** ‹*?a* ∈ [*?pre*]$_a$ *?init* → *?post*›)
using *append-aa-hnr-u*[*of R*, *simplified*] *assms*
unfolding *hfref-def uint32-nat-rel-def br-def pure-def hn-refine-def append-el-aa-append-el-aa-u'*
by *auto*

lemma *append-el-aa-u'-code*[*code*]:
append-el-aa-u' = (λ*a i x*. *nth-u-code a i* ⤜
(λ*j*. *arl-append j x* ⤜
(λ*a'*. *heap-array-set'-u a i a'* ⤜ (λ-. *return a*))))
unfolding *append-el-aa-u'-def nth-u-code-def heap-array-set'-u-def*
by *auto*

definition *update-raa-u32* **where**
‹*update-raa-u32 a i j y = do* {
  $x \leftarrow$ *arl-get-u a i*;
  Array.upd j y x $\gg=$ *arl-set-u a i*
}›

lemma *update-raa-u32-rule*[*sep-heap-rules*]:
  assumes *p*: ‹*is-pure R*› **and** ‹$bb <$ *length a*› **and** ‹$ba <$ *length-rll a bb*› **and**
    ‹$(bb', bb) \in$ *uint32-nat-rel*›
  shows ‹<*R b bi* $*$ *arlO-assn* (*array-assn R*) *a ai*> *update-raa-u32 ai bb′ ba bi*
    <λr. *R b bi* $*$ ($\exists_A x$. *arlO-assn* (*array-assn R*) *x r* $* \uparrow$ ($x =$ *update-rll a bb ba b*))>$_t$›
  using *assms*
  apply (*cases ai*)
  apply (*sep-auto simp add*: *update-raa-u32-def update-rll-def p*)
  apply (*sep-auto simp add*: *update-raa-u32-def arlO-assn-except-def array-assn-def hr-comp-def*
    arl-assn-def arl-set-u-def arl-set′-u-def)
   apply (*solves* ‹*simp add*: *br-def uint32-nat-rel-def*›)
  apply (*rule-tac x*=‹*a*[*bb* := (*a* ! *bb*)[*ba* := *b*]]› **in** *ent-ex-postI*)
  apply (*subst-tac i*=*bb* **in** *arlO-assn-except-array0-index*[*symmetric*])
  apply (*auto simp add*: *br-def uint32-nat-rel-def*)[]

  apply (*auto simp add*: *update-raa-def arlO-assn-except-def array-assn-def is-array-def hr-comp-def*)
  apply (*rule-tac x*=‹*p*[*bb* := *xa*]› **in** *ent-ex-postI*)
  apply (*rule-tac x*=‹*baa*› **in** *ent-ex-postI*)
  apply (*subst-tac* (*2*)*xs′*=*a* **and** *ys′*=*p* **in** *heap-list-all-nth-cong*)
    apply (*solves* ‹*auto*›)
   apply (*solves* ‹*auto*›)
  by (*sep-auto simp*: *arl-assn-def uint32-nat-rel-def br-def*)

lemma *update-raa-u32-hnr*[*sepref-fr-rules*]:
  assumes ‹*is-pure R*›
  shows ‹(*uncurry3 update-raa-u32*, *uncurry3* (*RETURN oooo update-rll*)) $\in$
    $[\lambda(((l,i), j), x).\ i <$ *length l* $\wedge\ j <$ *length-rll l i*$]_a$ (*arlO-assn* (*array-assn R*))$^d *_a$ *uint32-nat-assn*^{k}
$*_a$ *nat-assn*^{k} $*_a$ $R^k \rightarrow$ (*arlO-assn* (*array-assn R*))›
  by *sepref-to-hoare* (*sep-auto simp*: *assms*)

lemma *update-aa-u-rule*[*sep-heap-rules*]:
  assumes *p*: ‹*is-pure R*› **and** ‹$bb <$ *length a*› **and** ‹$ba <$ *length-ll a bb*› **and** ‹$(bb', bb) \in$ *uint32-nat-rel*›
  shows ‹<*R b bi* $*$ *arrayO-assn* (*arl-assn R*) *a ai*> *update-aa-u ai bb′ ba bi*
    <λr. *R b bi* $*$ ($\exists_A x$. *arrayO-assn* (*arl-assn R*) *x r* $* \uparrow$ ($x =$ *update-ll a bb ba b*))>$_t$›
    solve-direct
  using *assms*
  by (*sep-auto simp add*: *update-aa-u-def update-ll-def p uint32-nat-rel-def br-def*)

lemma *update-aa-hnr*[*sepref-fr-rules*]:
  assumes ‹*is-pure R*›
  shows ‹(*uncurry3 update-aa-u*, *uncurry3* (*RETURN oooo update-ll*)) $\in$
    $[\lambda(((l,i), j), x).\ i <$ *length l* $\wedge\ j <$ *length-ll l i*$]_a$
    (*arrayO-assn* (*arl-assn R*))$^d *_a$ *uint32-nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ $R^k \rightarrow$ (*arrayO-assn* (*arl-assn R*))›
  by *sepref-to-hoare* (*sep-auto simp*: *assms*)

Length

32-bits **definition** (**in** $-$)*length-u-code* **where**
⟨*length-u-code* C = *do* { $n \leftarrow$ *Array.len* C; *return* (*uint32-of-nat* n)}⟩

lemma (**in** $-$)*length-u-hnr*[*sepref-fr-rules*]:
⟨(*length-u-code*, *RETURN o length-uint32-nat*) $\in$ [λC. *length* $C \leq$ *uint32-max*]$_a$ (*array-assn* R)^k $\rightarrow$ *uint32-nat-assn*⟩
supply *length-rule*[*sep-heap-rules*]
by *sepref-to-hoare*
(*sep-auto simp*: *length-u-code-def array-assn-def hr-comp-def is-array-def uint32-nat-rel-def list-rel-imp-same-length br-def nat-of-uint32-uint32-of-nat-id*)

definition *length-arl-u-code* :: ⟨('*a*::*heap*) *array-list* $\Rightarrow$ *uint32 Heap*⟩ **where**
⟨*length-arl-u-code xs* = *do* {
$n \leftarrow$ *arl-length xs*;
return (*uint32-of-nat* n)}⟩

lemma *length-arl-u-hnr*[*sepref-fr-rules*]:
⟨(*length-arl-u-code*, *RETURN o length-uint32-nat*) $\in$
[λxs. *length xs* $\leq$ *uint32-max*]$_a$ (*arl-assn* R)^k $\rightarrow$ *uint32-nat-assn*⟩
by *sepref-to-hoare*
(*sep-auto simp*: *length-u-code-def nat-of-uint32-uint32-of-nat-id*
length-arl-u-code-def arl-assn-def
arl-length-def hr-comp-def is-array-list-def list-rel-pres-length[*symmetric*]
uint32-nat-rel-def br-def)

64-bits **definition** (**in** $-$)*length-u64-code* **where**
⟨*length-u64-code* C = *do* { $n \leftarrow$ *Array.len* C; *return* (*uint64-of-nat* n)}⟩

lemma (**in** $-$)*length-u64-hnr*[*sepref-fr-rules*]:
⟨(*length-u64-code*, *RETURN o length-uint64-nat*)
$\in$ [λC. *length* $C \leq$ *uint64-max*]$_a$ (*array-assn* R)^k $\rightarrow$ *uint64-nat-assn*⟩
supply *length-rule*[*sep-heap-rules*]
by *sepref-to-hoare*
(*sep-auto simp*: *length-u-code-def array-assn-def hr-comp-def is-array-def length-u64-code-def uint64-nat-rel-def list-rel-imp-same-length br-def nat-of-uint64-uint64-of-nat-id*)

Length for arrays in arrays

32-bits **definition** (**in** $-$)*length-aa-u* :: ⟨('*a*::*heap array-list*) *array* $\Rightarrow$ *uint32* $\Rightarrow$ *nat Heap*⟩ **where**
⟨*length-aa-u xs i* = *length-aa xs* (*nat-of-uint32 i*)⟩

lemma *length-aa-u-code*[*code*]:
⟨*length-aa-u xs i* = *nth-u-code xs i* $\gg=$ *arl-length*⟩
unfolding *length-aa-u-def length-aa-def nth-u-def*[*symmetric*] *nth-u-code-def Array.nth'-def*
by (*auto simp*: *nat-of-uint32-code*)

lemma *length-aa-u-hnr*[*sepref-fr-rules*]: ⟨(*uncurry length-aa-u*, *uncurry* (*RETURN* $\circ\circ$ *length-ll*)) $\in$
[$\lambda(xs, i)$. $i <$ *length xs*]$_a$ (*arrayO-assn* (*arl-assn* R))^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow$ *nat-assn*⟩
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def length-aa-u-def br-def*)

definition *length-raa-u* :: ⟨'*a*::*heap arrayO-raa* $\Rightarrow$ *nat* $\Rightarrow$ *uint32 Heap*⟩ **where**

⟨*length-raa-u xs i* = *do* {
  x ← *arl-get xs i*;
  length-u-code x}⟩

lemma *length-raa-u-alt-def*: ⟨*length-raa-u xs i* = *do* {
  n ← *length-raa xs i*;
  return (*uint32-of-nat n*)}⟩
  unfolding *length-raa-u-def length-raa-def length-u-code-def*
  by *auto*

definition *length-rll-n-uint32* **where**
  [*simp*]: ⟨*length-rll-n-uint32* = *length-rll*⟩

lemma *length-raa-rule*[*sep-heap-rules*]:
  ⟨*b* < *length xs* ⟹ <*arlO-assn* (*array-assn R*) *xs a*> *length-raa-u a b*
  <λ*r*. *arlO-assn* (*array-assn R*) *xs a* ∗ ↑ (*r* = *uint32-of-nat* (*length-rll xs b*))>$_t$⟩
  unfolding *length-raa-u-alt-def length-u-code-def*
  by *sep-auto*

lemma *length-raa-u-hnr*[*sepref-fr-rules*]:
  shows ⟨(*uncurry length-raa-u*, *uncurry* (*RETURN* ∘∘ *length-rll-n-uint32*)) ∈
    [λ(*xs*, *i*). *i* < *length xs* ∧ *length* (*xs* ! *i*) ≤ *uint32-max*]$_a$
      (*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} → *uint32-nat-assn*⟩
  by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def length-rll-def*
    nat-of-uint32-uint32-of-nat-id)+

TODO: proper fix to avoid the conversion to uint32

definition *length-aa-u-code* :: ⟨(′*a*::*heap array*) *array-list* ⇒ *nat* ⇒ *uint32 Heap*⟩ **where**
  ⟨*length-aa-u-code xs i* = *do* {
  n ← *length-raa xs i*;
  return (*uint32-of-nat n*)}⟩

64-bits **definition** (**in** −)*length-aa-u64* :: ⟨(′*a*::*heap array-list*) *array* ⇒ *uint64* ⇒ *nat Heap*⟩ **where**
  ⟨*length-aa-u64 xs i* = *length-aa xs* (*nat-of-uint64 i*)⟩

lemma *length-aa-u64-code*[*code*]:
  ⟨*length-aa-u64 xs i* = *nth-u64-code xs i* ⤜ *arl-length*⟩
  unfolding *length-aa-u64-def length-aa-def nth-u64-def*[*symmetric*] *nth-u64-code-def*
  Array.nth′-def
  by (*auto simp*: *nat-of-uint64-code*)

lemma *length-aa-u64-hnr*[*sepref-fr-rules*]: ⟨(*uncurry length-aa-u64*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈
    [λ(*xs*, *i*). *i* < *length xs*]$_a$ (*arrayO-assn* (*arl-assn R*))^k $*_a$ *uint64-nat-assn*^{k} → *nat-assn*⟩
  by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def length-aa-u64-def br-def*)

definition *length-raa-u64* :: ⟨′*a*::*heap arrayO-raa* ⇒ *nat* ⇒ *uint64 Heap*⟩ **where**
  ⟨*length-raa-u64 xs i* = *do* {
  x ← *arl-get xs i*;
  length-u64-code x}⟩

lemma *length-raa-u64-alt-def*: ⟨*length-raa-u64 xs i* = *do* {
  n ← *length-raa xs i*;
  return (*uint64-of-nat n*)}⟩
  unfolding *length-raa-u64-def length-raa-def length-u64-code-def*
  by *auto*

definition *length-rll-n-uint64* **where**
[*simp*]: ‹*length-rll-n-uint64* = *length-rll*›

lemma *length-raa-u64-hnr*[*sepref-fr-rules*]:
shows ‹(*uncurry length-raa-u64*, *uncurry* (*RETURN* ∘∘ *length-rll-n-uint64*)) ∈
$[\lambda(xs, i).\ i < length\ xs \wedge length\ (xs\ !\ i) \leq uint64\text{-}max]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def length-rll-def*
nat-of-uint64-uint64-of-nat-id length-raa-u64-alt-def)+

Delete at index

definition *delete-index-and-swap-aa* **where**
‹*delete-index-and-swap-aa xs i j* = *do* {
$x \leftarrow$ *last-aa xs i*;
$xs \leftarrow$ *update-aa xs i j x*;
set-butlast-aa xs i
}›

lemma *delete-index-and-swap-aa-ll-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry2 delete-index-and-swap-aa*, *uncurry2* (*RETURN ooo delete-index-and-swap-ll*))
$\in [\lambda((l,i), j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k$
$\rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
using *assms* **unfolding** *delete-index-and-swap-aa-def*
by *sepref-to-hoare* (*sep-auto dest*: *le-length-ll-nemptyD*
simp: *delete-index-and-swap-ll-def update-ll-def last-ll-def set-butlast-ll-def*
length-ll-def[*symmetric*])

Last (arrays of arrays)

definition *last-aa-u* **where**
‹*last-aa-u xs i* = *last-aa xs* (*nat-of-uint32 i*)›

lemma *last-aa-u-code*[*code*]:
‹*last-aa-u xs i* = *nth-u-code xs i* ≫= *arl-last*›
unfolding *last-aa-u-def last-aa-def nth-nat-of-uint32-nth′ nth-nat-of-uint32-nth′*
arl-get-u-def[*symmetric*] *nth-u-code-def*[*symmetric*] **..**

lemma *length-delete-index-and-swap-ll*[*simp*]:
‹*length* (*delete-index-and-swap-ll s i j*) = *length s*›
by (*auto simp*: *delete-index-and-swap-ll-def*)

definition *set-butlast-aa-u* **where**
‹*set-butlast-aa-u xs i* = *set-butlast-aa xs* (*nat-of-uint32 i*)›

lemma *set-butlast-aa-u-code*[*code*]:
‹*set-butlast-aa-u a i* = *do* {
$x \leftarrow$ *nth-u-code a i*;
$a' \leftarrow$ *arl-butlast x*;
Array-upd-u i a′ a
}› — Replace the *i*-th element by the itself execpt the last element.
unfolding *set-butlast-aa-u-def set-butlast-aa-def*
nth-u-code-def Array-upd-u-def

by (*auto simp*: *Array.nth'-def nat-of-uint32-code*)

definition *delete-index-and-swap-aa-u* **where**
⟨*delete-index-and-swap-aa-u xs i* = *delete-index-and-swap-aa xs* (*nat-of-uint32 i*)⟩

lemma *delete-index-and-swap-aa-u-code*[*code*]:
⟨*delete-index-and-swap-aa-u xs i j* = *do* {
x ← *last-aa-u xs i*;
xs ← *update-aa-u xs i j x*;
set-butlast-aa-u xs i
}⟩
unfolding *delete-index-and-swap-aa-u-def delete-index-and-swap-aa-def*
last-aa-u-def update-aa-u-def set-butlast-aa-u-def
by *auto*

lemma *delete-index-and-swap-aa-ll-hnr-u*[*sepref-fr-rules*]:
assumes ⟨*is-pure R*⟩
shows ⟨(*uncurry2 delete-index-and-swap-aa-u*, *uncurry2* (*RETURN ooo delete-index-and-swap-ll*))
∈ $[\lambda((l,i),\ j).\ i < \mathit{length}\ l \wedge j < \mathit{length\text{-}ll}\ l\ i]_a$ (*arrayO-assn* (*arl-assn R*))^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *nat-assn*^{k}
→ (*arrayO-assn* (*arl-assn R*))⟩
using *assms* **unfolding** *delete-index-and-swap-aa-def delete-index-and-swap-aa-u-def*
by *sepref-to-hoare* (*sep-auto dest*: *le-length-ll-nemptyD*
simp: *delete-index-and-swap-ll-def update-ll-def last-ll-def set-butlast-ll-def*
length-ll-def[*symmetric*] *uint32-nat-rel-def br-def*)

Swap

definition *swap-u-code* :: *'a* ::*heap array* ⇒ *uint32* ⇒ *uint32* ⇒ *'a array Heap* **where**
⟨*swap-u-code xs i j* = *do* {
ki ← *nth-u-code xs i*;
kj ← *nth-u-code xs j*;
xs ← *heap-array-set-u xs i kj*;
xs ← *heap-array-set-u xs j ki*;
return xs
}⟩

lemma *op-list-swap-u-hnr*[*sepref-fr-rules*]:
assumes *p*: ⟨*CONSTRAINT is-pure R*⟩
shows ⟨(*uncurry2 swap-u-code*, *uncurry2* (*RETURN ooo op-list-swap*)) ∈
$[\lambda((xs,\ i),\ j).\ \ i < \mathit{length}\ xs \wedge j < \mathit{length}\ xs]_a$
(*array-assn R*)^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} → *array-assn R*⟩
proof −
obtain *R'* **where** *R*: ⟨*the-pure R* = *R'*⟩ **and** *R'*: ⟨*R* = *pure R'*⟩
using *p* **by** *fastforce*
show *?thesis*
by (*sepref-to-hoare*)
(*sep-auto simp*: *swap-u-code-def swap-def nth-u-code-def is-array-def*
array-assn-def hr-comp-def nth-nat-of-uint32-nth'[*symmetric*]
list-rel-imp-same-length uint32-nat-rel-def br-def
heap-array-set-u-def heap-array-set'-u-def Array.upd'-def
nat-of-uint32-code[*symmetric*] *R IICF-List.swap-def*[*symmetric*] *IICF-List.swap-param*
intro!: *list-rel-update*[*of - - R true - -* ⟨(-, {})⟩, *unfolded R*] *param-nth*)
qed

definition *swap-u64-code* :: *'a ::heap array* ⇒ *uint64* ⇒ *uint64* ⇒ *'a array Heap* **where**
⟨*swap-u64-code xs i j* = *do* {
  ki ← *nth-u64-code xs i*;
  kj ← *nth-u64-code xs j*;
  xs ← *heap-array-set-u64 xs i kj*;
  xs ← *heap-array-set-u64 xs j ki*;
  return xs
}⟩

lemma *op-list-swap-u64-hnr*[*sepref-fr-rules*]:
assumes *p*: ⟨*CONSTRAINT is-pure R*⟩
shows ⟨(*uncurry2 swap-u64-code*, *uncurry2* (*RETURN ooo op-list-swap*)) ∈
$[\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ xs]_a$
$(array\text{-}assn\ R)^d *_a uint64\text{-}nat\text{-}assn^k\ *_a uint64\text{-}nat\text{-}assn^k \rightarrow array\text{-}assn\ R$⟩
proof −
obtain *R'* **where** *R*: ⟨*the-pure R* = *R'*⟩ **and** *R'*: ⟨*R* = *pure R'*⟩
using *p* **by** *fastforce*
show *?thesis*
by (*sepref-to-hoare*)
(*sep-auto simp*: *swap-u64-code-def swap-def nth-u64-code-def is-array-def*
array-assn-def hr-comp-def nth-nat-of-uint64-nth'[*symmetric*]
list-rel-imp-same-length uint64-nat-rel-def br-def
heap-array-set-u64-def heap-array-set'-u64-def Array.upd'-def
nat-of-uint64-code[*symmetric*] *R IICF-List.swap-def*[*symmetric*] *IICF-List.swap-param*
intro!: *list-rel-update*[*of - - R true - -* ⟨(-, {})⟩, *unfolded R*] *param-nth*)
qed

definition *swap-aa-u64* :: (*'a*::{*heap*,*default*}) *arrayO-raa* ⇒ *nat* ⇒ *uint64* ⇒ *uint64* ⇒ *'a arrayO-raa Heap* **where**
⟨*swap-aa-u64 xs k i j* = *do* {
  xi ← *arl-get xs k*;
  xj ← *swap-u64-code xi i j*;
  xs ← *arl-set xs k xj*;
  return xs
}⟩

lemma *swap-aa-u64-hnr*[*sepref-fr-rules*]:
assumes ⟨*is-pure R*⟩
shows ⟨(*uncurry3 swap-aa-u64*, *uncurry3* (*RETURN oooo swap-ll*)) ∈
$[\lambda(((xs, k), i), j).\ k < length\ xs \wedge i < length\text{-}rll\ xs\ k \wedge j < length\text{-}rll\ xs\ k]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow$
(*arlO-assn* (*array-assn R*))⟩
proof −
note *update-raa-rule-pure*[*sep-heap-rules*]
obtain *R'* **where** *R'*: ⟨*R'* = *the-pure R*⟩ **and** *RR'*: ⟨*R* = *pure R'*⟩
using *assms* **by** *fastforce*
have [*simp*]: ⟨*the-pure* (λ*a b*. ↑ ((*b*, *a*) ∈ *R'*)) = *R'*⟩
unfolding *pure-def*[*symmetric*] **by** *auto*
have *H*: ⟨<*is-array-list p* (*aa*, *bc*) ∗
heap-list-all-nth (*array-assn* (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*))) (*remove1 bb* [*0*..<*length p*]) *a p* ∗
array-assn (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*)) (*a* ! *bb*) (*p* ! *bb*)>
Array.nth (*p* ! *bb*) (*nat-of-integer* (*integer-of-uint64 bia*))
<λ*r*. $\exists_A$ *p'*. *is-array-list p'* (*aa*, *bc*) ∗ ↑ (*bb* < *length p'* ∧ *p'* ! *bb* = *p* ! *bb* ∧ *length a* = *length p'*) ∗

heap-list-all-nth (*array-assn* (λa *c*. $\uparrow$ ((*c*, *a*) $\in$ R'))) (*remove1 bb* [*0*..<*length* p']) *a* p' $*$
array-assn (λa *c*. $\uparrow$ ((*c*, *a*) $\in$ R')) (*a* ! *bb*) (p' ! *bb*) $*$
R (*a* ! *bb* ! (*nat-of-uint64 bia*)) *r* >›
if
‹*is-pure* (λa *c*. $\uparrow$ ((*c*, *a*) $\in$ R'))› **and**
‹*bb* < *length p*› **and**
‹*nat-of-uint64 bia* < *length* (*a* ! *bb*)› **and**
‹*nat-of-uint64 bi* < *length* (*a* ! *bb*)› **and**
‹*length a* = *length p*›
for *bi* :: ‹*uint64*› **and** *bia* :: ‹*uint64*› **and** *bb* :: ‹*nat*› **and** *a* :: ‹*'a list list*› **and**
aa :: ‹*'b array array*› **and** *bc* :: ‹*nat*› **and** *p* :: ‹*'b array list*›
using *that*
by (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def nat-of-uint64-code*[*symmetric*]
list-rel-imp-same-length RR' pure-def param-nth)
have *H'*: ‹*is-array-list* p' (*aa*, *ba*) $*$ p' ! *bb* $\mapsto_a$ *b* [*nat-of-uint64 bia* := *b* ! *nat-of-uint64 bi*,
nat-of-uint64 bi := *xa*] $*$
heap-list-all-nth (λa *b*. $\exists_A$*ba*. *b* $\mapsto_a$ *ba* $*$ $\uparrow$ ((*ba*, *a*) $\in$ $\langle R' \rangle$*list-rel*))
(*remove1 bb* [*0*..<*length* p']) *a* p' $*$ *R* (*a* ! *bb* ! *nat-of-uint64 bia*) *xa* $\Longrightarrow_A$
is-array-list p' (*aa*, *ba*) $*$
heap-list-all
(λa *c*. $\exists_A$*b*. *c* $\mapsto_a$ *b* $*$ $\uparrow$ ((*b*, *a*) $\in$ $\langle R' \rangle$*list-rel*))
(*a*[*bb* := (*a* ! *bb*) [*nat-of-uint64 bia* := *a* ! *bb* ! *nat-of-uint64 bi*,
nat-of-uint64 bi := *a* ! *bb* ! *nat-of-uint64 bia*]])
p' $*$ *true*›
if
‹*is-pure* (λa *c*. $\uparrow$ ((*c*, *a*) $\in$ R'))› **and**
le: ‹*nat-of-uint64 bia* < *length* (*a* ! *bb*)› **and**
le': ‹*nat-of-uint64 bi* < *length* (*a* ! *bb*)› **and**
‹*bb* < *length* p'› **and**
‹*length a* = *length* p'› **and**
a: ‹(*b*, *a* ! *bb*) $\in$ $\langle R' \rangle$*list-rel*›
for *bi* :: ‹*uint64*› **and** *bia* :: ‹*uint64*› **and** *bb* :: ‹*nat*› **and** *a* :: ‹*'a list list*› **and**
xa :: ‹*'b*› **and** p' :: ‹*'b array list*› **and** *b* :: ‹*'b list*› **and** *aa* :: ‹*'b array array*› **and** *ba* :: ‹*nat*›
proof −
have *1*: ‹(*b*[*nat-of-uint64 bia* := *b* ! *nat-of-uint64 bi*, *nat-of-uint64 bi* := *xa*],
(*a* ! *bb*)[*nat-of-uint64 bia* := *a* ! *bb* ! *nat-of-uint64 bi*,
nat-of-uint64 bi := *a* ! *bb* ! *nat-of-uint64 bia*]) $\in$ $\langle R' \rangle$*list-rel*›
if ‹(*xa*, *a* ! *bb* ! *nat-of-uint64 bia*) $\in$ R'›
using *that a le le'*
unfolding *list-rel-def list-all2-conv-all-nth*
by *auto*
have *2*: ‹*heap-list-all-nth* (λa *b*. $\exists_A$*ba*. *b* $\mapsto_a$ *ba* $*$ $\uparrow$ ((*ba*, *a*) $\in$ $\langle R' \rangle$*list-rel*)) (*remove1 bb* [*0*..<*length* p']) *a* p' =
heap-list-all-nth (λa *c*. $\exists_A$*b*. *c* $\mapsto_a$ *b* $*$ $\uparrow$ ((*b*, *a*) $\in$ $\langle R' \rangle$*list-rel*)) (*remove1 bb* [*0*..<*length* p'])
(*a*[*bb* := (*a* ! *bb*)[*nat-of-uint64 bia* := *a* ! *bb* ! *nat-of-uint64 bi*, *nat-of-uint64 bi* := *a* ! *bb* ! *nat-of-uint64 bia*]]) p'›
by (*rule heap-list-all-nth-cong*) *auto*
show *?thesis* **using** *that*
unfolding *heap-list-all-heap-list-all-nth-eq*
by (*subst* (*2*) *heap-list-all-nth-remove1*[*of bb*])
(*sep-auto simp*: *heap-list-all-heap-list-all-nth-eq swap-def fr-refl RR'*
pure-def 2[*symmetric*] *intro*!: *1*)+
qed

show *?thesis*
using *assms* **unfolding** R'[*symmetric*] **unfolding** *RR'*

apply *sepref-to-hoare*
apply (*sep-auto simp*: *swap-aa-u64-def swap-ll-def arlO-assn-except-def length-rll-def length-rll-update-rll nth-raa-i-u64-def uint64-nat-rel-def br-def swap-def nth-rll-def list-update-swap swap-u64-code-def nth-u64-code-def Array.nth′-def heap-array-set-u64-def heap-array-set′-u64-def arl-assn-def IICF-List.swap-def Array.upd′-def*)
apply (*rule H*; *assumption*)
apply (*sep-auto simp*: *array-assn-def nat-of-uint64-code*[*symmetric*] *hr-comp-def is-array-def list-rel-imp-same-length arlO-assn-def arl-assn-def hr-comp-def*[*abs-def*])
apply (*rule H′*; *assumption*)
done
qed

definition *arl-swap-u-code*
:: *′a ::heap array-list ⇒ uint32 ⇒ uint32 ⇒ ′a array-list Heap*
where
‹*arl-swap-u-code xs i j = do* {
ki ← arl-get-u xs i;
kj ← arl-get-u xs j;
xs ← arl-set-u xs i kj;
xs ← arl-set-u xs j ki;
return xs
}›

lemma *arl-op-list-swap-u-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*CONSTRAINT is-pure R*›
shows ‹(*uncurry2 arl-swap-u-code*, *uncurry2* (*RETURN ooo op-list-swap*)) ∈ $[\lambda((xs,\ i),\ j).\ \ i < length\ xs \wedge j < length\ xs]_a$ $(arl\text{-}assn\ R)^d *_a uint32\text{-}nat\text{-}assn^k\ *_a uint32\text{-}nat\text{-}assn^k \rightarrow arl\text{-}assn\ R$›
proof −
obtain *R′* **where** *R*: ‹*the-pure R = R′*› **and** *R′*: ‹*R = pure R′*›
using *p* **by** *fastforce*
show *?thesis*
by (*sepref-to-hoare*)
(*sep-auto simp*: *arl-swap-u-code-def swap-def nth-u-code-def is-array-def array-assn-def hr-comp-def nth-nat-of-uint32-nth′*[*symmetric*] *list-rel-imp-same-length uint32-nat-rel-def br-def arl-assn-def heap-array-set-u-def heap-array-set′-u-def Array.upd′-def arl-set′-u-def R R′ IICF-List.swap-def*[*symmetric*] *IICF-List.swap-param nat-of-uint32-code*[*symmetric*] *R arl-set-u-def arl-get′-def arl-get-u-def intro*!: *list-rel-update*[*of - - R true - -* ‹(-, {})›, *unfolded R*] *param-nth*)
qed

Take

definition *shorten-take-aa-u32* **where**
‹*shorten-take-aa-u32 L j W = do* {
(*a*, *n*) ← *nth-u-code W L*;
heap-array-set-u W L (*a*, *j*)
}›

lemma *shorten-take-aa-u32-alt-def*:
‹*shorten-take-aa-u32 L j W = shorten-take-aa* (*nat-of-uint32 L*) *j W*›
by (*auto simp*: *shorten-take-aa-u32-def shorten-take-aa-def uint32-nat-rel-def br-def Array.nth′-def heap-array-set-u-def heap-array-set′-u-def Array.upd′-def*

nth-u-code-def nat-of-uint32-code[*symmetric*] *upd-return*)

lemma *shorten-take-aa-u32-hnr*[*sepref-fr-rules*]:
‹(*uncurry2 shorten-take-aa-u32*, *uncurry2* (*RETURN ooo shorten-take-ll*)) ∈
[$\lambda((L, j), W).\ j \leq length\ (W\ !\ L) \wedge L < length\ W]_a$
$uint32\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d \rightarrow arrayO\text{-}assn\ (arl\text{-}assn\ R)$›
unfolding *shorten-take-aa-u32-alt-def shorten-take-ll-def nth-u-code-def uint32-nat-rel-def br-def*
Array.nth′-def heap-array-set-u-def heap-array-set′-u-def Array.upd′-def shorten-take-aa-def
by *sepref-to-hoare* (*sep-auto simp*: *nat-of-uint32-code*[*symmetric*])

List of Lists

Getters **definition** *nth-raa-i32* :: ‹′*a*::*heap arrayO-raa* ⇒ *uint32* ⇒ *nat* ⇒ ′*a Heap*› **where**
‹*nth-raa-i32 xs i j* = *do* {
  x ← *arl-get-u xs i*;
  y ← *Array.nth x j*;
  return y}›

lemma *nth-raa-i32-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-raa-i32*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
[$\lambda((xs, i), j).\ i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›
proof −
have *1*: ‹$a * b * array\text{-}assn\ R\ x\ y = array\text{-}assn\ R\ x\ y * a * b$› **for** *a b c* :: *assn* **and** *x y*
by (*auto simp*: *ac-simps*)
have *2*: ‹$a * arl\text{-}assn\ R\ x\ y * c = arl\text{-}assn\ R\ x\ y * a * c$› **for** *a c* :: *assn* **and** *x y* **and** *R*
by (*auto simp*: *ac-simps*)
have [*simp*]: ‹*R a b* = ↑((*b*,*a*) ∈ *the-pure R*)› **for** *a b*
using *assms* **by** (*metis CONSTRAINT-D pure-app-eq pure-the-pure*)
show *?thesis*
using *assms*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-raa-i32-def arl-get-u-def*
uint32-nat-rel-def br-def nat-of-uint32-code[*symmetric*]
arlO-assn-except-def 1 arl-get′-def
)
apply (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def list-rel-imp-same-length*
param-nth nth-rll-def)
apply (*sep-auto simp*: *arlO-assn-def 2*)
apply (*subst mult.assoc*)+
apply (*rule fr-refl′*)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst-tac* (*2*) *i*=‹*nat-of-uint32 bia*› **in** *heap-list-all-nth-remove1*)
apply (*sep-auto simp*: *nth-rll-def is-array-def hr-comp-def*)+
done
qed

definition *nth-raa-i32-u64* :: ‹′*a*::*heap arrayO-raa* ⇒ *uint32* ⇒ *uint64* ⇒ ′*a Heap*› **where**
‹*nth-raa-i32-u64 xs i j* = *do* {
  x ← *arl-get-u xs i*;
  y ← *nth-u64-code x j*;
  return y}›

lemma *nth-raa-i32-u64-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-raa-i32-u64*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
[λ((*xs*, *i*), *j*). *i* < *length xs* ∧ *j* < *length* (*xs* !*i*)]$_a$
(*arlO-assn* (*array-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
proof −
have *1*: ‹*a* ∗ *b* ∗ *array-assn R x y* = *array-assn R x y* ∗ *a* ∗ *b*› **for** *a b c* :: *assn* **and** *x y*
by (*auto simp*: *ac-simps*)
have *2*: ‹*a* ∗ *arl-assn R x y* ∗ *c* = *arl-assn R x y* ∗ *a* ∗ *c*› **for** *a c* :: *assn* **and** *x y* **and** *R*
by (*auto simp*: *ac-simps*)
have [*simp*]: ‹*R a b* = ↑((*b*,*a*) ∈ *the-pure R*)› **for** *a b*
using *assms* **by** (*metis CONSTRAINT-D pure-app-eq pure-the-pure*)
show *?thesis*
using *assms*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-raa-i32-u64-def arl-get-u-def*
uint32-nat-rel-def br-def nat-of-uint32-code[*symmetric*]
arlO-assn-except-def 1 arl-get′-def Array.nth′-def nth-u64-code-def
nat-of-uint64-code[*symmetric*] *uint64-nat-rel-def*)
apply (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def list-rel-imp-same-length*
param-nth nth-rll-def)
apply (*sep-auto simp*: *arlO-assn-def 2*)
apply (*subst mult.assoc*)+
apply (*rule fr-refl′*)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst-tac* (*2*) *i*=‹*nat-of-uint32 bia*› **in** *heap-list-all-nth-remove1*)
apply (*sep-auto simp*: *nth-rll-def is-array-def hr-comp-def*)+
done
qed

definition *nth-raa-i32-u32* :: ‹′*a*::*heap arrayO-raa* ⇒ *uint32* ⇒ *uint32* ⇒ ′*a Heap*› **where**
‹*nth-raa-i32-u32 xs i j* = *do* {
x ← *arl-get-u xs i*;
y ← *nth-u-code x j*;
return y}›

lemma *nth-raa-i32-u32-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-raa-i32-u32*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
[λ((*xs*, *i*), *j*). *i* < *length xs* ∧ *j* < *length* (*xs* !*i*)]$_a$
(*arlO-assn* (*array-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} → *R*›
proof −
have *1*: ‹*a* ∗ *b* ∗ *array-assn R x y* = *array-assn R x y* ∗ *a* ∗ *b*› **for** *a b c* :: *assn* **and** *x y*
by (*auto simp*: *ac-simps*)
have *2*: ‹*a* ∗ *arl-assn R x y* ∗ *c* = *arl-assn R x y* ∗ *a* ∗ *c*› **for** *a c* :: *assn* **and** *x y* **and** *R*
by (*auto simp*: *ac-simps*)
have [*simp*]: ‹*R a b* = ↑((*b*,*a*) ∈ *the-pure R*)› **for** *a b*
using *assms* **by** (*metis CONSTRAINT-D pure-app-eq pure-the-pure*)
show *?thesis*
using *assms*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-raa-i32-u32-def arl-get-u-def*
uint32-nat-rel-def br-def nat-of-uint32-code[*symmetric*]
arlO-assn-except-def 1 arl-get′-def Array.nth′-def nth-u-code-def

nat-of-uint32-code[*symmetric*] *uint32-nat-rel-def*)
apply (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def list-rel-imp-same-length param-nth nth-rll-def*)
apply (*sep-auto simp*: *arlO-assn-def 2*)
apply (*subst mult.assoc*)+
apply (*rule fr-refl′*)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst-tac* (*2*) *i*=‹*nat-of-uint32 bia*› **in** *heap-list-all-nth-remove1*)
apply (*sep-auto simp*: *nth-rll-def is-array-def hr-comp-def*)+
done
qed

definition *nth-aa-i32-u32* **where**
‹*nth-aa-i32-u32 x L L′* = *nth-aa x* (*nat-of-uint32 L*) (*nat-of-uint32 L′*)›

definition *nth-aa-i32-u32′* **where**
‹*nth-aa-i32-u32′ xs i j* = *do* {
x ← *nth-u-code xs i*;
y ← *arl-get-u x j*;
return y}›

lemma *nth-aa-i32-u32*[*code*]:
‹*nth-aa-i32-u32 x L L′* = *nth-aa-i32-u32′ x L L′*›
unfolding *nth-aa-u-def nth-aa′-def nth-aa-def Array.nth′-def nat-of-uint32-code nth-aa-i32-u32-def nth-aa-i32-u32′-def nth-u-code-def arl-get-u-def arl-get′-def*
by (*auto simp*: *nat-of-uint32-code*[*symmetric*])

lemma *nth-aa-i32-u32-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-aa-i32-u32*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
$[\lambda((x, L), L').\ L < \mathit{length}\ x \wedge L' < \mathit{length}\ (x\ !\ L)]_a$
$(\mathit{arrayO\text{-}assn}\ (\mathit{arl\text{-}assn}\ R))^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k \rightarrow R$›
unfolding *nth-aa-i32-u32-def*
by *sepref-to-hoare*
(*use assms* **in** ‹*sep-auto simp*: *uint32-nat-rel-def br-def length-ll-def nth-ll-def nth-rll-def*›)

definition *nth-raa-i64-u32* :: ‹′*a*::*heap arrayO-raa* ⇒ *uint64* ⇒ *uint32* ⇒ ′*a Heap*› **where**
‹*nth-raa-i64-u32 xs i j* = *do* {
x ← *arl-get-u64 xs i*;
y ← *nth-u-code x j*;
return y}›

lemma *nth-raa-i64-u32-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-raa-i64-u32*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
$[\lambda((xs, i), j).\ i < \mathit{length}\ xs \wedge j < \mathit{length}\ (xs\ !i)]_a$
$(\mathit{arlO\text{-}assn}\ (\mathit{array\text{-}assn}\ R))^k *_a \mathit{uint64\text{-}nat\text{-}assn}^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k \rightarrow R$›
proof −
have *1*: ‹*a* ∗ *b* ∗ *array-assn R x y* = *array-assn R x y* ∗ *a* ∗ *b*› **for** *a b c* :: *assn* **and** *x y*
by (*auto simp*: *ac-simps*)
have *2*: ‹*a* ∗ *arl-assn R x y* ∗ *c* = *arl-assn R x y* ∗ *a* ∗ *c*› **for** *a c* :: *assn* **and** *x y* **and** *R*

by (*auto simp*: *ac-simps*)
have [*simp*]: ‹$R\ a\ b = \uparrow((b,a) \in$ *the-pure R*)› **for** *a b*
using *assms* **by** (*metis CONSTRAINT-D pure-app-eq pure-the-pure*)
show *?thesis*
using *assms*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-raa-i64-u32-def arl-get-u64-def uint32-nat-rel-def br-def nat-of-uint32-code*[*symmetric*] *arlO-assn-except-def 1 arl-get'-def Array.nth'-def nth-u64-code-def nat-of-uint64-code*[*symmetric*] *uint64-nat-rel-def nth-u-code-def*)
apply (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def list-rel-imp-same-length param-nth nth-rll-def*)
apply (*sep-auto simp*: *arlO-assn-def 2*)
apply (*subst mult.assoc*)+
apply (*rule fr-refl'*)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst-tac* (*2*) *i*=‹*nat-of-uint64 bia*› **in** *heap-list-all-nth-remove1*)
apply (*sep-auto simp*: *nth-rll-def is-array-def hr-comp-def*)+
done
qed

thm *nth-aa-uint-hnr*
find-theorems *nth-aa-u*

lemma *nth-aa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 nth-aa*, *uncurry2* (*RETURN* ∘∘∘ *nth-ll*)) ∈
$[\lambda((l,i),j).\ i <$ *length l* $\wedge\ j <$ *length-ll l i*$]_a$
(*arrayO-assn* (*arl-assn R*))$^k *_a$ *nat-assn*$^k *_a$ *nat-assn*$^k \rightarrow R$›
proof −
obtain R' **where** *R*: ‹*the-pure* $R = R'$› **and** R': ‹$R =$ *pure* R'›
using *p* **by** *fastforce*
have *H*: ‹*list-all2* $(\lambda x\ x'.\ (x, x') \in$ *the-pure* $(\lambda a\ c. \uparrow ((c, a) \in R')))$ *bc* (*a* ! *ba*) ⟹
$b <$ *length* (*a* ! *ba*) ⟹
(*bc* ! *b*, *a* ! *ba* ! *b*) $\in R'$› **for** *bc a ba b*
by (*auto simp add*: *ent-refl-true list-all2-conv-all-nth is-pure-alt-def pure-app-eq*[*symmetric*])
show *?thesis*
apply *sepref-to-hoare*
apply (*subst* (*2*) *arrayO-except-assn-array0-index*[*symmetric*])
apply (*solves* ‹*auto*›)[]
apply (*sep-auto simp*: *nth-aa-def nth-ll-def length-ll-def*)
apply (*sep-auto simp*: *arrayO-except-assn-def arrayO-assn-def arl-assn-def hr-comp-def list-rel-def list-all2-lengthD*
star-aci(*3*) *R R' pure-def H*)
done
qed

definition *nth-raa-i64-u64* :: ‹'*a*::*heap arrayO-raa* ⇒ *uint64* ⇒ *uint64* ⇒ '*a Heap*› **where**
‹*nth-raa-i64-u64 xs i j* = *do* {
x ← *arl-get-u64 xs i*;
y ← *nth-u64-code x j*;
return y}›

lemma *nth-raa-i64-u64-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›

shows
‹(*uncurry2 nth-raa-i64-u64*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
[λ((*xs*, *i*), *j*). *i* < *length xs* ∧ *j* < *length* (*xs* !*i*)]$_a$
(*arlO-assn* (*array-assn R*))^k $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
proof −
have *1*: ‹*a* ∗ *b* ∗ *array-assn R x y* = *array-assn R x y* ∗ *a* ∗ *b*› **for** *a b c* :: *assn* **and** *x y*
by (*auto simp*: *ac-simps*)
have *2*: ‹*a* ∗ *arl-assn R x y* ∗ *c* = *arl-assn R x y* ∗ *a* ∗ *c*› **for** *a c* :: *assn* **and** *x y* **and** *R*
by (*auto simp*: *ac-simps*)
have [*simp*]: ‹*R a b* = ↑((*b*,*a*) ∈ *the-pure R*)› **for** *a b*
using *assms* **by** (*metis CONSTRAINT-D pure-app-eq pure-the-pure*)
show *?thesis*
using *assms*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-raa-i64-u64-def arl-get-u64-def*
uint32-nat-rel-def br-def nat-of-uint32-code[*symmetric*]
arlO-assn-except-def 1 arl-get′-def Array.nth′-def nth-u64-code-def
nat-of-uint64-code[*symmetric*] *uint64-nat-rel-def nth-u64-code-def*)
apply (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def list-rel-imp-same-length*
param-nth nth-rll-def)
apply (*sep-auto simp*: *arlO-assn-def 2*)
apply (*subst mult.assoc*)+
apply (*rule fr-refl′*)
apply (*subst heap-list-all-heap-list-all-nth-eq*)
apply (*subst-tac* (*2*) *i*=‹*nat-of-uint64 bia*› **in** *heap-list-all-nth-remove1*)
apply (*sep-auto simp*: *nth-rll-def is-array-def hr-comp-def*)+
done
qed

lemma *nth-aa-i64-u64-code*[*code*]:
‹*nth-aa-i64-u64 x L L′* = *nth-u64-code x L* ⤜ (λ*x*. *arl-get-u64 x L′* ⤜ *return*)›
unfolding *nth-aa-u-def nth-aa-def arl-get-u-def*[*symmetric*] *Array.nth′-def*[*symmetric*]
nth-nat-of-uint32-nth′ nth-u-code-def[*symmetric*] *nth-nat-of-uint64-nth′*
nth-aa-i64-u64-def nth-u64-code-def arl-get-u64-def arl-get′-def
nat-of-uint64-code[*symmetric*]
..

lemma *nth-aa-i64-u32-code*[*code*]:
‹*nth-aa-i64-u32 x L L′* = *nth-u64-code x L* ⤜ (λ*x*. *arl-get-u x L′* ⤜ *return*)›
unfolding *nth-aa-u-def nth-aa-def arl-get-u-def*[*symmetric*] *Array.nth′-def*[*symmetric*]
nth-nat-of-uint32-nth′ nth-u-code-def[*symmetric*] *nth-nat-of-uint64-nth′*
nth-aa-i64-u32-def nth-u64-code-def arl-get-u64-def arl-get′-def
nat-of-uint64-code[*symmetric*] *arl-get-u-def nat-of-uint32-code*[*symmetric*]
..

lemma *nth-aa-i32-u64-code*[*code*]:
‹*nth-aa-i32-u64 x L L′* = *nth-u-code x L* ⤜ (λ*x*. *arl-get-u64 x L′* ⤜ *return*)›
unfolding *nth-aa-u-def nth-aa-def arl-get-u-def*[*symmetric*] *Array.nth′-def*[*symmetric*]
nth-nat-of-uint32-nth′ nth-u-code-def[*symmetric*] *nth-nat-of-uint64-nth′*
nth-aa-i32-u64-def nth-u64-code-def arl-get-u64-def arl-get′-def
nat-of-uint64-code[*symmetric*] *arl-get-u-def nat-of-uint32-code*[*symmetric*]
..

Length **definition** *length-raa-i64-u* :: ‹′a::heap arrayO-raa ⇒ uint64 ⇒ uint32 Heap› **where**
‹length-raa-i64-u xs i = do {
  x ← arl-get-u64 xs i;
  length-u-code x}›

lemma *length-raa-i64-u-alt-def*: ‹length-raa-i64-u xs i = do {
  n ← length-raa xs (nat-of-uint64 i);
  return (uint32-of-nat n)}›
unfolding *length-raa-i64-u-def length-raa-def length-u-code-def arl-get-u64-def arl-get′-def*
by (*auto simp*: *nat-of-uint64-code*)

lemma *length-raa-i64-u-rule*[*sep-heap-rules*]:
‹nat-of-uint64 b < length xs ⟹ <arlO-assn (array-assn R) xs a> length-raa-i64-u a b
<λr. arlO-assn (array-assn R) xs a ∗ ↑ (r = uint32-of-nat (length-rll xs (nat-of-uint64 b)))>$_t$›
unfolding *length-raa-i64-u-alt-def length-u-code-def*
by *sep-auto*

lemma *length-raa-i64-u-hnr*[*sepref-fr-rules*]:
shows ‹(uncurry length-raa-i64-u, uncurry (RETURN ∘∘ length-rll-n-uint32)) ∈
[λ(xs, i). i < length xs ∧ length (xs ! i) ≤ uint32-max]$_a$
(arlO-assn (array-assn R))^k ∗$_a$ uint64-nat-assn^k → uint32-nat-assn›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def length-rll-def*
nat-of-uint32-uint32-of-nat-id uint64-nat-rel-def)+

definition *length-raa-i64-u64* :: ‹′a::heap arrayO-raa ⇒ uint64 ⇒ uint64 Heap› **where**
‹length-raa-i64-u64 xs i = do {
  x ← arl-get-u64 xs i;
  length-u64-code x}›

lemma *length-raa-i64-u64-alt-def*: ‹length-raa-i64-u64 xs i = do {
  n ← length-raa xs (nat-of-uint64 i);
  return (uint64-of-nat n)}›
unfolding *length-raa-i64-u64-def length-raa-def length-u64-code-def arl-get-u64-def arl-get′-def*
by (*auto simp*: *nat-of-uint64-code*)

lemma *length-raa-i64-u64-rule*[*sep-heap-rules*]:
‹nat-of-uint64 b < length xs ⟹ <arlO-assn (array-assn R) xs a> length-raa-i64-u64 a b
<λr. arlO-assn (array-assn R) xs a ∗ ↑ (r = uint64-of-nat (length-rll xs (nat-of-uint64 b)))>$_t$›
unfolding *length-raa-i64-u64-alt-def length-u64-code-def*
by *sep-auto*

lemma *length-raa-i64-u64-hnr*[*sepref-fr-rules*]:
shows ‹(uncurry length-raa-i64-u64, uncurry (RETURN ∘∘ length-rll-n-uint32)) ∈
[λ(xs, i). i < length xs ∧ length (xs ! i) ≤ uint64-max]$_a$
(arlO-assn (array-assn R))^k ∗$_a$ uint64-nat-assn^k → uint64-nat-assn›
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def length-rll-def*
nat-of-uint64-uint64-of-nat-id uint64-nat-rel-def)+

definition *length-raa-i32-u64* :: ‹′a::heap arrayO-raa ⇒ uint32 ⇒ uint64 Heap› **where**
‹length-raa-i32-u64 xs i = do {
  x ← arl-get-u xs i;
  length-u64-code x}›

lemma *length-raa-i32-u64-alt-def*: ⟨*length-raa-i32-u64 xs i* = *do* {
  n ← *length-raa xs* (*nat-of-uint32 i*);
  return (*uint64-of-nat n*)}⟩
 unfolding *length-raa-i32-u64-def length-raa-def length-u64-code-def arl-get-u-def*
  arl-get′-def nat-of-uint32-code[*symmetric*]
 by *auto*

definition *length-rll-n-i32-uint64* **where**
 [*simp*]: ⟨*length-rll-n-i32-uint64* = *length-rll*⟩

lemma *length-raa-i32-u64-hnr*[*sepref-fr-rules*]:
 shows ⟨(*uncurry length-raa-i32-u64*, *uncurry* (*RETURN* ∘∘ *length-rll-n-i32-uint64*)) ∈
  $[\lambda(xs,\ i).\ i < length\ xs \wedge length\ (xs\ !\ i) \leq uint64\text{-}max]_a$
   $(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$⟩
 by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def length-rll-def*
   nat-of-uint64-uint64-of-nat-id length-raa-i32-u64-alt-def arl-get-u-def
   arl-get′-def nat-of-uint32-code[*symmetric*] *uint32-nat-rel-def*)+

definition *delete-index-and-swap-aa-i64* **where**
 ⟨*delete-index-and-swap-aa-i64 xs i* = *delete-index-and-swap-aa xs* (*nat-of-uint64 i*)⟩

definition *last-aa-u64* **where**
 ⟨*last-aa-u64 xs i* = *last-aa xs* (*nat-of-uint64 i*)⟩

lemma *last-aa-u64-code*[*code*]:
 ⟨*last-aa-u64 xs i* = *nth-u64-code xs i* ⤜ *arl-last*⟩
 unfolding *last-aa-u64-def last-aa-def nth-nat-of-uint32-nth′ nth-nat-of-uint32-nth′*
  arl-get-u-def[*symmetric*] *nth-u64-code-def Array.nth′-def comp-def*
  nat-of-uint64-code[*symmetric*]
 ..

definition *length-raa-i32-u* :: ⟨′*a*::*heap arrayO-raa* ⇒ *uint32* ⇒ *uint32 Heap*⟩ **where**
 ⟨*length-raa-i32-u xs i* = *do* {
   x ← *arl-get-u xs i*;
  length-u-code x}⟩

lemma *length-raa-i32-rule*[*sep-heap-rules*]:
 assumes ⟨*nat-of-uint32 b* < *length xs*⟩
 shows ⟨<*arlO-assn* (*array-assn R*) *xs a*> *length-raa-i32-u a b*
 <*λr*. *arlO-assn* (*array-assn R*) *xs a* ∗ ↑ (*r* = *uint32-of-nat* (*length-rll xs* (*nat-of-uint32 b*)))>$_t$⟩
proof −
 have *1*: ⟨*a* ∗ *b*∗ *c* = *c* ∗ *a* ∗*b*⟩ **for** *a b c* :: *assn*
  by (*auto simp*: *ac-simps*)
 have [*sep-heap-rules*]: ⟨<*arlO-assn-except* (*array-assn R*) [*nat-of-uint32 b*] *xs a*
    (*λr′*. *array-assn R* (*xs* ! *nat-of-uint32 b*) *x* ∗
      ↑ (*x* = *r′* ! *nat-of-uint32 b*))>
   Array.len x <*λr*. *arlO-assn* (*array-assn R*) *xs a* ∗
      ↑ (*r* = *length* (*xs* ! *nat-of-uint32 b*))>⟩
  for *x*
  unfolding *arlO-assn-except-def*
  apply (*subst arlO-assn-except-array0-index*[*symmetric*, *OF assms*])

apply *sep-auto*
apply (*subst 1*)
by (*sep-auto simp*: *array-assn-def is-array-def hr-comp-def list-rel-imp-same-length arlO-assn-except-def*)
show *?thesis*
using *assms*
unfolding *length-raa-i32-u-def length-u-code-def arl-get-u-def arl-get′-def length-rll-def*
by (*sep-auto simp*: *nat-of-uint32-code*[*symmetric*])
qed

lemma *length-raa-i32-u-hnr*[*sepref-fr-rules*]:
shows ‹(*uncurry length-raa-i32-u*, *uncurry* (*RETURN* ∘∘ *length-rll-n-uint32*)) ∈ $[\lambda(xs, i).\ i < \mathit{length}\ xs \wedge \mathit{length}\ (xs\ !\ i) \leq \mathit{uint32\text{-}max}]_a$ $(\mathit{arlO\text{-}assn}\ (\mathit{array\text{-}assn}\ R))^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k \rightarrow \mathit{uint32\text{-}nat\text{-}assn}$›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def br-def length-rll-def nat-of-uint32-uint32-of-nat-id*)+

definition (**in** −)*length-aa-u64-o64* :: ‹(′*a*::*heap array-list*) *array* ⇒ *uint64* ⇒ *uint64 Heap*› **where**
‹*length-aa-u64-o64 xs i* = *length-aa-u64 xs i* >>= (λ*n*. *return* (*uint64-of-nat n*))›

definition *arl-length-o64* **where**
‹*arl-length-o64 x* = *do* {*n* ← *arl-length x*; *return* (*uint64-of-nat n*)}›

lemma *length-aa-u64-o64-code*[*code*]:
‹*length-aa-u64-o64 xs i* = *nth-u64-code xs i* ⤜ *arl-length-o64*›
unfolding *length-aa-u64-o64-def length-aa-u64-def nth-u-def*[*symmetric*] *nth-u64-code-def Array.nth′-def arl-length-o64-def length-aa-def*
by (*auto simp*: *nat-of-uint32-code nat-of-uint64-code*[*symmetric*])

lemma *length-aa-u64-o64-hnr*[*sepref-fr-rules*]:
‹(*uncurry length-aa-u64-o64*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈ $[\lambda(xs, i).\ i < \mathit{length}\ xs \wedge \mathit{length}\ (xs\ !\ i) \leq \mathit{uint64\text{-}max}]_a$ $(\mathit{arrayO\text{-}assn}\ (\mathit{arl\text{-}assn}\ R))^k *_a \mathit{uint64\text{-}nat\text{-}assn}^k \rightarrow \mathit{uint64\text{-}nat\text{-}assn}$›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def length-aa-u64-o64-def br-def length-aa-u64-def uint64-nat-rel-def nat-of-uint64-uint64-of-nat-id length-ll-def*)

definition (**in** −)*length-aa-u32-o64* :: ‹(′*a*::*heap array-list*) *array* ⇒ *uint32* ⇒ *uint64 Heap*› **where**
‹*length-aa-u32-o64 xs i* = *length-aa-u xs i* >>= (λ*n*. *return* (*uint64-of-nat n*))›

lemma *length-aa-u32-o64-code*[*code*]:
‹*length-aa-u32-o64 xs i* = *nth-u-code xs i* ⤜ *arl-length-o64*›
unfolding *length-aa-u32-o64-def length-aa-u64-def nth-u-def*[*symmetric*] *nth-u-code-def Array.nth′-def arl-length-o64-def length-aa-u-def length-aa-def*
by (*auto simp*: *nat-of-uint64-code*[*symmetric*] *nat-of-uint32-code*[*symmetric*])

lemma *length-aa-u32-o64-hnr*[*sepref-fr-rules*]:
‹(*uncurry length-aa-u32-o64*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈ $[\lambda(xs, i).\ i < \mathit{length}\ xs \wedge \mathit{length}\ (xs\ !\ i) \leq \mathit{uint64\text{-}max}]_a$ $(\mathit{arrayO\text{-}assn}\ (\mathit{arl\text{-}assn}\ R))^k *_a \mathit{uint32\text{-}nat\text{-}assn}^k \rightarrow \mathit{uint64\text{-}nat\text{-}assn}$›
by *sepref-to-hoare* (*sep-auto simp*: *uint32-nat-rel-def length-aa-u32-o64-def br-def length-aa-u64-def uint64-nat-rel-def nat-of-uint64-uint64-of-nat-id length-ll-def length-aa-u-def*)

definition *length-raa-u32* :: ‹'a::heap arrayO-raa ⇒ uint32 ⇒ nat Heap› **where**
‹length-raa-u32 xs i = do {
  x ← arl-get-u xs i;
  Array.len x}›

lemma *length-raa-u32-rule*[*sep-heap-rules*]:
‹b < length xs ⟹ (b', b) ∈ uint32-nat-rel ⟹ <arlO-assn (array-assn R) xs a> length-raa-u32 a b'
<λr. arlO-assn (array-assn R) xs a ∗ ↑ (r = length-rll xs b)>$_t$›
supply *arrayO-raa-nth-rule*[*sep-heap-rules*]
unfolding *length-raa-u32-def arl-get-u-def arl-get'-def uint32-nat-rel-def br-def*
apply (*cases a*)
apply (*sep-auto simp*: *nat-of-uint32-code*[*symmetric*])
apply (*sep-auto simp*: *arlO-assn-except-def arl-length-def array-assn-def*
eq-commute[*of* ‹(-, -)›] *is-array-def hr-comp-def length-rll-def*
dest: *list-all2-lengthD*)
apply (*sep-auto simp*: *arlO-assn-except-def arl-length-def arl-assn-def*
hr-comp-def[*abs-def*] *arl-get'-def*
eq-commute[*of* ‹(-, -)›] *is-array-list-def hr-comp-def length-rll-def list-rel-def*
dest: *list-all2-lengthD*)[]
unfolding *arlO-assn-def*[*symmetric*] *arl-assn-def*[*symmetric*]
apply (*subst arlO-assn-except-array0-index*[*symmetric*, *of b*])
apply *simp*
unfolding *arlO-assn-except-def arl-assn-def hr-comp-def is-array-def*
apply *sep-auto*
done

lemma *length-raa-u32-hnr*[*sepref-fr-rules*]:
‹(uncurry length-raa-u32, uncurry (RETURN ∘∘ length-rll)) ∈
[λ(xs, i). i < length xs]$_a$ (arlO-assn (array-assn R))^k $*_a$ uint32-nat-assn^k → nat-assn›
by *sepref-to-hoare sep-auto*

definition *length-raa-u32-u64* :: ‹'a::heap arrayO-raa ⇒ uint32 ⇒ uint64 Heap› **where**
‹length-raa-u32-u64 xs i = do {
  x ← arl-get-u xs i;
  length-u64-code x}›

lemma *length-raa-u32-u64-hnr*[*sepref-fr-rules*]:
shows ‹(uncurry length-raa-u32-u64, uncurry (RETURN ∘∘ length-rll-n-uint64)) ∈
[λ(xs, i). i < length xs ∧ length (xs ! i) ≤ uint64-max]$_a$
(arlO-assn (array-assn R))^k $*_a$ uint32-nat-assn^k → uint64-nat-assn›
proof −
have *1*: ‹a ∗ b ∗ c = c ∗ a ∗ b› **for** *a b c* :: *assn*
by (*auto simp*: *ac-simps*)
have *H*: ‹<arlO-assn-except (array-assn R) [nat-of-uint32 bi] a (aa, ba)
(λr'. array-assn R (a ! nat-of-uint32 bi) x ∗
↑ (x = r' ! nat-of-uint32 bi))>
Array.len x <λr. ↑(r = length (a ! nat-of-uint32 bi)) ∗
arlO-assn (array-assn R) a (aa, ba)>›
if
‹nat-of-uint32 bi < length a› **and**
‹length (a ! nat-of-uint32 bi) ≤ uint64-max›
for *bi* :: ‹uint32› **and** *a* :: ‹'b list list› **and** *aa* :: ‹'a array array› **and** *ba* :: ‹nat› **and**
x :: ‹'a array›
proof −

show *?thesis*
using *that* **apply** −
apply (*subst arlO-assn-except-array0-index*[*symmetric*, *OF that*(*1*)])
by (*sep-auto simp*: *array-assn-def arl-get-def hr-comp-def is-array-def list-rel-imp-same-length arlO-assn-except-def*)
qed
show *?thesis*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *uint64-nat-rel-def br-def length-rll-def nat-of-uint64-uint64-of-nat-id length-raa-u32-u64-def arl-get-u-def arl-get′-def uint32-nat-rel-def nat-of-uint32-code*[*symmetric*] *length-u64-code-def intro*!:)+
apply (*rule H*; *assumption*)
apply (*sep-auto simp*: *array-assn-def arl-get-def nat-of-uint64-uint64-of-nat-id*)
done
qed

definition *length-raa-u64-u64* :: ⟨′*a*::*heap arrayO-raa* ⇒ *uint64* ⇒ *uint64 Heap*⟩ **where**
⟨*length-raa-u64-u64 xs i* = *do* {
x ← *arl-get-u64 xs i*;
length-u64-code x}⟩

lemma *length-raa-u64-u64-hnr*[*sepref-fr-rules*]:
shows ⟨(*uncurry length-raa-u64-u64*, *uncurry* (*RETURN* ∘∘ *length-rll-n-uint64*)) ∈ $[\lambda(xs,\ i).\ i < length\ xs \wedge length\ (xs\ !\ i) \leq uint64\text{-}max]_a$ $(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$⟩
proof −
have *1*: ⟨*a* ∗ *b* ∗ *c* = *c* ∗ *a* ∗ *b*⟩ **for** *a b c* :: *assn*
by (*auto simp*: *ac-simps*)
have *H*: ⟨<*arlO-assn-except* (*array-assn R*) [*nat-of-uint64 bi*] *a* (*aa*, *ba*)
(λ*r*′. *array-assn R* (*a* ! *nat-of-uint64 bi*) *x* ∗
↑ (*x* = *r*′ ! *nat-of-uint64 bi*))>
Array.len x <λ*r*. ↑(*r* = *length* (*a* ! *nat-of-uint64 bi*)) ∗
arlO-assn (*array-assn R*) *a* (*aa*, *ba*)>⟩
if
⟨*nat-of-uint64 bi* < *length a*⟩ **and**
⟨*length* (*a* ! *nat-of-uint64 bi*) ≤ *uint64-max*⟩
for *bi* :: ⟨*uint64*⟩ **and** *a* :: ⟨′*b list list*⟩ **and** *aa* :: ⟨′*a array array*⟩ **and** *ba* :: ⟨*nat*⟩ **and** *x* :: ⟨′*a array*⟩
proof −
show *?thesis*
using *that* **apply** −
apply (*subst arlO-assn-except-array0-index*[*symmetric*, *OF that*(*1*)])
by (*sep-auto simp*: *array-assn-def arl-get-def hr-comp-def is-array-def list-rel-imp-same-length arlO-assn-except-def*)
qed
show *?thesis*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *uint64-nat-rel-def br-def length-rll-def nat-of-uint64-uint64-of-nat-id length-raa-u32-u64-def arl-get-u64-def arl-get′-def uint32-nat-rel-def nat-of-uint32-code*[*symmetric*] *length-u64-code-def length-raa-u64-u64-def nat-of-uint64-code*[*symmetric*]
intro!:)+
apply (*rule H*; *assumption*)
apply (*sep-auto simp*: *array-assn-def arl-get-def nat-of-uint64-uint64-of-nat-id*)
done

qed

definition *length-arlO-u* **where**
⟨*length-arlO-u xs* = *do* {
n ← *length-ra xs*;
return (*uint32-of-nat n*)}⟩

lemma *length-arlO-u*[*sepref-fr-rules*]:
⟨(*length-arlO-u*, *RETURN o length-uint32-nat*) ∈ $[\lambda xs.\ length\ xs \le uint32\text{-}max]_a\ (arlO\text{-}assn\ R)^k \to$ *uint32-nat-assn*⟩
by *sepref-to-hoare*
(*sep-auto simp*: *length-arlO-u-def arl-length-def uint32-nat-rel-def*
br-def nat-of-uint32-uint32-of-nat-id)

definition *arl-length-u64-code* **where**
⟨*arl-length-u64-code C* = *do* {
n ← *arl-length C*;
return (*uint64-of-nat n*)
}⟩

lemma *arl-length-u64-code*[*sepref-fr-rules*]:
⟨(*arl-length-u64-code*, *RETURN o length-uint64-nat*) ∈
$[\lambda xs.\ length\ xs \le uint64\text{-}max]_a\ (arl\text{-}assn\ R)^k \to uint64\text{-}nat\text{-}assn$⟩
by *sepref-to-hoare*
(*sep-auto simp*: *arl-length-u64-code-def arl-length-def uint64-nat-rel-def*
br-def nat-of-uint64-uint64-of-nat-id arl-assn-def hr-comp-def[*abs-def*]
is-array-list-def dest: *list-rel-imp-same-length*)

Setters **definition** *update-aa-u64* **where**
⟨*update-aa-u64 xs i j* = *update-aa xs* (*nat-of-uint64 i*) *j*⟩

definition *Array-upd-u64* **where**
⟨*Array-upd-u64 i x a* = *Array.upd* (*nat-of-uint64 i*) *x a*⟩

lemma *Array-upd-u64-code*[*code*]: ⟨*Array-upd-u64 i x a* = *heap-array-set′-u64 a i x* ≫ *return a*⟩
unfolding *Array-upd-u64-def heap-array-set′-u64-def*
Array.upd′-def
by (*auto simp*: *nat-of-uint64-code upd-return*)

lemma *update-aa-u64-code*[*code*]:
⟨*update-aa-u64 a i j y* = *do* {
x ← *nth-u64-code a i*;
a′ ← *arl-set x j y*;
Array-upd-u64 i a′ a
}⟩
unfolding *update-aa-u64-def update-aa-def nth-nat-of-uint32-nth′ nth-nat-of-uint32-nth′*
arl-get-u-def[*symmetric*] *nth-u64-code-def Array.nth′-def comp-def*
heap-array-set′-u-def[*symmetric*] *Array-upd-u64-def nat-of-uint64-code*[*symmetric*]
by *auto*

definition *set-butlast-aa-u64* **where**
⟨*set-butlast-aa-u64 xs i* = *set-butlast-aa xs* (*nat-of-uint64 i*)⟩

lemma *set-butlast-aa-u64-code*[*code*]:

‹*set-butlast-aa-u64 a i* = *do* {
  x ← *nth-u64-code a i*;
  a′ ← *arl-butlast x*;
  Array-upd-u64 i a′ a
}› — Replace the *i*-th element by the itself except the last element.
unfolding *set-butlast-aa-u64-def set-butlast-aa-def*
nth-u64-code-def Array-upd-u64-def
by (*auto simp*: *Array.nth′-def nat-of-uint64-code*)

lemma *delete-index-and-swap-aa-i64-code*[*code*]:
‹*delete-index-and-swap-aa-i64 xs i j* = *do* {
  x ← *last-aa-u64 xs i*;
  xs ← *update-aa-u64 xs i j x*;
  set-butlast-aa-u64 xs i
}›
unfolding *delete-index-and-swap-aa-i64-def delete-index-and-swap-aa-def*
last-aa-u64-def update-aa-u64-def set-butlast-aa-u64-def
by *auto*

lemma *delete-index-and-swap-aa-i64-ll-hnr-u*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry2 delete-index-and-swap-aa-i64*, *uncurry2* (*RETURN ooo delete-index-and-swap-ll*))
$\in [\lambda((l,i), j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a uint64\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k$
$\rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
using *assms* **unfolding** *delete-index-and-swap-aa-def delete-index-and-swap-aa-i64-def*
by *sepref-to-hoare* (*sep-auto dest*: *le-length-ll-nemptyD*
simp: *delete-index-and-swap-ll-def update-ll-def last-ll-def set-butlast-ll-def*
length-ll-def[*symmetric*] *uint32-nat-rel-def br-def uint64-nat-rel-def*)

definition *delete-index-and-swap-aa-i32-u64* **where**
‹*delete-index-and-swap-aa-i32-u64 xs i j* =
delete-index-and-swap-aa xs (*nat-of-uint32 i*) (*nat-of-uint64 j*)›

definition *update-aa-u32-i64* **where**
‹*update-aa-u32-i64 xs i j* = *update-aa xs* (*nat-of-uint32 i*) (*nat-of-uint64 j*)›

lemma *update-aa-u32-i64-code*[*code*]:
‹*update-aa-u32-i64 a i j y* = *do* {
  x ← *nth-u-code a i*;
  a′ ← *arl-set-u64 x j y*;
  Array-upd-u i a′ a
}›
unfolding *update-aa-u32-i64-def update-aa-def nth-nat-of-uint32-nth′ nth-nat-of-uint32-nth′*
arl-get-u-def[*symmetric*] *nth-u-code-def Array.nth′-def comp-def arl-set′-u64-def*
heap-array-set′-u-def[*symmetric*] *Array-upd-u-def nat-of-uint64-code*[*symmetric*]
nat-of-uint32-code arl-set-u64-def
by *auto*

lemma *delete-index-and-swap-aa-i32-u64-code*[*code*]:
‹*delete-index-and-swap-aa-i32-u64 xs i j* = *do* {
  x ← *last-aa-u xs i*;
  xs ← *update-aa-u32-i64 xs i j x*;

set-butlast-aa-u xs i
}›
unfolding *delete-index-and-swap-aa-i32-u64-def delete-index-and-swap-aa-def last-aa-u-def update-aa-u-def set-butlast-aa-u-def update-aa-u32-i64-def*
by *auto*

lemma *delete-index-and-swap-aa-i32-u64-ll-hnr-u*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry2 delete-index-and-swap-aa-i32-u64*, *uncurry2* (*RETURN ooo delete-index-and-swap-ll*)) $\in [\lambda((l,i), j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
using *assms* **unfolding** *delete-index-and-swap-aa-def delete-index-and-swap-aa-i32-u64-def*
by *sepref-to-hoare* (*sep-auto dest*: *le-length-ll-nemptyD*
simp: *delete-index-and-swap-ll-def update-ll-def last-ll-def set-butlast-ll-def length-ll-def*[*symmetric*] *uint32-nat-rel-def br-def uint64-nat-rel-def*)

Swap **definition** *swap-aa-i32-u64* :: (*'a*::{*heap*,*default*}) *arrayO-raa* ⇒ *uint32* ⇒ *uint64* ⇒ *uint64* ⇒ *'a arrayO-raa Heap* **where**
‹*swap-aa-i32-u64 xs k i j* = *do* {
  xi ← *arl-get-u xs k*;
  xj ← *swap-u64-code xi i j*;
  xs ← *arl-set-u xs k xj*;
  return xs
}›

lemma *swap-aa-i32-u64-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 swap-aa-i32-u64*, *uncurry3* (*RETURN oooo swap-ll*)) ∈
$[\lambda(((xs, k), i), j).\ k < length\ xs \wedge i < length\text{-}rll\ xs\ k \wedge j < length\text{-}rll\ xs\ k]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^d *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow$
(*arlO-assn* (*array-assn R*))›
proof −
note *update-raa-rule-pure*[*sep-heap-rules*]
obtain *R'* **where** *R'*: ‹*R'* = *the-pure R*› **and** *RR'*: ‹*R* = *pure R'*›
  using *assms* **by** *fastforce*
have [*simp*]: ‹*the-pure* (λ*a b*. ↑ ((*b*, *a*) ∈ *R'*)) = *R'*›
  unfolding *pure-def*[*symmetric*] **by** *auto*
have *H*: ‹<*is-array-list p* (*aa*, *bc*) ∗
    heap-list-all-nth (*array-assn* (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*))) (*remove1 bb* [*0*..<*length p*]) *a p* ∗
    array-assn (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*)) (*a* ! *bb*) (*p* ! *bb*)>
   Array.nth (*p* ! *bb*) (*nat-of-integer* (*integer-of-uint64 bia*))
   <λ*r*. $\exists_A$ *p'*. *is-array-list p'* (*aa*, *bc*) ∗ ↑ (*bb* < *length p'* ∧ *p'* ! *bb* = *p* ! *bb* ∧ *length a* = *length p'*) ∗
     heap-list-all-nth (*array-assn* (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*))) (*remove1 bb* [*0*..<*length p'*]) *a p'* ∗
    array-assn (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*)) (*a* ! *bb*) (*p'* ! *bb*) ∗
    R (*a* ! *bb* ! (*nat-of-uint64 bia*)) *r* >›
  if
   ‹*is-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*))› **and**
   ‹*bb* < *length p*› **and**
   ‹*nat-of-uint64 bia* < *length* (*a* ! *bb*)› **and**
   ‹*nat-of-uint64 bi* < *length* (*a* ! *bb*)› **and**
   ‹*length a* = *length p*›
  for *bi* :: ‹*uint64*› **and** *bia* :: ‹*uint64*› **and** *bb* :: ‹*nat*› **and** *a* :: ‹*'a list list*› **and**
   aa :: ‹*'b array array*› **and** *bc* :: ‹*nat*› **and** *p* :: ‹*'b array list*›
  using *that*
  by (*sep-auto simp*: *array-assn-def hr-comp-def is-array-def nat-of-uint64-code*[*symmetric*]

list-rel-imp-same-length RR′ pure-def param-nth)
have *H′*: ‹*is-array-list p′* (*aa*, *ba*) ∗ *p′* ! *bb* $\mapsto_a$ *b* [*nat-of-uint64 bia* := *b* ! *nat-of-uint64 bi*,
nat-of-uint64 bi := *xa*] ∗
heap-list-all-nth (λ*a b*. $\exists_A$*ba*. *b* $\mapsto_a$ *ba* ∗ ↑ ((*ba*, *a*) ∈ ⟨*R′*⟩*list-rel*))
(*remove1 bb* [*0*..<*length p′*]) *a p′* ∗ *R* (*a* ! *bb* ! *nat-of-uint64 bia*) *xa* $\Longrightarrow_A$
is-array-list p′ (*aa*, *ba*) ∗
heap-list-all
(λ*a c*. $\exists_A$*b*. *c* $\mapsto_a$ *b* ∗ ↑ ((*b*, *a*) ∈ ⟨*R′*⟩*list-rel*))
(*a*[*bb* := (*a* ! *bb*) [*nat-of-uint64 bia* := *a* ! *bb* ! *nat-of-uint64 bi*,
nat-of-uint64 bi := *a* ! *bb* ! *nat-of-uint64 bia*]])
p′ ∗ *true*›
if
‹*is-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *R′*))› **and**
le: ‹*nat-of-uint64 bia* < *length* (*a* ! *bb*)› **and**
le′: ‹*nat-of-uint64 bi* < *length* (*a* ! *bb*)› **and**
‹*bb* < *length p′*› **and**
‹*length a* = *length p′*› **and**
a: ‹(*b*, *a* ! *bb*) ∈ ⟨*R′*⟩*list-rel*›
for *bi* :: ‹*uint64*› **and** *bia* :: ‹*uint64*› **and** *bb* :: ‹*nat*› **and** *a* :: ‹*′a list list*› **and**
xa :: ‹*′b*› **and** *p′* :: ‹*′b array list*› **and** *b* :: ‹*′b list*› **and** *aa* :: ‹*′b array array*› **and** *ba* :: ‹*nat*›
proof −
have *1*: ‹(*b*[*nat-of-uint64 bia* := *b* ! *nat-of-uint64 bi*, *nat-of-uint64 bi* := *xa*],
(*a* ! *bb*)[*nat-of-uint64 bia* := *a* ! *bb* ! *nat-of-uint64 bi*,
nat-of-uint64 bi := *a* ! *bb* ! *nat-of-uint64 bia*]) ∈ ⟨*R′*⟩*list-rel*›
if ‹(*xa*, *a* ! *bb* ! *nat-of-uint64 bia*) ∈ *R′*›
using *that a le le′*
unfolding *list-rel-def list-all2-conv-all-nth*
by *auto*
have *2*: ‹*heap-list-all-nth* (λ*a b*. $\exists_A$*ba*. *b* $\mapsto_a$ *ba* ∗ ↑ ((*ba*, *a*) ∈ ⟨*R′*⟩*list-rel*)) (*remove1 bb* [*0*..<*length p′*]) *a p′* =
heap-list-all-nth (λ*a c*. $\exists_A$*b*. *c* $\mapsto_a$ *b* ∗ ↑ ((*b*, *a*) ∈ ⟨*R′*⟩*list-rel*)) (*remove1 bb* [*0*..<*length p′*])
(*a*[*bb* := (*a* ! *bb*)[*nat-of-uint64 bia* := *a* ! *bb* ! *nat-of-uint64 bi*, *nat-of-uint64 bi* := *a* ! *bb* ! *nat-of-uint64 bia*]]) *p′*›
by (*rule heap-list-all-nth-cong*) *auto*
show *?thesis* **using** *that*
unfolding *heap-list-all-heap-list-all-nth-eq*
by (*subst* (*2*) *heap-list-all-nth-remove1*[*of bb*])
(*sep-auto simp*: *heap-list-all-heap-list-all-nth-eq swap-def fr-refl RR′*
pure-def 2[*symmetric*] *intro*!: *1*)+
qed

show *?thesis*
using *assms* **unfolding** *R′*[*symmetric*] **unfolding** *RR′*
apply *sepref-to-hoare*

apply (*sep-auto simp*: *swap-aa-i32-u64-def swap-ll-def arlO-assn-except-def length-rll-def*
length-rll-update-rll nth-raa-i-u64-def uint64-nat-rel-def br-def
swap-def nth-rll-def list-update-swap swap-u64-code-def nth-u64-code-def Array.nth′-def
heap-array-set-u64-def heap-array-set′-u64-def arl-assn-def
Array.upd′-def)
apply (*rule H*; *assumption*)
apply (*sep-auto simp*: *array-assn-def nat-of-uint64-code*[*symmetric*] *hr-comp-def is-array-def*
list-rel-imp-same-length arlO-assn-def arl-assn-def hr-comp-def[*abs-def*] *arl-set-u-def*
arl-set′-u-def list-rel-pres-length uint32-nat-rel-def br-def)
apply (*rule H′*; *assumption*)
done

qed

Conversion from list of lists of *nat* to list of lists of *uint64*

sepref-definition *array-nat-of-uint64-code*
  is *array-nat-of-uint64*
  :: ‹$(array\text{-}assn\ uint64\text{-}nat\text{-}assn)^k \rightarrow_a array\text{-}assn\ nat\text{-}assn$›
  unfolding *op-map-def array-nat-of-uint64-def array-fold-custom-replicate*
  apply (*rewrite at* ‹*do* {*let* - = ⌑; -}› *annotate-assn*[**where** *A*=‹*array-assn nat-assn*›])
  by *sepref*

lemma *array-nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
  ‹(*array-nat-of-uint64-code*, (*RETURN* ∘ *array-nat-of-uint64-conv*))
    ∈ $(array\text{-}assn\ uint64\text{-}nat\text{-}assn)^k \rightarrow_a array\text{-}assn\ nat\text{-}assn$›
  using *array-nat-of-uint64-code.refine*[*unfolded array-nat-of-uint64-def*,
    FCOMP op-map-map-rel[*unfolded convert-fref*]] **unfolding** *array-nat-of-uint64-conv-alt-def*
  by *simp*

sepref-definition *array-uint64-of-nat-code*
  is *array-uint64-of-nat*
  :: ‹$[\lambda xs.\ \forall a{\in}set\ xs.\ a \le uint64\text{-}max]_a$
      $(array\text{-}assn\ nat\text{-}assn)^k \rightarrow array\text{-}assn\ uint64\text{-}nat\text{-}assn$›
  supply [[*goals-limit=1*]]
  unfolding *op-map-def array-uint64-of-nat-def array-fold-custom-replicate*
  apply (*rewrite at* ‹*do* {*let* - = ⌑; -}› *annotate-assn*[**where** *A*=‹*array-assn uint64-nat-assn*›])
  by *sepref*

lemma *array-uint64-of-nat-conv-alt-def*:
  ‹*array-uint64-of-nat-conv* = *map uint64-of-nat-conv*›
  unfolding *uint64-of-nat-conv-def array-uint64-of-nat-conv-def* **by** *auto*

lemma *array-uint64-of-nat-conv-hnr*[*sepref-fr-rules*]:
  ‹(*array-uint64-of-nat-code*, (*RETURN* ∘ *array-uint64-of-nat-conv*))
    ∈ $[\lambda xs.\ \forall a{\in}set\ xs.\ a \le uint64\text{-}max]_a$
      $(array\text{-}assn\ nat\text{-}assn)^k \rightarrow array\text{-}assn\ uint64\text{-}nat\text{-}assn$›
  using *array-uint64-of-nat-code.refine*[*unfolded array-uint64-of-nat-def*,
    FCOMP op-map-map-rel[*unfolded convert-fref*]] **unfolding** *array-uint64-of-nat-conv-alt-def*
  by *simp*

definition *swap-arl-u64* **where**
  ‹*swap-arl-u64* = (λ(*xs*, *n*) *i j*. *do* {
    ki ← *nth-u64-code xs i*;
    kj ← *nth-u64-code xs j*;
    xs ← *heap-array-set-u64 xs i kj*;
    xs ← *heap-array-set-u64 xs j ki*;
    return (*xs*, *n*)
  })›

lemma *swap-arl-u64-hnr*[*sepref-fr-rules*]:
  ‹(*uncurry2 swap-arl-u64*, *uncurry2* (*RETURN ooo op-list-swap*)) ∈
  $[pre\text{-}list\text{-}swap]_a\ (arl\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow arl\text{-}assn\ A$›
  unfolding *swap-arl-u64-def arl-assn-def is-array-list-def hr-comp-def*
    nth-u64-code-def Array.nth′-def heap-array-set-u64-def heap-array-set-def
    heap-array-set′-u64-def Array.upd′-def
  apply *sepref-to-hoare*
  apply (*sep-auto simp*: *nat-of-uint64-code*[*symmetric*] *uint64-nat-rel-def br-def*

list-rel-imp-same-length[*symmetric*] *swap-def*)
apply (*subst-tac n*=‹*bb*› **in** *nth-take*[*symmetric*])
apply (*simp*; *fail*)
apply (*subst-tac* (*2*) *n*=‹*bb*› **in** *nth-take*[*symmetric*])
apply (*simp*; *fail*)
by (*sep-auto simp*: *nat-of-uint64-code*[*symmetric*] *uint64-nat-rel-def br-def*
list-rel-imp-same-length[*symmetric*] *swap-def IICF-List.swap-def*
simp del: *nth-take*
intro!: *list-rel-update′ param-nth*)

definition *butlast-nonresizing* :: ‹*′a list* ⇒ *′a list*›**where**
[*simp*]: ‹*butlast-nonresizing* = *butlast*›

definition *arl-butlast-nonresizing* :: ‹*′a array-list* ⇒ *′a array-list*› **where**
‹*arl-butlast-nonresizing* = (λ(*xs*, *a*). (*xs*, *fast-minus a 1*))›

lemma *butlast-nonresizing-hnr*[*sepref-fr-rules*]:
‹(*return o arl-butlast-nonresizing*, *RETURN o butlast-nonresizing*) ∈
$[\lambda xs.\ xs \neq []]_a$ $(arl\text{-}assn\ R)^d \rightarrow arl\text{-}assn\ R$›
by *sepref-to-hoare*
(*sep-auto simp*: *arl-butlast-nonresizing-def arl-assn-def hr-comp-def*
is-array-list-def butlast-take list-rel-imp-same-length
dest:
list-rel-butlast[*of* ‹*take - -*›])

lemma *update-aa-u64-rule*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› **and** ‹*bb* < *length a*› **and** ‹*ba* < *length-ll a bb*› **and** ‹(*bb′*, *bb*) ∈ *uint32-nat-rel*›
and
‹(*ba′*, *ba*) ∈ *uint64-nat-rel*›
shows ‹<*R b bi* ∗ *arrayO-assn* (*arl-assn R*) *a ai*> *update-aa-u32-i64 ai bb′ ba′ bi*
<λ*r*. *R b bi* ∗ ($\exists_A$*x*. *arrayO-assn* (*arl-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
using *assms*
by (*sep-auto simp add*: *update-aa-u32-i64-def update-ll-def p uint64-nat-rel-def uint32-nat-rel-def br-def*)

lemma *update-aa-u32-i64-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa-u32-i64*, *uncurry3* (*RETURN oooo update-ll*)) ∈
$[\lambda(((l,i), j), x).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a R^k \rightarrow (arrayO\text{-}assn$
$(arl\text{-}assn\ R))$›
by *sepref-to-hoare* (*sep-auto simp*: *assms*)

lemma *min-uint64-nat-assn*:
‹(*uncurry* (*return oo min*), *uncurry* (*RETURN oo min*)) ∈ $uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a$
uint64-nat-assn›
by (*sepref-to-hoare*; *sep-auto simp*: *br-def uint64-nat-rel-def min-def nat-of-uint64-le-iff*)

lemma *nat-of-uint64-shiftl*: ‹*nat-of-uint64* (*xs* >> *a*) = *nat-of-uint64 xs* >> *a*›
by *transfer* (*auto simp*: *unat-shiftr nat-shifl-div*)

lemma *bit-lshift-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (>>)), *uncurry* (*RETURN oo* (>>))) ∈
$uint64\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a uint64\text{-}nat\text{-}assn$›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-shiftl*)

lemma [*code*]: *uint32-max-uint32* = *4294967295*
  using *nat-of-uint32-uint32-max-uint32*
  by (*auto simp*: *uint32-max-uint32-def uint32-max-def*)

end
theory *IICF-Array-List64*
imports
  Refine-Imperative-HOL.IICF-List
  Separation-Logic-Imperative-HOL.Array-Blit
  Array-UInt
  WB-Word-Assn
begin

type-synonym *'a array-list64* = *'a Heap.array* × *uint64*

definition *is-array-list64* $l \equiv \lambda(a,n).\ \exists_A l'.\ a \mapsto_a l' * \uparrow(nat\text{-}of\text{-}uint64\ n \leq length\ l' \wedge l = take\ (nat\text{-}of\text{-}uint64\ n)\ l' \wedge length\ l'{>}0 \wedge nat\text{-}of\text{-}uint64\ n \leq uint64\text{-}max \wedge length\ l' \leq uint64\text{-}max)$

lemma *is-array-list64-prec*[*safe-constraint-rules*]: *precise is-array-list64*
  unfolding *is-array-list64-def*[*abs-def*]
  apply(*rule preciseI*)
  apply(*simp split*: *prod.splits*)
 using *preciseD snga-prec* **by** *fastforce*

definition *arl64-empty* ≡ *do* {
  a ← *Array.new initial-capacity default*;
  return (*a*,*0*)
}

definition *arl64-empty-sz init-cap* ≡ *do* {
  a ← *Array.new* (*min uint64-max* (*max init-cap minimum-capacity*)) *default*;
  return (*a*,*0*)
}

definition *uint64-max-uint64* :: *uint64* **where**
  ‹*uint64-max-uint64* = *2* ^*64* − *1*›

definition *arl64-append* ≡ λ(*a*,*n*) *x*. *do* {
  len ← *length-u64-code a*;

  if n<*len then do* {
    a ← *Array-upd-u64 n x a*;
    return (*a*,*n*+*1*)
  } *else do* {
    let newcap = (*if len* < *uint64-max-uint64* >> *1 then 2* ∗ *len else uint64-max-uint64*);
    a ← *array-grow a* (*nat-of-uint64 newcap*) *default*;
    a ← *Array-upd-u64 n x a*;
    return (*a*,*n*+*1*)
  }
}

definition *arl64-copy* ≡ λ(*a*,*n*). *do* {
  a ← *array-copy a*;
  return (*a*,*n*)
}

definition *arl64-length* :: *'a::heap array-list64* ⇒ *uint64 Heap* **where**
arl64-length ≡ λ(*a*,*n*). *return* (*n*)

definition *arl64-is-empty* :: *'a::heap array-list64* ⇒ *bool Heap* **where**
arl64-is-empty ≡ λ(*a*,*n*). *return* (*n*=*0*)

definition *arl64-last* :: *'a::heap array-list64* ⇒ *'a Heap* **where**
arl64-last ≡ λ(*a*,*n*). *do* {
nth-u64-code a (*n* − *1*)
}

definition *arl64-butlast* :: *'a::heap array-list64* ⇒ *'a array-list64 Heap* **where**
arl64-butlast ≡ λ(*a*,*n*). *do* {
let n = *n* − *1*;
len ← *length-u64-code a*;
if (*n*∗*4* < *len* ∧ *nat-of-uint64 n*∗*2*≥*minimum-capacity*) *then do* {
a ← *array-shrink a* (*nat-of-uint64 n*∗*2*);
return (*a*,*n*)
} *else*
return (*a*,*n*)
}

definition *arl64-get* :: *'a::heap array-list64* ⇒ *uint64* ⇒ *'a Heap* **where**
arl64-get ≡ λ(*a*,*n*) *i*. *nth-u64-code a i*

definition *arl64-set* :: *'a::heap array-list64* ⇒ *uint64* ⇒ *'a* ⇒ *'a array-list64 Heap* **where**
arl64-set ≡ λ(*a*,*n*) *i x*. *do* { *a* ← *heap-array-set-u64 a i x*; *return* (*a*,*n*)}

lemma *arl64-empty-rule*[*sep-heap-rules*]: < *emp* > *arl64-empty* <*is-array-list64* []>
by (*sep-auto simp*: *arl64-empty-def is-array-list64-def initial-capacity-def uint64-max-def*)

lemma *arl64-empty-sz-rule*[*sep-heap-rules*]: < *emp* > *arl64-empty-sz N* <*is-array-list64* []>
by (*sep-auto simp*: *arl64-empty-sz-def is-array-list64-def minimum-capacity-def uint64-max-def*)

lemma *arl64-copy-rule*[*sep-heap-rules*]: < *is-array-list64 l a* > *arl64-copy a* <λ*r*. *is-array-list64 l a* ∗ *is-array-list64 l r*>
by (*sep-auto simp*: *arl64-copy-def is-array-list64-def*)
lemma [*simp*]: ‹*nat-of-uint64 uint64-max-uint64* = *uint64-max*›
by (*auto simp*: *nat-of-uint64-mult-le nat-of-uint64-shiftl uint64-max-uint64-def uint64-max-def*)[]
lemma ‹*2* ∗ (*uint64-max div 2*) = *uint64-max* − *1*›
by (*auto simp*: *nat-of-uint64-mult-le nat-of-uint64-shiftl uint64-max-uint64-def uint64-max-def*)[]

lemma *nat-of-uint64-0-iff*: ‹*nat-of-uint64 x2* = *0* ⟷ *x2* = *0*›
using *word-nat-of-uint64-Rep-inject* **by** *fastforce*

lemma *arl64-append-rule*[*sep-heap-rules*]:
assumes ‹*length l* < *uint64-max*›
shows < *is-array-list64 l a* >
arl64-append a x
<λ*a*. *is-array-list64* (*l*@[*x*]) *a* >$_t$
proof −
have [*simp*]: ‹⋀*x1 x2 y ys*.
x2 < *uint64-of-nat ys* ⟹
nat-of-uint64 x2 ≤ *ys* ⟹

ys ≤ *uint64-max* ⟹ *nat-of-uint64 x2* < *ys*›
by (*metis nat-of-uint64-less-iff nat-of-uint64-uint64-of-nat-id*)
have [*simp*]: ‹⋀*x2 ys. x2* < *uint64-of-nat* (*Suc* (*ys*)) ⟹
Suc (*ys*) ≤ *uint64-max* ⟹
nat-of-uint64 (*x2* + *1*) = *1* + *nat-of-uint64 x2*›
by (*smt ab-semigroup-add-class.add.commute le-neq-implies-less less-or-eq-imp-le less-trans-Suc linorder-neqE-nat nat-of-uint64-012*(*3*) *nat-of-uint64-add nat-of-uint64-less-iff nat-of-uint64-uint64-of-nat-id not-less-eq plus-1-eq-Suc*)
have [*dest*]: ‹⋀*x2a x2 ys. x2* < *uint64-of-nat* (*Suc* (*ys*)) ⟹
Suc (*ys*) ≤ *uint64-max* ⟹
nat-of-uint64 x2 = *Suc x2a* ⟹*Suc x2a* ≤ *ys*›
by (*metis less-Suc-eq-le nat-of-uint64-less-iff nat-of-uint64-uint64-of-nat-id*)
have [*simp*]: ‹⋀*ys. ys* ≤ *uint64-max* ⟹
uint64-of-nat ys ≤ *uint64-max-uint64* >> *Suc 0* ⟹
nat-of-uint64 (*2* ∗ *uint64-of-nat ys*) = *2* ∗ *ys*›
by (*subst* (*asm*) *nat-of-uint64-le-iff*[*symmetric*])
(*auto simp*: *nat-of-uint64-uint64-of-nat-id uint64-max-uint64-def uint64-max-def nat-of-uint64-shiftl nat-of-uint64-mult-le*)
have [*simp*]: ‹⋀*ys. ys* ≤ *uint64-max* ⟹
uint64-of-nat ys ≤ *uint64-max-uint64* >> *Suc 0* ⟷ *ys* ≤ *uint64-max div 2*›
by (*subst nat-of-uint64-le-iff*[*symmetric*])
(*auto simp*: *nat-of-uint64-uint64-of-nat-id uint64-max-uint64-def uint64-max-def nat-of-uint64-shiftl nat-of-uint64-mult-le*)
have [*simp*]: ‹⋀*ys. ys* ≤ *uint64-max* ⟹
uint64-of-nat ys < *uint64-max-uint64* >> *Suc 0* ⟷ *ys* < *uint64-max div 2*›
by (*subst nat-of-uint64-less-iff*[*symmetric*])
(*auto simp*: *nat-of-uint64-uint64-of-nat-id uint64-max-uint64-def uint64-max-def nat-of-uint64-shiftl nat-of-uint64-mult-le*)

show *?thesis*
using *assms*
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append nat-of-uint64-0-iff*
split: *if-split*
split: *prod.splits nat.split*)
apply (*subst Array-upd-u64-def*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)

apply (*subst Array-upd-u64-def*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*subst Array-upd-u64-def*)
apply (*rule frame-rule*)
apply (*rule upd-rule*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append nat-of-uint64-0-iff*
split: *if-splits*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl64-append-def is-array-list64-def take-update-last neq-Nil-conv nat-of-uint64-mult-le length-u64-code-def min-def nat-of-uint64-add nat-of-uint64-uint64-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-splits*
split: *prod.splits nat.split*)
done
qed

lemma *arl64-length-rule*[*sep-heap-rules*]:
<*is-array-list64 l a*>
arl64-length a
<$\lambda r.$ *is-array-list64 l a* $*$ $\uparrow$(*nat-of-uint64 r*=*length l*)>
by (*sep-auto simp*: *arl64-length-def is-array-list64-def*)

lemma *arl64-is-empty-rule*[*sep-heap-rules*]:
<*is-array-list64 l a*>
arl64-is-empty a
<$\lambda r.$ *is-array-list64 l a* $*$ $\uparrow$($r \longleftrightarrow (l=[])$)>
by (*sep-auto simp*: *arl64-is-empty-def nat-of-uint64-0-iff is-array-list64-def*)

lemma *arl64-last-rule*[*sep-heap-rules*]:
$l \neq [] \Longrightarrow$
<*is-array-list64 l a*>
arl64-last a
<$\lambda r.$ *is-array-list64 l a* $*$ $\uparrow$(r=*last l*)>
by (*sep-auto simp*: *arl64-last-def is-array-list64-def nth-u64-code-def Array.nth′-def last-take-nth-conv*

nat-of-integer-integer-of-nat nat-of-uint64-ge-minus nat-of-uint64-le-iff[*symmetric*]
simp flip: *nat-of-uint64-code*)

lemma *arl64-get-rule*[*sep-heap-rules*]:
i<*length l* ⟹ (*i'*, *i*) ∈ *uint64-nat-rel* ⟹
<*is-array-list64 l a*>
arl64-get a i'
<λ*r*. *is-array-list64 l a* ∗ ↑(*r*=*l*!*i*)>
by (*sep-auto simp*: *arl64-get-def nth-u64-code-def is-array-list64-def uint64-nat-rel-def Array.nth'-def br-def split*: *prod.split simp flip*: *nat-of-uint64-code*)

lemma *arl64-set-rule*[*sep-heap-rules*]:
i<*length l* ⟹ (*i'*, *i*) ∈ *uint64-nat-rel* ⟹
<*is-array-list64 l a*>
arl64-set a i' x
<*is-array-list64* (*l*[*i*:=*x*])>
by (*sep-auto simp*: *arl64-set-def is-array-list64-def heap-array-set-u64-def uint64-nat-rel-def heap-array-set'-u64-def br-def Array.upd'-def split*: *prod.split simp flip*: *nat-of-uint64-code*)

definition *arl64-assn A* ≡ *hr-comp is-array-list64* (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure arl64-assn A* **for** *A*]

lemma *arl64-assn-comp*: *is-pure A* ⟹ *hr-comp* (*arl64-assn A*) (⟨*B*⟩*list-rel*) = *arl64-assn* (*hr-comp A B*)
unfolding *arl64-assn-def*
by (*auto simp*: *hr-comp-the-pure hr-comp-assoc list-rel-compp*)

lemma *arl64-assn-comp'*: *hr-comp* (*arl64-assn id-assn*) (⟨*B*⟩*list-rel*) = *arl64-assn* (*pure B*)
by (*simp add*: *arl64-assn-comp*)

context
notes [*fcomp-norm-unfold*] = *arl64-assn-def*[*symmetric*] *arl64-assn-comp'*
notes [*intro*!] = *hfrefI hn-refineI*[*THEN hn-refine-preI*]
notes [*simp*] = *pure-def hn-ctxt-def invalid-assn-def*
begin

lemma *arl64-empty-hnr-aux*: (*uncurry0 arl64-empty*,*uncurry0* (*RETURN op-list-empty*)) ∈ $unit\text{-}assn^k \rightarrow_a$ *is-array-list64*
by *sep-auto*
sepref-decl-impl (*no-register*) *arl64-empty*: *arl64-empty-hnr-aux* .

lemma *arl64-empty-sz-hnr-aux*: (*uncurry0* (*arl64-empty-sz N*),*uncurry0* (*RETURN op-list-empty*)) ∈ $unit\text{-}assn^k \rightarrow_a$ *is-array-list64*
by *sep-auto*

sepref-decl-impl (*no-register*) *arl64-empty-sz*: *arl64-empty-sz-hnr-aux* .

definition *op-arl64-empty* ≡ *op-list-empty*
definition *op-arl64-empty-sz* (*N*::*nat*) ≡ *op-list-empty*

lemma *arl64-copy-hnr-aux*: (*arl64-copy*,*RETURN o op-list-copy*) ∈ $is\text{-}array\text{-}list64^k \rightarrow_a$ *is-array-list64*
by *sep-auto*
sepref-decl-impl *arl64-copy*: *arl64-copy-hnr-aux* .

lemma *arl64-append-hnr-aux*: (*uncurry arl64-append*,*uncurry* (*RETURN oo op-list-append*)) $\in$ [λ(*xs*, *x*). *length xs* < *uint64-max*]$_a$ (*is-array-list64*^{d} $*_a$ *id-assn*^{k}) $\rightarrow$ *is-array-list64*
by *sep-auto*
sepref-decl-impl *arl64-append*: *arl64-append-hnr-aux*
unfolding *fref-param1* **by** (*auto intro*!: *frefI nres-relI simp*: *list-rel-imp-same-length*)

lemma *arl64-length-hnr-aux*: (*arl64-length*,*RETURN o op-list-length*) $\in$ *is-array-list64*^{k} $\rightarrow_a$ *uint64-nat-assn*
by (*sep-auto simp*: *uint64-nat-rel-def br-def*)
sepref-decl-impl *arl64-length*: *arl64-length-hnr-aux* **.**

lemma *arl64-is-empty-hnr-aux*: (*arl64-is-empty*,*RETURN o op-list-is-empty*) $\in$ *is-array-list64*^{k} $\rightarrow_a$ *bool-assn*
by *sep-auto*
sepref-decl-impl *arl64-is-empty*: *arl64-is-empty-hnr-aux* **.**

lemma *arl64-last-hnr-aux*: (*arl64-last*,*RETURN o op-list-last*) $\in$ [*pre-list-last*]$_a$ *is-array-list64*^{k} $\rightarrow$ *id-assn*
by *sep-auto*
sepref-decl-impl *arl64-last*: *arl64-last-hnr-aux* **.**

lemma *arl64-get-hnr-aux*: (*uncurry arl64-get*,*uncurry* (*RETURN oo op-list-get*)) $\in$ [λ(*l*,*i*). *i*<*length l*]$_a$ (*is-array-list64*^{k} $*_a$ *uint64-nat-assn*^{k}) $\rightarrow$ *id-assn*
by *sep-auto*
sepref-decl-impl *arl64-get*: *arl64-get-hnr-aux* **.**

lemma *arl64-set-hnr-aux*: (*uncurry2 arl64-set*,*uncurry2* (*RETURN ooo op-list-set*)) $\in$ [λ((*l*,*i*),-). *i*<*length l*]$_a$ (*is-array-list64*^{d} $*_a$ *uint64-nat-assn*^{k} $*_a$ *id-assn*^{k}) $\rightarrow$ *is-array-list64*
by *sep-auto*
sepref-decl-impl *arl64-set*: *arl64-set-hnr-aux* **.**

sepref-definition *arl64-swap* **is** *uncurry2 mop-list-swap* :: ((*arl64-assn id-assn*)^d $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *arl64-assn id-assn*)
unfolding *gen-mop-list-swap*[*abs-def*]
by *sepref*
sepref-decl-impl (*ismop*) *arl64-swap*: *arl64-swap.refine* **.**
end

interpretation *arl64*: *list-custom-empty arl64-assn A arl64-empty op-arl64-empty*
apply *unfold-locales*
apply (*rule arl64-empty-hnr*)
by (*auto simp*: *op-arl64-empty-def*)

lemma [*def-pat-rules*]: *op-arl64-empty-sz*\$*N* $\equiv$ *UNPROTECT* (*op-arl64-empty-sz N*) **by** *simp*

interpretation *arl64-sz*: *list-custom-empty arl64-assn A arl64-empty-sz N PR-CONST* (*op-arl64-empty-sz N*)
apply *unfold-locales*
apply (*rule arl64-empty-sz-hnr*)
by (*auto simp*: *op-arl64-empty-sz-def*)

definition *arl64-to-arl-conv* **where**

‹*arl64-to-arl-conv S = S*›

definition *arl64-to-arl* :: ‹*'a array-list64 ⇒ 'a array-list*› **where**
‹*arl64-to-arl* = (λ(*xs*, *n*). (*xs*, *nat-of-uint64 n*))›

lemma *arl64-to-arl-hnr*[*sepref-fr-rules*]:
‹(*return o arl64-to-arl*, *RETURN o arl64-to-arl-conv*) ∈ (*arl64-assn R*)^d $\rightarrow_a$ *arl-assn R*›
by (*sepref-to-hoare*)
(*sep-auto simp*: *arl64-to-arl-def arl64-to-arl-conv-def arl-assn-def arl64-assn-def is-array-list64-def is-array-list-def hr-comp-def*)

definition *arl64-take* **where**
‹*arl64-take n* = (λ(*xs*, -). (*xs*, *n*))›

lemma *arl64-take*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo arl64-take*), *uncurry* (*RETURN oo take*)) ∈
[λ(*n*, *xs*). *n* ≤ *length xs*]$_a$ *uint64-nat-assn*^{k} $*_a$ (*arl64-assn R*)^d → *arl64-assn R*›
by (*sepref-to-hoare*)
(*sep-auto simp*: *arl64-assn-def arl64-take-def is-array-list64-def hr-comp-def uint64-nat-rel-def br-def list-rel-def list-all2-conv-all-nth*)
definition *arl64-of-arl* :: ‹*'a list ⇒ 'a list*› **where**
‹*arl64-of-arl S = S*›

definition *arl64-of-arl-code* :: ‹*'a :: heap array-list ⇒ 'a array-list64 Heap*› **where**
‹*arl64-of-arl-code* = (λ(*a*, *n*). *do* {
m ← *Array.len a*;
if m > *uint64-max then do* {
a ← *array-shrink a uint64-max*;
return (*a*, (*uint64-of-nat n*))}
else return (*a*, (*uint64-of-nat n*))})›

lemma *arl64-of-arl*[*sepref-fr-rules*]:
‹(*arl64-of-arl-code*, *RETURN o arl64-of-arl*) ∈ [λ*n*. *length n* ≤ *uint64-max*]$_a$ (*arl-assn R*)^d → *arl64-assn R*›
proof −
have [*iff*]: ‹*take uint64-max l'* = [] ⟷ *l'* = []› ‹*0* < *uint64-max*› **for** *l'*
by (*auto simp*: *uint64-max-def*)
have *H*: ‹*x2* ≤ *length l'* ⟹
(*take x2 l'*, *x*) ∈ ⟨*the-pure R*⟩*list-rel* ⟹ *length x* = *x2*›
‹*x2* ≤ *length l'* ⟹
(*take x2 l'*, *x*) ∈ ⟨*the-pure R*⟩*list-rel* ⟹ *take* (*length x*) = *take x2*› **for** *x x2 l'*
subgoal *H* **by** (*auto dest*: *list-rel-imp-same-length*)
subgoal using *H* **by** *blast*
done
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *arl-assn-def arl64-assn-def is-array-list-def is-array-list64-def hr-comp-def arl64-of-arl-def arl64-of-arl-code-def nat-of-uint64-code*[*symmetric*] *nat-of-uint64-uint64-of-nat-id H min-def*
split: *prod.splits if-splits*)
qed

definition *arl-nat-of-uint64-conv* :: ‹*nat list ⇒ nat list*› **where**
‹*arl-nat-of-uint64-conv S = S*›

lemma *arl-nat-of-uint64-conv-alt-def*:

‹*arl-nat-of-uint64-conv* = *map nat-of-uint64-conv*›
  unfolding *nat-of-uint64-conv-def arl-nat-of-uint64-conv-def* **by** *auto*

sepref-definition *arl-nat-of-uint64-code*
  is *array-nat-of-uint64*
  :: ‹$(arl\text{-}assn\ uint64\text{-}nat\text{-}assn)^k \rightarrow_a arl\text{-}assn\ nat\text{-}assn$›
  unfolding *op-map-def array-nat-of-uint64-def arl-fold-custom-replicate*
  apply (*rewrite at* ‹*do* {*let* - = ¤; -}› *annotate-assn*[**where** *A*=‹*arl-assn nat-assn*›])
  by *sepref*

lemma *arl-nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
  ‹(*arl-nat-of-uint64-code*, (*RETURN* ∘ *arl-nat-of-uint64-conv*))
    ∈ $(arl\text{-}assn\ uint64\text{-}nat\text{-}assn)^k \rightarrow_a arl\text{-}assn\ nat\text{-}assn$›
  using *arl-nat-of-uint64-code.refine*[*unfolded array-nat-of-uint64-def*,
    FCOMP op-map-map-rel[*unfolded convert-fref*]] **unfolding** *arl-nat-of-uint64-conv-alt-def*
  by *simp*

end
theory *Array-Array-List64*
  imports *Array-Array-List IICF-Array-List64*
begin

0.1.8 Array of Array Lists of maximum length *uint64-max*

definition *length-aa64* :: ‹('*a*::*heap array-list64*) *array* ⇒ *uint64* ⇒ *uint64 Heap*› **where**
  ‹*length-aa64 xs i* = *do* {
     x ← *nth-u64-code xs i*;
    arl64-length x}›

lemma *arrayO-assn-Array-nth*[*sep-heap-rules*]:
  ‹*b* < *length xs* ⟹
    <*arrayO-assn* (*arl64-assn R*) *xs a*> *Array.nth a b*
    <λ*p*. *arrayO-except-assn* (*arl64-assn R*) [*b*] *xs a* (λ*p*'. ↑(*p*=*p*'!*b*))∗
     arl64-assn R (*xs* ! *b*) (*p*)>›
  unfolding *length-aa64-def nth-u64-code-def Array.nth'-def*
  apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of b*])
  apply *simp*
  unfolding *arrayO-except-assn-def arl-assn-def hr-comp-def*
  apply (*sep-auto simp*: *arrayO-except-assn-def arl-length-def arl-assn-def arl64-assn-def*
      eq-commute[*of* ‹(-, -)›] *is-array-list64-def hr-comp-def length-ll-def array-assn-def*
      is-array-def uint64-nat-rel-def br-def
      dest: *list-all2-lengthD split*: *prod.splits*)
  done

lemma *arl64-length*[*sep-heap-rules*]:
  ‹<*arl64-assn R b a*> *arl64-length a*< λ*r*. *arl64-assn R b a* ∗ ↑(*nat-of-uint64 r* = *length b*)>›
  by (*sep-auto simp*: *arrayO-except-assn-def arl-length-def arl-assn-def arl64-assn-def*
    eq-commute[*of* ‹(-, -)›] *is-array-list64-def hr-comp-def length-ll-def array-assn-def*
    is-array-def uint64-nat-rel-def br-def arl64-length-def list-rel-imp-same-length[*symmetric*]
    dest: *list-all2-lengthD split*: *prod.splits*)

lemma *length-aa64-rule*[*sep-heap-rules*]:
   ‹*b* < *length xs* ⟹ (*b*', *b*) ∈ *uint64-nat-rel* ⟹ <*arrayO-assn* (*arl64-assn R*) *xs a*> *length-aa64 a b*'
   <λ*r*. *arrayO-assn* (*arl64-assn R*) *xs a* ∗ ↑ (*nat-of-uint64 r* = *length-ll xs b*)>$_t$›
  unfolding *length-aa64-def nth-u64-code-def Array.nth'-def*

apply (*sep-auto simp flip*: *nat-of-uint64-code simp*: *br-def uint64-nat-rel-def length-ll-def*)
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of b*])
apply (*simp add*: *nat-of-uint64-code br-def uint64-nat-rel-def*)
apply (*sep-auto simp*: *arrayO-except-assn-def*)
done

lemma *length-aa64-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-aa64*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈ $[\lambda(xs, i).\ i < length\ xs]_a$ (*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint64-nat-assn*^{k} → *uint64-nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

lemma *arl64-get-hnr*[*sep-heap-rules*]:
assumes ‹$(n', n) \in$ *uint64-nat-rel*› **and** ‹$n <$ *length a*› **and** ‹*CONSTRAINT is-pure R*›
shows ‹<*arl64-assn R a b*>
arl64-get b n'
<λ*r*. *arl64-assn R a b* ∗ *R* (*a* ! *n*) *r*>›
proof −
obtain *A'* **where**
A: ‹*pure A'* = *R*›
using *assms pure-the-pure* **by** *auto*
then have *A'*: ‹*the-pure R* = *A'*›
by *auto*
show *?thesis*
using *param-nth*[*of n a n* ‹*take* (*nat-of-uint64* (*snd b*)) -› ‹*the-pure R*›, *simplified*] *assms*
unfolding *arl64-get-def arl64-assn-def nth-u64-code-def Array.nth'-def*
by (*sep-auto simp flip*: *nat-of-uint64-code A simp*: *br-def uint64-nat-rel-def hr-comp-def is-array-list64-def list-rel-imp-same-length*[*symmetric*] *pure-app-eq dest*: *split*: *prod.splits*)
qed

definition *nth-aa64* **where**
‹*nth-aa64 xs i j* = *do* {
x ← *Array.nth xs i*;
y ← *arl64-get x j*;
return y}›

lemma *nth-aa64-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-aa64*, *uncurry2* (*RETURN* ∘∘∘ *nth-ll*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
(*arrayO-assn* (*arl64-assn R*))^k $*_a$ *nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
have *H*: ‹*list-all2* (λ*x x'*. (*x*, *x'*) ∈ *the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*))) *bc* (*a* ! *ba*) ⟹
b < *length* (*a* ! *ba*) ⟹
(*bc* ! *b*, *a* ! *ba* ! *b*) ∈ *R'*› **for** *bc a ba b*
by (*auto simp add*: *ent-refl-true list-all2-conv-all-nth is-pure-alt-def pure-app-eq*[*symmetric*])
show *?thesis*
using *p*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-aa64-def length-ll-def nth-ll-def*)
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of ba*])
apply *simp*
apply (*sep-auto simp*: *arrayO-except-assn-def arrayO-assn-def arl64-assn-def hr-comp-def list-rel-def*

list-all2-lengthD
star-aci(*3*) *R R′ pure-def H*)
done
qed

definition *append64-el-aa* :: (*′a*::{*default*,*heap*} *array-list64*) *array* ⇒
nat ⇒ *′a* ⇒ (*′a array-list64*) *array Heap***where**
append64-el-aa ≡ *λa i x*. *do* {
j ← *Array.nth a i*;
a′ ← *arl64-append j x*;
Array.upd i a′ a
}

declare *arrayO-nth-rule*[*sep-heap-rules*]

lemma *sep-auto-is-stupid*:
fixes *R* :: ‹*′a* ⇒ *′b*::{*heap*,*default*} ⇒ *assn*›
assumes *p*: ‹*is-pure R*› **and** ‹*length l′* < *uint64-max*›
shows
‹<∃$_A$*p*. *R1 p* ∗ *R2 p* ∗ *arl64-assn R l′ aa* ∗ *R x x′* ∗ *R4 p*>
arl64-append aa x′ <*λr*. (∃$_A$*p*. *arl64-assn R* (*l′* @ [*x*]) *r* ∗ *R1 p* ∗ *R2 p* ∗ *R x x′* ∗ *R4 p* ∗ *true*) >›
proof −
obtain *R′* **where** *R*: ‹*the-pure R* = *R′*› **and** *R′*: ‹*R* = *pure R′*›
using *p* **by** *fastforce*
have *bbi*: ‹(*x′*, *x*) ∈ *the-pure R*› **if**
‹(*aa*, *bb*) ⊨ *is-array-list64* (*ba* @ [*x′*]) (*a*, *baa*) ∗ *R1 p* ∗ *R2 p* ∗ *pure R′ x x′* ∗ *R4 p* ∗ *true*›
for *aa bb a ba baa p*
using *that* **by** (*auto simp*: *mod-star-conv R R′*)
show *?thesis*
using *assms*(*2*)
unfolding *arl-assn-def hr-comp-def*
by (*sep-auto simp*: *list-rel-def R R′ arl64-assn-def hr-comp-def list-all2-lengthD*
intro!: *list-all2-appendI dest*!: *bbi*)
qed

lemma *append-aa64-hnr*[*sepref-fr-rules*]:
fixes *R* :: ‹*′a* ⇒ *′b* :: {*heap*, *default*} ⇒ *assn*›
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 append64-el-aa*, *uncurry2* (*RETURN* ∘∘∘ *append-ll*)) ∈
[*λ*((*l*,*i*),*x*). *i* < *length l* ∧ *length* (*l* ! *i*) < *uint64-max*]$_a$ (*arrayO-assn* (*arl64-assn R*))^d ∗$_a$ *nat-assn*^{k}
∗$_a$ *R*^{k} → (*arrayO-assn* (*arl64-assn R*))›
proof −
obtain *R′* **where** *R*: ‹*the-pure R* = *R′*› **and** *R′*: ‹*R* = *pure R′*›
using *p* **by** *fastforce*
have [*simp*]: ‹(∃$_A$*x*. *arrayO-assn* (*arl64-assn R*) *a ai* ∗ *R x r* ∗ *true* ∗ ↑ (*x* = *a* ! *ba* ! *b*)) =
(*arrayO-assn* (*arl64-assn R*) *a ai* ∗ *R* (*a* ! *ba* ! *b*) *r* ∗ *true*)› **for** *a ai ba b r*
by (*auto simp*: *ex-assn-def*)
show *?thesis* — TODO tune proof
apply *sepref-to-hoare*
apply (*sep-auto simp*: *append64-el-aa-def*)
apply (*simp add*: *arrayO-except-assn-def*)
apply (*rule sep-auto-is-stupid*[*OF p*])
apply *simp*
apply (*sep-auto simp*: *array-assn-def is-array-def append-ll-def*)

apply (*simp add*: *arrayO-except-assn-array0*[*symmetric*] *arrayO-except-assn-def*)
apply (*subst-tac* (*2*) *i* = *ba* **in** *heap-list-all-nth-remove1*)
apply (*solves* ‹*simp*›)
apply (*simp add*: *array-assn-def is-array-def*)
apply (*rule-tac x*=‹*p*[*ba* := (*ab*, *bc*)]› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)+
apply *sep-auto*
done
qed

definition *update-aa64* :: (*'a*::{*heap*} *array-list64*) *array* ⇒ *nat* ⇒ *uint64* ⇒ *'a* ⇒ (*'a array-list64*) *array Heap* **where**
‹*update-aa64 a i j y* = *do* {
x ← *Array.nth a i*;
a' ← *arl64-set x j y*;
Array.upd i a' a
}› — is the Array.upd really needed?

declare *nth-rule*[*sep-heap-rules del*]
declare *arrayO-nth-rule*[*sep-heap-rules*]

lemma *arrayO-except-assn-arl-set*[*sep-heap-rules*]:
fixes *R* :: ‹*'a* ⇒ *'b* :: {*heap*}⇒ *assn*›
assumes *p*: ‹*is-pure R*› **and** ‹*bb* < *length a*› **and**
‹*ba* < *length-ll a bb*› **and** ‹(*ba'* , *ba*) ∈ *uint64-nat-rel*›
shows ‹
<*arrayO-except-assn* (*arl64-assn R*) [*bb*] *a ai*
(λ*p'*. ↑ ((*aa*, *bc*) = *p'* ! *bb*)) ∗
arl64-assn R (*a* ! *bb*) (*aa*, *bc*) ∗
R b bi>
arl64-set (*aa*, *bc*) *ba' bi*
<λ(*aa*, *bc*). *arrayO-except-assn* (*arl64-assn R*) [*bb*] *a ai*
(λ*r'*. *arl64-assn R* ((*a* ! *bb*)[*ba* := *b*]) (*aa*, *bc*)) ∗ *R b bi* ∗ *true*>›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
show *?thesis*
using *assms*
apply (*sep-auto simp*: *arrayO-except-assn-def arl64-assn-def hr-comp-def list-rel-imp-same-length list-rel-update length-ll-def*)
done
qed

lemma *Array-upd-arrayO-except-assn*[*sep-heap-rules*]:
assumes
‹*bb* < *length a*› **and**
‹*ba* < *length-ll a bb*› **and** ‹(*ba'*, *ba*) ∈ *uint64-nat-rel*›
shows ‹<*arrayO-except-assn* (*arl64-assn R*) [*bb*] *a ai*
(λ*r'*. *arl64-assn R xu* (*aa*, *bc*)) ∗
R b bi ∗
true>
Array.upd bb (*aa*, *bc*) *ai*
<λ*r*. $\exists_A$*x*. *R b bi* ∗ *arrayO-assn* (*arl64-assn R*) *x r* ∗ *true* ∗
↑ (*x* = *a*[*bb* := *xu*])>›
proof −

have *H*[*simp*, *intro*]: ‹*ba* ≤ *length l′*›
  if
    ‹*ba* ≤ *length* (*a* ! *bb*)› **and**
    aa: ‹(*take n′ l′*, *a* ! *bb*) ∈ ⟨*the-pure R*⟩*list-rel*›
  for *l′* :: ‹*′b list*› **and** *n′*
proof −
  show *?thesis*
    using *list-rel-imp-same-length*[*OF aa*] *that assms*(*3*)
    by (*auto simp*: *uint64-nat-rel-def br-def list-rel-imp-same-length*[*symmetric*])
qed
have [*simp*]: ‹(*take ba l′*, *take ba* (*a* ! *bb*)) ∈ ⟨*the-pure R*⟩*list-rel*›
  if
    ‹*ba* ≤ *length* (*a* ! *bb*)› **and**
    ‹*n′* ≤ *length l′*› **and**
    take: ‹(*take n′ l′*, *a* ! *bb*) ∈ ⟨*the-pure R*⟩*list-rel*›
  for *l′* :: ‹*′b list*› **and** *n′*
proof −
  have [*simp*]: ‹*n′* = *length* (*a* ! *bb*)›
    using *list-rel-imp-same-length*[*OF take*] *that* **by** *auto*
  have *1*: ‹*take ba l′* = *take ba* (*take n′ l′*)›
    using *that* **by** (*auto simp*: *min-def*)
  show *?thesis*
    using *take*
    unfolding *1*
    by (*rule list-rel-take*)
qed

show *?thesis*
  using *assms*
  unfolding *arrayO-except-assn-def*
  apply (*subst* (*2*) *arl64-assn-def*)
  apply (*subst is-array-list64-def*[*abs-def*])
  apply (*subst hr-comp-def*[*abs-def*])
  apply (*subst array-assn-def*)
  apply (*subst is-array-def*[*abs-def*])
  apply (*subst hr-comp-def*[*abs-def*])
  apply *sep-auto*
  apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of bb*])
  apply (*solves simp*)
  unfolding *arrayO-except-assn-def array-assn-def is-array-def*
  apply (*subst* (*3*) *arl64-assn-def*)
  apply (*subst is-array-list64-def*[*abs-def*])
  apply (*subst* (*2*) *hr-comp-def*[*abs-def*])
  apply (*subst ex-assn-move-out*)+
  apply (*rule-tac x*=‹*p*[*bb* := (*aa*, *bc*)]› **in** *ent-ex-postI*)
  apply (*rule-tac x*=‹*take* (*nat-of-uint64 bc*) *l′*› **in** *ent-ex-postI*)
  apply (*sep-auto simp*: *uint64-nat-rel-def br-def list-rel-imp-same-length intro*!: *split*: *prod.splits*)
  apply (*subst* (*2*) *heap-list-all-nth-cong*[*of - - a - p*])
  apply *auto*
  apply *sep-auto*
  done
qed

lemma *update-aa64-rule*[*sep-heap-rules*]:
  assumes *p*: ‹*is-pure R*› **and** ‹*bb* < *length a*› **and** ‹*ba* < *length-ll a bb*› ‹(*ba′*, *ba*) ∈ *uint64-nat-rel*›
  shows ‹<*R b bi* ∗ *arrayO-assn* (*arl64-assn R*) *a ai*> *update-aa64 ai bb ba′ bi*

<λr. *R b bi* ∗ ($\exists_A x$. *arrayO-assn* (*arl64-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
using *assms*
by (*sep-auto simp add*: *update-aa64-def update-ll-def p*)

lemma *update-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa64*, *uncurry3* (*RETURN oooo update-ll*)) ∈
[λ(((*l*,*i*), *j*), *x*). *i* < *length l* ∧ *j* < *length-ll l i*]$_a$ (*arrayO-assn* (*arl64-assn R*))^d $*_a$ *nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$ R^k → (*arrayO-assn* (*arl64-assn R*))›
by *sepref-to-hoare* (*sep-auto simp*: *assms*)

definition *last-aa64* :: (′*a*::*heap array-list64*) *array* ⇒ *uint64* ⇒ ′*a Heap* **where**
‹*last-aa64 xs i* = *do* {
x ← *nth-u64-code xs i*;
arl64-last x
}›

lemma *arl64-last-rule*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› ‹*ai* ≠ []›
shows ‹<*arl64-assn R ai a*> *arl64-last a*
<λr. *arl64-assn R ai a* ∗ *R* (*last ai*) *r*>$_t$›
proof −
obtain *R*′ **where** *R*: ‹*the-pure R* = *R*′› **and** *R*′: ‹*R* = *pure R*′›
using *p* **by** *fastforce*
have [*simp*]: ‹⋀*aa n l*′.
(*take* (*nat-of-uint64 n*) *l*′, *ai*) ∈ ⟨*the-pure R*⟩*list-rel* ⟹
l′ ≠ [] ⟹ *nat-of-uint64 n* > *0*›
using *assms* **by** (*cases ai*; *auto simp*: *min-def split*: *if-splits dest*!: *list-rel-imp-same-length*[*symmetric*]
simp flip: *nat-of-uint64-le-iff simp*: *nat-of-uint64-ge-minus*; *fail*)+
have [*simp*]: ‹⋀*aa n l*′.
(*take* (*nat-of-uint64 n*) *l*′, *ai*) ∈ ⟨*the-pure R*⟩*list-rel* ⟹
l′ ≠ [] ⟹ *nat-of-uint64* (*n* − *1*) = *nat-of-uint64 n* − *1*›
using *assms* **by** (*cases ai*; *auto simp*: *min-def split*: *if-splits dest*!: *list-rel-imp-same-length*[*symmetric*]
simp flip: *nat-of-uint64-le-iff simp*: *nat-of-uint64-ge-minus*; *fail*)+
have [*simp*]: ‹⋀*aa n l*′.
(*take* (*nat-of-uint64 n*) *l*′, *ai*) ∈ ⟨*the-pure R*⟩*list-rel* ⟹
nat-of-uint64 n ≤ *length l*′ ⟹
l′ ≠ [] ⟹ *length l*′ ≤ *uint64-max* ⟹ *nat-of-uint64 n* − *Suc 0* < *length l*′›
using *assms* **by** (*cases ai*; *auto simp*: *min-def split*: *if-splits dest*!: *list-rel-imp-same-length*[*symmetric*]
simp flip: *nat-of-uint64-le-iff*)+
have [*intro*!]: ‹(*take* (*nat-of-uint64 n*) *l*′, *ai*) ∈ ⟨*R*′⟩*list-rel* ⟹
a = (*aa*, *n*) ⟹
nat-of-uint64 n ≤ *length l*′ ⟹
l′ ≠ [] ⟹
length l′ ≤ *uint64-max* ⟹
(*aaa*, *b*) ⊨ *aa* $\mapsto_a$ *l*′ ⟹
(*l*′ ! (*nat-of-uint64 n* − *Suc 0*), *ai* ! (*length ai* − *Suc 0*)) ∈ *R*′› **for** *aa n l*′ *aaa b*
using *assms*
nat-of-uint64-ge-minus[*of 1 n*] *param-last*[*OF assms*(*2*), *of* ‹*take* (*nat-of-uint64 n*) *l*′› *R*′]
by (*auto simp*: *min-def R*′ *last-conv-nth split*: *if-splits*
simp flip: *nat-of-uint64-le-iff*)
show *?thesis*
using *assms* **supply** *nth-rule*[*sep-heap-rules*] **apply** −
by (*sep-auto simp add*: *update-aa64-def update-ll-def p arl64-last-def arl64-assn-def R*′
pure-app-eq last-take-nth-conv last-conv-nth
nth-u64-code-def Array.nth′*-def hr-comp-def is-array-list64-def nat-of-uint64-ge-minus*

simp flip: *nat-of-uint64-code*
dest: *list-rel-imp-same-length*[*symmetric*])
qed

lemma *last-aa64-rule*[*sep-heap-rules*]:
assumes
p: ‹*is-pure R*› **and**
‹*b* < *length a*› **and**
‹*a* ! *b* ≠ []› **and** ‹(*b'*, *b*) ∈ *uint64-nat-rel*›
shows ‹
<*arrayO-assn* (*arl64-assn R*) *a ai*>
last-aa64 ai b'
<λ*r*. *arrayO-assn* (*arl64-assn R*) *a ai* ∗ (∃$_A$*x*. *R x r* ∗ ↑ (*x* = *last-ll a b*))>$_t$›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
have ‹⋀*b*.
b < *length a* ⟹ (*b'*, *b*) ∈ *uint64-nat-rel* ⟹
a ! *b* ≠ [] ⟹
<*arrayO-assn* (*arl64-assn R*) *a ai*>
last-aa64 ai b'
<λ*r*. *arrayO-assn* (*arl64-assn R*) *a ai* ∗ (∃$_A$*x*. *R x r* ∗ ↑ (*x* = *last-ll a b*))>$_t$›
apply (*sep-auto simp add*: *last-aa64-def last-ll-def assms nth-u64-code-def Array.nth'-def*
uint64-nat-rel-def br-def
simp flip: *nat-of-uint64-code*)

apply (*sep-auto simp add*: *last-aa64-def arrayO-except-assn-def array-assn-def is-array-def*
hr-comp-def arl64-assn-def)
apply (*subst-tac i*= ‹*nat-of-uint64 b'*› **in** *arrayO-except-assn-array0-index*[*symmetric*])
apply (*solves* ‹*simp*›)
apply (*subst arrayO-except-assn-def*)
apply (*auto simp add*: *last-aa-def arrayO-except-assn-def array-assn-def is-array-def hr-comp-def*)

apply (*rule-tac x*=‹*p*› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)
apply (*solves* ‹*auto*›)

apply (*rule-tac x*=‹*ba*› **in** *ent-ex-postI*)
unfolding *R* **unfolding** *R'*
apply (*sep-auto simp*: *pure-def param-last*)
done
from *this*[*of b*] **show** *?thesis*
using *assms* **unfolding** *R'* **by** *blast*
qed

lemma *last-aa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows ‹(*uncurry last-aa64*, *uncurry* (*RETURN oo last-ll*)) ∈
$[\lambda(l,i).\ i < length\ l \wedge l\ !\ i \neq []]_a\ (arrayO\text{-}assn\ (arl64\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
show *?thesis*
using *assms* **by** *sepref-to-hoare sep-auto*
qed

definition *swap-aa64* :: ('*a*::*heap array-list64*) *array* ⇒ *nat* ⇒ *uint64* ⇒ *uint64* ⇒ ('*a array-list64*) *array Heap* **where**
‹*swap-aa64 xs k i j* = *do* {
xi ← *nth-aa64 xs k i*;
xj ← *nth-aa64 xs k j*;
xs ← *update-aa64 xs k i xj*;
xs ← *update-aa64 xs k j xi*;
return xs
}›

lemma *nth-aa64-heap*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› **and** ‹*b* < *length aa*› **and** ‹*ba* < *length-ll aa b*› **and** ‹(*ba*′, *ba*) ∈ *uint64-nat-rel*›
shows ‹
<*arrayO-assn* (*arl64-assn R*) *aa a*>
nth-aa64 a b ba′
<λ*r*. $\exists_A$*x*. *arrayO-assn* (*arl64-assn R*) *aa a* ∗
(*R x r* ∗
↑ (*x* = *nth-ll aa b ba*)) ∗
true>›
proof −
have ‹<*arrayO-assn* (*arl64-assn R*) *aa a*>
nth-aa64 a b ba′
<λ*r*. $\exists_A$*x*. *arrayO-assn* (*arl64-assn R*) *aa a* ∗
R x r ∗
true ∗
↑ (*x* = *nth-ll aa b ba*)>›
using *p assms nth-aa64-hnr*[*of R*] **unfolding** *hfref-def hn-refine-def nth-aa64-def pure-app-eq*
by *auto*
then show *?thesis*
unfolding *hoare-triple-def*
by (*auto simp*: *Let-def pure-def*)
qed

lemma *update-aa-rule-pure*:
assumes *p*: ‹*is-pure R*› **and** ‹*b* < *length aa*› **and** ‹*ba* < *length-ll aa b*› **and**
‹(*ba*′, *ba*) ∈ *uint64-nat-rel*›
shows ‹
<*arrayO-assn* (*arl64-assn R*) *aa a* ∗ *R be bb*>
update-aa64 a b ba′ *bb*
<λ*r*. $\exists_A$*x*. *invalid-assn* (*arrayO-assn* (*arl64-assn R*)) *aa a* ∗ *arrayO-assn* (*arl64-assn R*) *x r* ∗
true ∗
↑ (*x* = *update-ll aa b ba be*)>›
proof −
obtain *R*′ **where** *R*′: ‹*R*′ = *the-pure R*› **and** *RR*′: ‹*R* = *pure R*′›
using *p* **by** *fastforce*
have *bb*: ‹*pure R*′ *be bb* = ↑((*bb*, *be*) ∈ *R*′)›
by (*auto simp*: *pure-def*)
have ‹<*arrayO-assn* (*arl64-assn R*) *aa a* ∗ *R be bb*>
update-aa64 a b ba′ *bb*
<λ*r*. $\exists_A$*x*. *invalid-assn* (*arrayO-assn* (*arl64-assn R*)) *aa a* ∗ *nat-assn b b* ∗ *nat-assn ba ba* ∗
R be bb ∗
arrayO-assn (*arl64-assn R*) *x r* ∗
true ∗

↑ (*x* = *update-ll aa b ba be*)>›
using *p assms update-aa-hnr*[*of R*] **unfolding** *hfref-def hn-refine-def pure-app-eq*
by *auto*
then show *?thesis*
unfolding *R′*[*symmetric*] **unfolding** *hoare-triple-def RR′ bb*
by (*auto simp*: *Let-def pure-def*)
qed

lemma *arl64-set-rule-arl64-assn*:
i<*length l* ⟹ (*i′*, *i*) ∈ *uint64-nat-rel* ⟹ (*x′*, *x*) ∈ *the-pure R* ⟹
<*arl64-assn R l a*>
arl64-set a i′ x′
<*arl64-assn R* (*l*[*i*:=*x*])>
supply *arl64-set-rule*[**where** *i*=*i*, *sep-heap-rules*]
by (*sep-auto simp*: *arl64-assn-def hr-comp-def list-rel-imp-same-length*
split: *prod.split simp flip*: *nat-of-uint64-code intro*!: *list-rel-update′*)

lemma *swap-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 swap-aa64*, *uncurry3* (*RETURN oooo swap-ll*)) ∈
$[\lambda(((xs, k), i), j).\ k < length\ xs \wedge i < length\text{-}ll\ xs\ k \wedge j < length\text{-}ll\ xs\ k]_a$
$(arrayO\text{-}assn\ (arl64\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow (arrayO\text{-}assn$
$(arl64\text{-}assn\ R))$›
proof −
note *update-aa-rule-pure*[*sep-heap-rules*]
obtain *R′* **where** *R′*: ‹*R′* = *the-pure R*› **and** *RR′*: ‹*R* = *pure R′*›
using *assms* **by** *fastforce*
have [*simp*]: ‹*the-pure* (λ*a b*. ↑ ((*b*, *a*) ∈ *R′*)) = *R′*›
unfolding *pure-def*[*symmetric*] **by** *auto*
show *?thesis*
using *assms* **unfolding** *R′*[*symmetric*] **unfolding** *swap-aa64-def*
apply *sepref-to-hoare*
supply *nth-aa64-heap*[*sep-heap-rules del*]
apply (*sep-auto simp*: *swap-ll-def arrayO-except-assn-def*
length-ll-update-ll uint64-nat-rel-def br-def)
supply *nth-aa64-heap*[*sep-heap-rules*]
apply (*sep-auto simp*: *swap-ll-def arrayO-except-assn-def*
length-ll-update-ll uint64-nat-rel-def br-def)
supply *nth-aa64-heap*[*sep-heap-rules del*]
apply (*sep-auto simp*: *swap-ll-def arrayO-except-assn-def*
length-ll-update-ll uint64-nat-rel-def br-def)
apply (*rule frame-rule*)
apply (*rule frame-rule*)
apply (*rule-tac ba*= ‹*nat-of-uint64 bi*› **in** *nth-aa64-heap*[*of*])
apply (*auto simp*: *swap-ll-def arrayO-except-assn-def*
length-ll-update-ll uint64-nat-rel-def br-def)
supply *update-aa-rule-pure*[*sep-heap-rules del*] *update-aa64-rule*[*sep-heap-rules del*]
apply (*sep-auto simp*: *uint64-nat-rel-def br-def*)
apply (*rule frame-rule*, *rule frame-rule*)
apply (*rule update-aa-rule-pure*)
apply (*auto simp*: *swap-ll-def arrayO-except-assn-def*
length-ll-update-ll uint64-nat-rel-def br-def)
apply *sep-auto*
apply (*rule cons-post-rule*)
apply (*subst assn-times-assoc*)
apply (*rule frame-rule*)

apply (*rule frame-rule-left*)
apply (*subst assn-times-comm*)
apply (*rule-tac R=R* **and** *ba=* ‹*nat-of-uint64 bi*› **in** *update-aa64-rule*)
apply (*auto simp*: *length-ll-def update-ll-def uint64-nat-rel-def br-def*)[*4*]
apply (*sep-auto simp*: *uint64-nat-rel-def br-def length-ll-def update-ll-def nth-ll-def swap-def*)
done
qed

It is not possible to do a direct initialisation: there is no element that can be put everywhere.

definition *arrayO-ara-empty-sz* **where**
‹*arrayO-ara-empty-sz n* =
(*let xs* = *fold* (λ- *xs*. [] # *xs*) [*0*..<*n*] [] *in*
op-list-copy xs)
›

lemma *of-list-op-list-copy-arrayO*[*sepref-fr-rules*]:
‹(*Array.of-list*, *RETURN* ∘ *op-list-copy*) ∈ (*list-assn* (*arl64-assn R*))^d $\rightarrow_a$ *arrayO-assn* (*arl64-assn R*)›
apply *sepref-to-hoare*
apply (*sep-auto simp*: *arrayO-assn-def array-assn-def*)
apply (*rule-tac ?psi*=‹*xa* $\mapsto_a$ *xi* ∗ *list-assn* (*arl64-assn R*) *x xi* $\Longrightarrow_A$
is-array xi xa ∗ *heap-list-all* (*arl64-assn R*) *x xi* ∗ *true*› **in** *asm-rl*)
by (*sep-auto simp*: *heap-list-all-list-assn is-array-def*)

sepref-definition
arrayO-ara-empty-sz-code
is *RETURN o arrayO-ara-empty-sz*
:: ‹*nat-assn*^{k} $\rightarrow_a$ *arrayO-assn* (*arl64-assn* (*R*::′*a* ⇒ ′*b*::{*heap*, *default*} ⇒ *assn*))›
unfolding *arrayO-ara-empty-sz-def op-list-empty-def*[*symmetric*]
apply (*rewrite at* ‹(#) ¤› *op-arl64-empty-def*[*symmetric*])
apply (*rewrite at* ‹*fold* - - ¤› *op-HOL-list-empty-def*[*symmetric*])
supply [[*goals-limit* = *1*]]
by *sepref*

definition *init-lrl64* :: ‹*nat* ⇒ -› **where**
[*simp*]: ‹*init-lrl64* = *init-lrl*›

lemma *arrayO-ara-empty-sz-init-lrl*: ‹*arrayO-ara-empty-sz n* = *init-lrl64 n*›
by (*induction n*) (*auto simp*: *arrayO-ara-empty-sz-def init-lrl-def*)

lemma *arrayO-raa-empty-sz-init-lrl*[*sepref-fr-rules*]:
‹(*arrayO-ara-empty-sz-code*, *RETURN o init-lrl64*) ∈
nat-assn^k $\rightarrow_a$ *arrayO-assn* (*arl64-assn R*)›
using *arrayO-ara-empty-sz-code.refine* **unfolding** *arrayO-ara-empty-sz-init-lrl* **.**

definition (**in** −) *shorten-take-aa64* **where**
‹*shorten-take-aa64 L j W* = *do* {
(*a*, *n*) ← *Array.nth W L*;
Array.upd L (*a*, *j*) *W*
}›

lemma *Array-upd-arrayO-except-assn2*[*sep-heap-rules*]:
assumes
‹*ba* ≤ *length* (*b* ! *a*)› **and**
‹*a* < *length b*› **and** ‹(*ba*′, *ba*) ∈ *uint64-nat-rel*›

shows ‹<*arrayO-except-assn* (*arl64-assn R*) [*a*] *b bi*
(*λr'*. ↑ ((*aaa*, *n*) = *r'* ! *a*)) ∗ *arl64-assn R* (*b* ! *a*) (*aaa*, *n*)>
Array.upd a (*aaa*, *ba'*) *bi*
<*λr*. ∃$_A$*x*. *arrayO-assn* (*arl64-assn R*) *x r* ∗ *true* ∗
↑ (*x* = *b*[*a* := *take ba* (*b* ! *a*)])>›
using *Array-upd-arrayO-except-assn*
proof −
have [*simp*]: ‹*nat-of-uint64 ba'* ≤ *length l'*›
if
‹*ba* ≤ *length* (*b* ! *a*)› **and**
aa: ‹(*take n' l'*, *b* ! *a*) ∈ ⟨*the-pure R*⟩*list-rel*›
for *l'* :: ‹*'b list*› **and** *n'*
proof −
show *?thesis*
using *list-rel-imp-same-length*[*OF aa*] *that assms*(*3*)
by (*auto simp*: *uint64-nat-rel-def br-def*)
qed
have [*simp*]: ‹(*take* (*nat-of-uint64 ba'*) *l'*, *take* (*nat-of-uint64 ba'*) (*b* ! *a*)) ∈ ⟨*the-pure R*⟩*list-rel*›
if
‹*ba* ≤ *length* (*b* ! *a*)› **and**
‹*n'* ≤ *length l'*› **and**
take: ‹(*take n' l'*, *b* ! *a*) ∈ ⟨*the-pure R*⟩*list-rel*›
for *l'* :: ‹*'b list*› **and** *n'*
proof −
have [*simp*]: ‹*n'* = *length* (*b* ! *a*)›
using *list-rel-imp-same-length*[*OF take*] *that* **by** *auto*
have *1*: ‹*take* (*nat-of-uint64 ba'*) *l'* = *take* (*nat-of-uint64 ba'*) (*take n' l'*)›
using *that assms*(*3*) **by** (*auto simp*: *min-def uint64-nat-rel-def br-def*)
show *?thesis*
using *take*
unfolding *1*
by (*rule list-rel-take*)
qed
show *?thesis*
using *assms*
unfolding *arrayO-except-assn-def*
apply (*subst* (*2*) *arl64-assn-def*)
apply (*subst is-array-list64-def*[*abs-def*])
apply (*subst hr-comp-def*[*abs-def*])
apply (*subst array-assn-def*)
apply (*subst is-array-def*[*abs-def*])
apply (*subst hr-comp-def*[*abs-def*])
apply *sep-auto*
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of a*])
apply (*solves simp*)
unfolding *arrayO-except-assn-def array-assn-def is-array-def*
apply (*subst* (*3*) *arl64-assn-def*)
apply (*subst is-array-list64-def*[*abs-def*])
apply (*subst* (*2*) *hr-comp-def*[*abs-def*])
apply (*subst ex-assn-move-out*)+
apply (*rule-tac x*=‹*p*[*a* := (*aaa*, *ba'*)]› **in** *ent-ex-postI*)
apply (*rule-tac x*=‹*take ba l'*› **in** *ent-ex-postI*)
apply (*sep-auto simp*: *uint64-nat-rel-def br-def list-rel-imp-same-length*
nat-of-uint64-le-uint64-max intro!: *split*: *prod.splits*)
apply (*subst* (*2*) *heap-list-all-nth-cong*[*of - - b - p*])
apply *auto*

apply *sep-auto*
done
qed

lemma *shorten-take-aa-hnr*[*sepref-fr-rules*]:
‹(*uncurry2 shorten-take-aa64*, *uncurry2* (*RETURN ooo shorten-take-ll*)) ∈
[λ((*L*, *j*), *W*). $j \leq$ *length* (*W* ! *L*) $\wedge$ *L* < *length W*]$_a$
nat-assn^k $*_a$ *uint64-nat-assn*^{k} $*_a$ (*arrayO-assn* (*arl64-assn R*))^d → *arrayO-assn* (*arl64-assn R*)›
unfolding *shorten-take-aa64-def shorten-take-ll-def*
by *sepref-to-hoare sep-auto*

definition *nth-aa64-u* **where**
‹*nth-aa64-u x L L′* = *nth-aa64 x* (*nat-of-uint32 L*) *L′*›

lemma *nth-aa-uint-hnr*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-aa64-u*, *uncurry2* (*RETURN ooo nth-rll*)) ∈
[λ((*x*, *L*), *L′*). *L* < *length x* $\wedge$ *L′* < *length* (*x* ! *L*)]$_a$
(*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
unfolding *nth-aa-u-def*
apply *auto*
by *sepref-to-hoare*
(*use assms* **in** ‹*sep-auto simp*: *uint32-nat-rel-def br-def length-ll-def nth-ll-def*
nth-rll-def nth-aa64-u-def›)

lemma *nth-aa64-u-code*[*code*]:
‹*nth-aa64-u x L L′* = *nth-u-code x L* ⤜ (λ*x*. *arl64-get x L′* ⤜ *return*)›
unfolding *nth-aa64-u-def nth-aa64-def arl-get-u-def*[*symmetric*] *Array.nth′-def*[*symmetric*]
nth-nat-of-uint32-nth′ nth-u-code-def[*symmetric*] **..**

definition *nth-aa64-i64-u64* **where**
‹*nth-aa64-i64-u64 xs x L* = *nth-aa64 xs* (*nat-of-uint64 x*) *L*›

lemma *nth-aa64-i64-u64-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 nth-aa64-i64-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
[λ((*l*,*i*),*j*). *i* < *length l* $\wedge$ *j* < *length-rll l i*]$_a$
(*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
unfolding *nth-aa64-i64-u64-def*
supply *nth-aa64-hnr*[*to-hnr*, *sep-heap-rules*]
using *assms*
by *sepref-to-hoare*
(*sep-auto simp*: *br-def nth-aa64-i64-u64-def uint64-nat-rel-def*
length-rll-def length-ll-def nth-rll-def nth-ll-def)

definition *nth-aa64-i32-u64* **where**
‹*nth-aa64-i32-u64 xs x L* = *nth-aa64 xs* (*nat-of-uint32 x*) *L*›

lemma *nth-aa64-i32-u64-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 nth-aa64-i32-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-rll*)) ∈
[λ((*l*,*i*),*j*). *i* < *length l* $\wedge$ *j* < *length-rll l i*]$_a$

$(arrayO\text{-}assn\ (arl64\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
unfolding *nth-aa64-i32-u64-def*
supply *nth-aa64-hnr*[*to-hnr*, *sep-heap-rules*]
using *assms*
by *sepref-to-hoare*
(*sep-auto simp*: *uint32-nat-rel-def br-def uint64-nat-rel-def*
length-rll-def length-ll-def nth-rll-def nth-ll-def)

definition *append64-el-aa32* :: (*'a*::{*default*,*heap*} *array-list64*) *array* ⇒
uint32 ⇒ *'a* ⇒ (*'a array-list64*) *array Heap***where**
append64-el-aa32 ≡ *λa i x. do* {
j ← *nth-u-code a i*;
a' ← *arl64-append j x*;
heap-array-set-u a i a'
}

lemma *append64-aa32-hnr*[*sepref-fr-rules*]:
fixes *R* :: ‹*'a* ⇒ *'b* :: {*heap*, *default*} ⇒ *assn*›
assumes *p*: ‹*is-pure R*›
shows
‹(*uncurry2 append64-el-aa32*, *uncurry2* (*RETURN* ∘∘∘ *append-ll*)) ∈
$[\lambda((l,i),x).\ i < length\ l \wedge length\ (l\ !\ i) < uint64\text{-}max]_a\ (arrayO\text{-}assn\ (arl64\text{-}assn\ R))^d *_a uint32\text{-}nat\text{-}assn^k *_a R^k \rightarrow (arrayO\text{-}assn\ (arl64\text{-}assn\ R))$›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
have [*simp*]: ‹($\exists_A x$. *arrayO-assn* (*arl64-assn R*) *a ai* ∗ *R x r* ∗ *true* ∗ ↑ (*x* = *a* ! *ba* ! *b*)) =
(*arrayO-assn* (*arl64-assn R*) *a ai* ∗ *R* (*a* ! *ba* ! *b*) *r* ∗ *true*)› **for** *a ai ba b r*
by (*auto simp*: *ex-assn-def*)
show *?thesis* — TODO tune proof
apply *sepref-to-hoare*
apply (*sep-auto simp*: *append64-el-aa32-def nth-u-code-def Array.nth'-def uint32-nat-rel-def br-def*
nat-of-uint32-code[*symmetric*] *heap-array-set'-u-def heap-array-set-u-def Array.upd'-def*)
apply (*simp add*: *arrayO-except-assn-def*)
apply (*rule sep-auto-is-stupid*[*OF p*])
apply *simp*
apply (*sep-auto simp*: *array-assn-def is-array-def append-ll-def*)
apply (*simp add*: *arrayO-except-assn-array0*[*symmetric*] *arrayO-except-assn-def*)
apply (*subst-tac* (*2*) *i* = ‹*nat-of-uint32 bia*› **in** *heap-list-all-nth-remove1*)
apply (*solves* ‹*simp*›)
apply (*simp add*: *array-assn-def is-array-def*)
apply (*rule-tac x*=‹*p*[*nat-of-uint32 bia* := (*ab*, *bb*)]› **in** *ent-ex-postI*)
apply (*subst-tac* (*2*)*xs'*=*a* **and** *ys'*=*p* **in** *heap-list-all-nth-cong*)
apply (*solves* ‹*auto*›)+
apply *sep-auto*
done
qed

definition *update-aa64-u32* :: (*'a*::{*heap*} *array-list64*) *array* ⇒ *uint32* ⇒ *uint64* ⇒ *'a* ⇒ (*'a array-list64*) *array Heap* **where**
‹*update-aa64-u32 a i j y* = *update-aa64 a* (*nat-of-uint32 i*) *j y*›

lemma *update-aa-u64-u32-code*[*code*]:
‹*update-aa64-u32 a i j y* = *do* {
x ← *nth-u-code a i*;

$a' \leftarrow$ *arl64-set x j y*;
Array-upd-u i a' a
}›
unfolding *update-aa64-u32-def update-aa64-def update-aa-def nth-nat-of-uint32-nth' nth-nat-of-uint32-nth' arl-get-u-def*[*symmetric*] *nth-u64-code-def Array.nth'-def comp-def Array-upd-u-def nth-u-code-def heap-array-set'-u-def*[*symmetric*] *Array-upd-u64-def nat-of-uint64-code*[*symmetric*]
by *auto*

lemma *update-aa64-u32-rule*[*sep-heap-rules*]:
assumes *p*: ‹*is-pure R*› **and** ‹$bb <$ *length a*› **and** ‹$ba <$ *length-ll a bb*› ‹$(ba', ba) \in$ *uint64-nat-rel*› ‹$(bb', bb) \in$ *uint32-nat-rel*›
shows ‹<*R b bi* $*$ *arrayO-assn* (*arl64-assn R*) *a ai*> *update-aa64-u32 ai bb' ba' bi*
<λr. *R b bi* $*$ $(\exists_A x.$ *arrayO-assn* (*arl64-assn R*) *x r* $* \uparrow (x =$ *update-ll a bb ba b*$))>_t$›
using *assms* **supply** *return-sp-rule*[*sep-heap-rules*] *upd-rule*[*sep-heap-rules del*]
apply (*sep-auto simp add*: *update-aa64-u32-def update-ll-def nth-u-code-def Array.nth'-def nat-of-uint32-code*[*symmetric*] *uint32-nat-rel-def br-def*)
done

lemma *update-aa64-u32-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa64-u32*, *uncurry3* (*RETURN oooo update-ll*)) $\in$
$[\lambda(((l,i), j), x).\ i <$ *length l* $\wedge\ j <$ *length-ll l i*$]_a$ (*arrayO-assn* (*arl64-assn R*))$^d *_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$ $R^k \rightarrow$ (*arrayO-assn* (*arl64-assn R*))›
by *sepref-to-hoare* (*sep-auto simp*: *assms*)

definition *nth-aa64-u64* **where**
‹*nth-aa64-u64 xs i j* = *do* {
$x \leftarrow$ *nth-u64-code xs i*;
$y \leftarrow$ *arl64-get x j*;
return y}›

lemma *nth-aa64-u64-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-aa64-u64*, *uncurry2* (*RETURN* ∘∘∘ *nth-ll*)) $\in$
$[\lambda((l,i),j).\ i <$ *length l* $\wedge\ j <$ *length-ll l i*$]_a$
(*arrayO-assn* (*arl64-assn R*))$^k *_a$ *uint64-nat-assn*$^k *_a$ *uint64-nat-assn*$^k \rightarrow R$›
proof −
obtain R' **where** *R*: ‹*the-pure R* = R'› **and** R': ‹R = *pure* R'›
using *p* **by** *fastforce*
have *H*: ‹*list-all2* $(\lambda x\ x'.\ (x, x') \in$ *the-pure* $(\lambda a\ c. \uparrow ((c, a) \in R')))$ *bc* (*a* ! *ba*) $\Longrightarrow$
$b <$ *length* (*a* ! *ba*) $\Longrightarrow$
(*bc* ! *b*, *a* ! *ba* ! *b*) $\in R'$› **for** *bc a ba b*
by (*auto simp add*: *ent-refl-true list-all2-conv-all-nth is-pure-alt-def pure-app-eq*[*symmetric*])
show *?thesis*
using *p*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-aa64-u64-def length-ll-def nth-ll-def nth-u64-def nth-u64-code-def Array.nth'-def*
nat-of-uint64-code[*symmetric*] *br-def uint64-nat-rel-def*)
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of* ‹*nat-of-uint64 bia*›])
apply *simp*
apply (*sep-auto simp*: *arrayO-except-assn-def arrayO-assn-def arl64-assn-def hr-comp-def list-rel-def list-all2-lengthD*
star-aci(*3*) *R R' pure-def H*)
done

qed

definition *arl64-get-nat* :: *'a::heap array-list64* ⇒ *nat* ⇒ *'a Heap* **where**
arl64-get-nat ≡ λ(*a*,*n*) *i*. *Array.nth a i*

lemma *arl-get-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-array-list64 l a*>
arl64-get-nat a i
<λ*r*. *is-array-list64 l a* ∗ ↑(*r*=*l*!*i*)>
supply *nth-rule*[*sep-heap-rules*]
by (*sep-auto simp*: *is-array-list64-def arl64-get-nat-def is-array-list-def split*: *prod.split*)

lemma *arl-get-rule-arl64*[*sep-heap-rules*]:
i<*length l* ⟹
<*arl64-assn T l a*>
arl64-get-nat a i
<λ*r*. *arl64-assn T l a* ∗ ↑((*r*, *l*!*i*) ∈ *the-pure T*)>
using *param-nth*[*of i l i* - ‹*the-pure T*›]
by (*sep-auto simp*: *arl64-assn-def hr-comp-def dest*: *list-rel-imp-same-length split*: *prod.split*)

definition *nth-aa64-nat* **where**
‹*nth-aa64-nat xs i j* = *do* {
x ← *Array.nth xs i*;
y ← *arl64-get-nat x j*;
return y}›

lemma *nth-aa64-nat-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*CONSTRAINT is-pure R*›
shows
‹(*uncurry2 nth-aa64-nat*, *uncurry2* (*RETURN* ∘∘∘ *nth-ll*)) ∈
[λ((*l*,*i*),*j*). *i* < *length l* ∧ *j* < *length-ll l i*]$_a$
(*arrayO-assn* (*arl64-assn R*))^k ∗$_a$ *nat-assn*^{k} ∗$_a$ *nat-assn*^{k} → *R*›
proof −
obtain *R'* **where** *R*: ‹*the-pure R* = *R'*› **and** *R'*: ‹*R* = *pure R'*›
using *p* **by** *fastforce*
have [*simp*]: ‹*the-pure* (λ*a c*. ↑ ((*c*, *a*) ∈ *R'*)) = *R'*›
unfolding *R*[*symmetric*] *pure-app-eq*[*symmetric*] **by** *auto*
show *?thesis*
using *p*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *nth-aa64-nat-def length-ll-def nth-ll-def*)
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of ba*])
apply *simp*
apply (*sep-auto simp*: *arrayO-except-assn-def arrayO-assn-def arl64-assn-def hr-comp-def list-rel-def star-aci*(*3*) *R R' pure-def*)
done
qed

definition *length-aa64-nat* :: ‹(*'a::heap array-list64*) *array* ⇒ *nat* ⇒ *nat Heap*› **where**
‹*length-aa64-nat xs i* = *do* {
x ← *Array.nth xs i*;
n ← *arl64-length x*;
return (*nat-of-uint64 n*)}›

lemma *length-aa64-nat-rule*[*sep-heap-rules*]:

‹*b* < *length xs* ⟹ <*arrayO-assn* (*arl64-assn R*) *xs a*> *length-aa64-nat a b*
<*λr. arrayO-assn* (*arl64-assn R*) *xs a* ∗ ↑ (*r* = *length-ll xs b*)>$_t$›
unfolding *length-aa64-nat-def nth-u64-code-def Array.nth′-def*
apply (*sep-auto simp flip*: *nat-of-uint64-code simp*: *br-def uint64-nat-rel-def length-ll-def*)
apply (*subst arrayO-except-assn-array0-index*[*symmetric*, *of b*])
apply (*simp add*: *nat-of-uint64-code br-def uint64-nat-rel-def*)
apply (*sep-auto simp*: *arrayO-except-assn-def*)
done

lemma *length-aa64-nat-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-aa64-nat*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈
[λ(*xs*, *i*). *i* < *length xs*]$_a$ (*arrayO-assn* (*arl64-assn R*))^k ∗$_a$ *nat-assn*^{k} → *nat-assn*›
by *sepref-to-hoare* (*sep-auto simp*: *uint64-nat-rel-def br-def*)

end
theory *IICF-Array-List32*
imports
Refine-Imperative-HOL.IICF-List
Separation-Logic-Imperative-HOL.Array-Blit
Array-UInt
WB-Word-Assn
begin

type-synonym ′*a array-list32* = ′*a Heap.array* × *uint32*

definition *is-array-list32 l* ≡ λ(*a*,*n*). ∃$_A$*l*′. *a* ↦$_a$ *l*′ ∗ ↑(*nat-of-uint32 n* ≤ *length l*′ ∧ *l* = *take* (*nat-of-uint32 n*) *l*′ ∧ *length l*′>*0* ∧ *nat-of-uint32 n* ≤ *uint32-max* ∧ *length l*′ ≤ *uint32-max*)

lemma *is-array-list32-prec*[*safe-constraint-rules*]: *precise is-array-list32*
unfolding *is-array-list32-def*[*abs-def*]
apply(*rule preciseI*)
apply(*simp split*: *prod.splits*)
using *preciseD snga-prec* **by** *fastforce*

definition *arl32-empty* ≡ *do* {
a ← *Array.new initial-capacity default*;
return (*a*,*0*)
}

definition *arl32-empty-sz init-cap* ≡ *do* {
a ← *Array.new* (*min uint32-max* (*max init-cap minimum-capacity*)) *default*;
return (*a*,*0*)
}

definition *uint32-max-uint32* :: *uint32* **where**
‹*uint32-max-uint32* = *2* ^*32* − *1*›

definition *arl32-append* ≡ λ(*a*,*n*) *x*. *do* {
len ← *length-u-code a*;

if n<*len then do* {
a ← *Array-upd-u n x a*;
return (*a*,*n*+*1*)
} *else do* {
let newcap = (*if len* < *uint32-max-uint32* >> *1 then 2* ∗ *len else uint32-max-uint32*);
a ← *array-grow a* (*nat-of-uint32 newcap*) *default*;

    a ← Array-upd-u n x a;
    return (a,n+1)
  }
}

definition arl32-copy ≡ λ(a,n). do {
  a ← array-copy a;
  return (a,n)
}

definition arl32-length :: 'a::heap array-list32 ⇒ uint32 Heap **where**
  arl32-length ≡ λ(a,n). return (n)

definition arl32-is-empty :: 'a::heap array-list32 ⇒ bool Heap **where**
  arl32-is-empty ≡ λ(a,n). return (n=0)

definition arl32-last :: 'a::heap array-list32 ⇒ 'a Heap **where**
  arl32-last ≡ λ(a,n). do {
    nth-u-code a (n − 1)
  }

definition arl32-butlast :: 'a::heap array-list32 ⇒ 'a array-list32 Heap **where**
  arl32-butlast ≡ λ(a,n). do {
    let n = n − 1;
    len ← length-u-code a;
    if (n*4 < len ∧ nat-of-uint32 n*2≥minimum-capacity) then do {
      a ← array-shrink a (nat-of-uint32 n*2);
      return (a,n)
    } else
      return (a,n)
  }

definition arl32-get :: 'a::heap array-list32 ⇒ uint32 ⇒ 'a Heap **where**
  arl32-get ≡ λ(a,n) i. nth-u-code a i

definition arl32-set :: 'a::heap array-list32 ⇒ uint32 ⇒ 'a ⇒ 'a array-list32 Heap **where**
  arl32-set ≡ λ(a,n) i x. do { a ← heap-array-set-u a i x; return (a,n)}

lemma arl32-empty-rule[sep-heap-rules]: < emp > arl32-empty <is-array-list32 []>
  by (sep-auto simp: arl32-empty-def is-array-list32-def initial-capacity-def uint32-max-def)

lemma arl32-empty-sz-rule[sep-heap-rules]: < emp > arl32-empty-sz N <is-array-list32 []>
  by (sep-auto simp: arl32-empty-sz-def is-array-list32-def minimum-capacity-def uint32-max-def)

lemma arl32-copy-rule[sep-heap-rules]: < is-array-list32 l a > arl32-copy a <λr. is-array-list32 l a * is-array-list32 l r>
  by (sep-auto simp: arl32-copy-def is-array-list32-def)

lemma nat-of-uint32-shiftl: ‹nat-of-uint32 (xs >> a) = nat-of-uint32 xs >> a›
  by transfer (auto simp: unat-shiftr nat-shifl-div)

lemma [simp]: ‹nat-of-uint32 uint32-max-uint32 = uint32-max›
  by (auto simp: nat-of-uint32-mult-le nat-of-uint32-shiftl uint32-max-uint32-def uint32-max-def)[]

lemma ‹2 * (uint32-max div 2) = uint32-max − 1›

```
  by (auto simp:  nat-of-uint32-mult-le nat-of-uint32-shiftl uint32-max-uint32-def uint32-max-def)[]

lemma arl32-append-rule[sep-heap-rules]:
  assumes ‹length l < uint32-max›
  shows < is-array-list32 l a >
      arl32-append a x
    <λa. is-array-list32 (l@[x]) a >t
proof −
  have [simp]: ‹⋀x1 x2 y ys.
       x2 < uint32-of-nat ys ⟹
       nat-of-uint32 x2 ≤ ys ⟹
       ys ≤ uint32-max ⟹ nat-of-uint32 x2 < ys›
    by (metis nat-of-uint32-less-iff nat-of-uint32-uint32-of-nat-id)
  have [simp]: ‹⋀x2 ys. x2 < uint32-of-nat (Suc (ys)) ⟹
       Suc (ys) ≤ uint32-max ⟹
       nat-of-uint32 (x2 + 1) = 1 + nat-of-uint32 x2›
     by (smt ab-semigroup-add-class.add.commute le-neq-implies-less less-or-eq-imp-le
         less-trans-Suc linorder-neqE-nat nat-of-uint32-012(3) nat-of-uint32-add
          nat-of-uint32-less-iff nat-of-uint32-uint32-of-nat-id not-less-eq plus-1-eq-Suc)
  have [dest]: ‹⋀x2a x2 ys. x2 < uint32-of-nat (Suc (ys)) ⟹
       Suc (ys) ≤ uint32-max ⟹
       nat-of-uint32 x2 = Suc x2a ⟹Suc x2a ≤ ys›
    by (metis less-Suc-eq-le nat-of-uint32-less-iff nat-of-uint32-uint32-of-nat-id)
  have [simp]: ‹⋀ys. ys ≤ uint32-max ⟹
       uint32-of-nat ys ≤ uint32-max-uint32 >> Suc 0 ⟹
       nat-of-uint32 (2 ∗ uint32-of-nat ys) = 2 ∗ ys›
   by (subst (asm) nat-of-uint32-le-iff[symmetric])
   (auto simp: nat-of-uint32-uint32-of-nat-id uint32-max-uint32-def uint32-max-def nat-of-uint32-shiftl
       nat-of-uint32-mult-le)
  have [simp]: ‹⋀ys. ys ≤ uint32-max ⟹
       uint32-of-nat ys ≤ uint32-max-uint32 >> Suc 0 ⟷ ys ≤ uint32-max div 2›
   by (subst nat-of-uint32-le-iff[symmetric])
   (auto simp: nat-of-uint32-uint32-of-nat-id uint32-max-uint32-def uint32-max-def nat-of-uint32-shiftl
       nat-of-uint32-mult-le)
  have [simp]: ‹⋀ys. ys ≤ uint32-max ⟹
       uint32-of-nat ys < uint32-max-uint32 >> Suc 0 ⟷ ys < uint32-max div 2›
   by (subst nat-of-uint32-less-iff[symmetric])
   (auto simp: nat-of-uint32-uint32-of-nat-id uint32-max-uint32-def uint32-max-def nat-of-uint32-shiftl
       nat-of-uint32-mult-le)

  show ?thesis
    using assms
    apply (sep-auto
      simp: arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le
        length-u-code-def  min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id
    take-Suc-conv-app-nth list-update-append nat-of-uint32-0-iff
      split: if-split
      split: prod.splits nat.split)
  apply (subst Array-upd-u-def)
apply (sep-auto
      simp: arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le
        length-u-code-def  min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id
        take-Suc-conv-app-nth list-update-append
      split: if-split
      split: prod.splits nat.split)
apply (sep-auto
```

simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*subst Array-upd-u-def*)
apply (*sep-auto*
simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-split*
split: *prod.splits nat.split*)
apply (*subst Array-upd-u-def*)
apply (*rule frame-rule*)
apply (*rule upd-rule*)
apply (*sep-auto*
simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append nat-of-uint32-0-iff*
split: *if-splits*
split: *prod.splits nat.split*)
apply (*sep-auto*
simp: *arl32-append-def is-array-list32-def take-update-last neq-Nil-conv nat-of-uint32-mult-le length-u-code-def min-def nat-of-uint32-add nat-of-uint32-uint32-of-nat-id take-Suc-conv-app-nth list-update-append*
split: *if-splits*
split: *prod.splits nat.split*)
done
qed

lemma *arl32-length-rule*[*sep-heap-rules*]:
<*is-array-list32 l a*>
arl32-length a
<*λr. is-array-list32 l a* ∗ ↑(*nat-of-uint32 r*=*length l*)>
by (*sep-auto simp*: *arl32-length-def is-array-list32-def*)

lemma *arl32-is-empty-rule*[*sep-heap-rules*]:

```
  <is-array-list32 l a>
    arl32-is-empty a
  <λr. is-array-list32 l a ∗ ↑(r⟷(l=[]))>
  by (sep-auto simp: arl32-is-empty-def nat-of-uint32-0-iff is-array-list32-def)

lemma nat-of-uint32-ge-minus:
  ‹ai ≥ bi ⟹
      nat-of-uint32 (ai − bi) = nat-of-uint32 ai − nat-of-uint32 bi›
  apply transfer
  unfolding unat-def
  by (subst uint-sub-lem[THEN iffD1])
    (auto simp: unat-def uint-nonnegative nat-diff-distrib word-le-def[symmetric] intro: leI)

lemma arl32-last-rule[sep-heap-rules]:
  l≠[] ⟹
  <is-array-list32 l a>
    arl32-last a
  <λr. is-array-list32 l a ∗ ↑(r=last l)>
  by (sep-auto simp: arl32-last-def is-array-list32-def nth-u-code-def Array.nth′-def last-take-nth-conv
    nat-of-integer-integer-of-nat nat-of-uint32-ge-minus  nat-of-uint32-le-iff[symmetric]
    simp flip: nat-of-uint32-code)


lemma arl32-get-rule[sep-heap-rules]:
  i<length l ⟹ (i′, i) ∈ uint32-nat-rel ⟹
  <is-array-list32 l a>
    arl32-get a i′
  <λr. is-array-list32 l a ∗ ↑(r=l!i)>
  by (sep-auto simp: arl32-get-def nth-u-code-def is-array-list32-def uint32-nat-rel-def
   Array.nth′-def br-def split: prod.split simp flip: nat-of-uint32-code)

lemma arl32-set-rule[sep-heap-rules]:
  i<length l ⟹ (i′, i) ∈ uint32-nat-rel ⟹
  <is-array-list32 l a>
    arl32-set a i′ x
  <is-array-list32 (l[i:=x])>
  by (sep-auto simp: arl32-set-def is-array-list32-def heap-array-set-u-def  uint32-nat-rel-def
   heap-array-set′-u-def br-def Array.upd′-def split: prod.split simp flip: nat-of-uint32-code)

definition arl32-assn A ≡ hr-comp is-array-list32 (⟨the-pure A⟩list-rel)
lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure arl32-assn A for A]


lemma arl32-assn-comp: is-pure A ⟹ hr-comp (arl32-assn A) (⟨B⟩list-rel) = arl32-assn (hr-comp A
B)
  unfolding arl32-assn-def
  by (auto simp: hr-comp-the-pure hr-comp-assoc list-rel-compp)

lemma arl32-assn-comp′: hr-comp (arl32-assn id-assn) (⟨B⟩list-rel) = arl32-assn (pure B)
  by (simp add: arl32-assn-comp)

context
  notes [fcomp-norm-unfold] = arl32-assn-def[symmetric] arl32-assn-comp′
  notes [intro!] = hfrefI hn-refineI[THEN hn-refine-preI]
  notes [simp] = pure-def hn-ctxt-def invalid-assn-def
begin
```

lemma *arl32-empty-hnr-aux*: (*uncurry0 arl32-empty*,*uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *is-array-list32*
by *sep-auto*
sepref-decl-impl (*no-register*) *arl32-empty*: *arl32-empty-hnr-aux* **.**

lemma *arl32-empty-sz-hnr-aux*: (*uncurry0* (*arl32-empty-sz N*),*uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *is-array-list32*
by *sep-auto*

sepref-decl-impl (*no-register*) *arl32-empty-sz*: *arl32-empty-sz-hnr-aux* **.**

definition *op-arl32-empty* ≡ *op-list-empty*
definition *op-arl32-empty-sz* (*N*::*nat*) ≡ *op-list-empty*

lemma *arl32-copy-hnr-aux*: (*arl32-copy*,*RETURN o op-list-copy*) ∈ *is-array-list32*^{k} $\rightarrow_a$ *is-array-list32*
by *sep-auto*
sepref-decl-impl *arl32-copy*: *arl32-copy-hnr-aux* **.**

lemma *arl32-append-hnr-aux*: (*uncurry arl32-append*,*uncurry* (*RETURN oo op-list-append*)) ∈ [λ(*xs*, *x*). *length xs* < *uint32-max*]$_a$ (*is-array-list32*^{d} $*_a$ *id-assn*^{k}) → *is-array-list32*
by *sep-auto*
sepref-decl-impl *arl32-append*: *arl32-append-hnr-aux*
unfolding *fref-param1* **by** (*auto intro*!: *frefI nres-relI simp*: *list-rel-imp-same-length*)

lemma *arl32-length-hnr-aux*: (*arl32-length*,*RETURN o op-list-length*) ∈ *is-array-list32*^{k} $\rightarrow_a$ *uint32-nat-assn*
by (*sep-auto simp*: *uint32-nat-rel-def br-def*)
sepref-decl-impl *arl32-length*: *arl32-length-hnr-aux* **.**

lemma *arl32-is-empty-hnr-aux*: (*arl32-is-empty*,*RETURN o op-list-is-empty*) ∈ *is-array-list32*^{k} $\rightarrow_a$ *bool-assn*
by *sep-auto*
sepref-decl-impl *arl32-is-empty*: *arl32-is-empty-hnr-aux* **.**

lemma *arl32-last-hnr-aux*: (*arl32-last*,*RETURN o op-list-last*) ∈ [*pre-list-last*]$_a$ *is-array-list32*^{k} → *id-assn*
by *sep-auto*
sepref-decl-impl *arl32-last*: *arl32-last-hnr-aux* **.**

lemma *arl32-get-hnr-aux*: (*uncurry arl32-get*,*uncurry* (*RETURN oo op-list-get*)) ∈ [λ(*l*,*i*). *i*<*length l*]$_a$ (*is-array-list32*^{k} $*_a$ *uint32-nat-assn*^{k}) → *id-assn*
by *sep-auto*
sepref-decl-impl *arl32-get*: *arl32-get-hnr-aux* **.**

lemma *arl32-set-hnr-aux*: (*uncurry2 arl32-set*,*uncurry2* (*RETURN ooo op-list-set*)) ∈ [λ((*l*,*i*),-). *i*<*length l*]$_a$ (*is-array-list32*^{d} $*_a$ *uint32-nat-assn*^{k} $*_a$ *id-assn*^{k}) → *is-array-list32*
by *sep-auto*
sepref-decl-impl *arl32-set*: *arl32-set-hnr-aux* **.**

sepref-definition *arl32-swap* **is** *uncurry2 mop-list-swap* :: ((*arl32-assn id-assn*)^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *arl32-assn id-assn*)
unfolding *gen-mop-list-swap*[*abs-def*]
by *sepref*

sepref-decl-impl (*ismop*) *arl32-swap*: *arl32-swap.refine* **.**
end

interpretation *arl32*: *list-custom-empty arl32-assn A arl32-empty op-arl32-empty*
apply *unfold-locales*
apply (*rule arl32-empty-hnr*)
by (*auto simp*: *op-arl32-empty-def*)

lemma [*def-pat-rules*]: *op-arl32-empty-sz*$N ≡ *UNPROTECT* (*op-arl32-empty-sz N*) **by** *simp*

interpretation *arl32-sz*: *list-custom-empty arl32-assn A arl32-empty-sz N PR-CONST* (*op-arl32-empty-sz N*)
apply *unfold-locales*
apply (*rule arl32-empty-sz-hnr*)
by (*auto simp*: *op-arl32-empty-sz-def*)

definition *arl32-to-arl-conv* **where**
‹*arl32-to-arl-conv S* = *S*›

definition *arl32-to-arl* :: ‹*'a array-list32* ⇒ *'a array-list*› **where**
‹*arl32-to-arl* = (λ(*xs*, *n*). (*xs*, *nat-of-uint32 n*))›

lemma *arl32-to-arl-hnr*[*sepref-fr-rules*]:
‹(*return o arl32-to-arl*, *RETURN o arl32-to-arl-conv*) ∈ (*arl32-assn R*)^d $\rightarrow_a$ *arl-assn R*›
by (*sepref-to-hoare*)
(*sep-auto simp*: *arl32-to-arl-def arl32-to-arl-conv-def arl-assn-def arl32-assn-def is-array-list32-def is-array-list-def hr-comp-def*)

definition *arl32-take* **where**
‹*arl32-take n* = (λ(*xs*, -). (*xs*, *n*))›

lemma *arl32-take*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo arl32-take*), *uncurry* (*RETURN oo take*)) ∈
[λ(*n*, *xs*). *n* ≤ *length xs*]$_a$ *uint32-nat-assn*^{k} $*_a$ (*arl32-assn R*)^d → *arl32-assn R*›
by (*sepref-to-hoare*)
(*sep-auto simp*: *arl32-assn-def arl32-take-def is-array-list32-def hr-comp-def uint32-nat-rel-def br-def list-rel-def list-all2-conv-all-nth*)

definition *arl32-butlast-nonresizing* :: ‹*'a array-list32* ⇒ *'a array-list32*› **where**
‹*arl32-butlast-nonresizing* = (λ(*xs*, *a*). (*xs*, *a* − *1*))›

lemma *butlast32-nonresizing-hnr*[*sepref-fr-rules*]:
‹(*return o arl32-butlast-nonresizing*, *RETURN o butlast-nonresizing*) ∈
[λ*xs*. *xs* ≠ []]$_a$ (*arl32-assn R*)^d → *arl32-assn R*›
proof −
have [*simp*]: ‹*nat-of-uint32* (*b* − *1*) = *nat-of-uint32 b* − *1*›
if
‹*x* ≠ []› **and**
‹(*take* (*nat-of-uint32 b*) *l'*, *x*) ∈ ⟨*the-pure R*⟩*list-rel*›
for *x* :: ‹*'b list*› **and** *a* :: ‹*'a array*› **and** *b* :: ‹*uint32*› **and** *l'* :: ‹*'a list*› **and** *aa* :: ‹*Heap.heap*› **and** *ba* :: ‹*nat set*›
by (*metis less-one list-rel-pres-neq-nil nat-of-uint32-012*(*3*) *nat-of-uint32-less-iff nat-of-uint32-notle-minus take-eq-Nil that*)

show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *arl32-butlast-nonresizing-def arl32-assn-def hr-comp-def*
is-array-list32-def butlast-take list-rel-imp-same-length nat-of-uint32-ge-minus
dest:
list-rel-butlast[*of* ‹*take* - -›]
simp flip: *nat-of-uint32-le-iff*)
qed

end

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

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

definition *choose-pivot* :: ‹($'b \Rightarrow 'b \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow 'b$) $\Rightarrow$ $'a$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat nres*› **where**
‹*choose-pivot* - - - *lo hi* = *SPEC*($\lambda k.\ k \geq lo \wedge k \leq hi$)›

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

definition *isPartition-wrt* :: ‹($'b \Rightarrow 'b \Rightarrow bool$) $\Rightarrow$ $'b$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*› **where**
‹*isPartition-wrt R xs lo hi p* $\equiv$ ($\forall\ i.\ i \geq lo \wedge i < p \longrightarrow R\ (xs!i)\ (xs!p)$) $\wedge$ ($\forall\ j.\ j > p \wedge j \leq hi \longrightarrow R\ (xs!p)\ (xs!j)$)›

lemma *isPartition-wrtI*:
‹($\bigwedge\ i.\ [\![i \geq lo;\ i < p]\!] \Longrightarrow R\ (xs!i)\ (xs!p)$) $\Longrightarrow$ ($\bigwedge\ j.\ [\![j > p;\ j \leq hi]\!] \Longrightarrow R\ (xs!p)\ (xs!j)$) $\Longrightarrow$ *isPartition-wrt R xs lo hi p*›
by (*simp add*: *isPartition-wrt-def*)

definition *isPartition* :: ‹$'a$ :: *order list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*› **where**
‹*isPartition xs lo hi p* $\equiv$ *isPartition-wrt* ($\leq$) *xs lo hi p*›

abbreviation *isPartition-map* :: ‹($'b \Rightarrow 'b \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow 'b$) $\Rightarrow$ $'a$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*›
where
‹*isPartition-map R h xs i j k* $\equiv$ *isPartition-wrt* ($\lambda a\ b.\ R\ (h\ a)\ (h\ b)$) *xs i j k*›

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

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

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

definition *sublist* :: ‹$'a$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *list*› **where**
‹*sublist xs i j* $\equiv$ *take* (*Suc j* − *i*) (*drop i xs*)›

lemma *take-Suc0*:
$l \neq [] \Longrightarrow$ *take* (*Suc 0*) *l* = [*l*!*0*]
0 < *length l* $\Longrightarrow$ *take* (*Suc 0*) *l* = [*l*!*0*]

Suc n ≤ *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
by (*cases l*, *auto*)+

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

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

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

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

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

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

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

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

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

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

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

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

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

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

apply (*auto simp add*: *sublist-def*)
by (*simp add*: *drop-map take-map*)

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

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

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

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

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

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

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

Elements in a sorted sublists are actually sorted

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

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

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

lemma *sorted-sublist-le*: ‹$hi \leq lo \Longrightarrow hi <$ *length xs* $\Longrightarrow$ *sorted-sublist xs lo hi*›
  by (*auto simp add*: *sorted-sublist-def sorted-sublist-wrt-le*)

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

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

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

lemma *sorted-sublist-wrt-cons*:
  assumes *trans*: ‹($\bigwedge$ *x y z*. $[\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z$)› **and**
    ‹*sorted-sublist-wrt R xs* (*lo*+*1*) *hi*› **and**
    ‹$lo \leq hi$› **and** ‹$hi <$ *length xs* › **and** ‹*R* (*xs*!*lo*) (*xs*!(*lo*+*1*))›
  shows ‹*sorted-sublist-wrt R xs lo hi*›
proof −
  show *?thesis*
    apply (*rule sorted-sublist-wrt-cons'*) **using** *assms* **apply** *auto*

subgoal premises *assms′* **for** *j*
proof −
have *A*: ‹*j*=*lo*+*1* ∨ *j*>*lo*+*1*› **using** *assms′*(*5*) **by** *linarith*
show *?thesis*
using *A* **proof**
assume *A*: ‹*j*=*lo*+*1*› **show** *?thesis*
by (*simp add*: *A assms′*)
next
assume *A*: ‹*j*>*lo*+*1*› **show** *?thesis*
apply (*rule trans*)
apply (*rule assms*(*5*))
apply (*rule sorted-sublist-wrt-nth-le*[*OF assms*(*2*), **where** *i*=‹*lo*+*1*›, **where** *j*=*j*])
subgoal using *A assms′*(*6*) **by** *linarith*
subgoal using *assms′*(*3*) *less-imp-diff-less* **by** *blast*
subgoal using *assms′*(*5*) **by** *auto*
subgoal using *A* **by** *linarith*
subgoal by (*simp add*: *assms′*(*6*))
done
qed
qed
done
qed

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

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

lemma *sorted-sublist-wrt-snoc′*:
‹*sorted-sublist-wrt R xs lo* (*hi*−*1*) ⟹ *lo* ≤ *hi* ⟹ *hi* < *length xs* ⟹ (∀ *j*. *lo*≤*j*∧*j*<*hi* ⟶ *R* (*xs*!*j*) (*xs*!*hi*)) ⟹ *sorted-sublist-wrt R xs lo hi*›
apply (*simp add*: *sorted-sublist-wrt-def*)
apply (*auto simp add*: *nat-le-eq-or-lt*)
subgoal by (*simp add*: *sublist-single*)
apply (*auto simp add*: *sublist-snoc sublist-el sorted-wrt-append*)
by (*metis less-diff-conv linorder-neqE-nat linordered-field-class.sign-simps*(*2*) *not-add-less1*)

lemma *sorted-sublist-wrt-snoc*:
assumes *trans*: ‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*)› **and**
‹*sorted-sublist-wrt R xs lo* (*hi*−*1*)› **and**

⟨$lo \le hi$⟩ **and** ⟨$hi < length\ xs$⟩ **and** ⟨$(R\ (xs!(hi-1))\ (xs!hi))$⟩
shows ⟨*sorted-sublist-wrt R xs lo hi*⟩
proof −
show *?thesis*
apply (*rule sorted-sublist-wrt-snoc'*) **using** *assms* **apply** *auto*
subgoal premises *assms'* **for** *j*
proof −
have *A*: ⟨$j=hi-1 \lor j<hi-1$⟩ **using** *assms'(6)* **by** *linarith*
show *?thesis*
using *A* **proof**
assume *A*: ⟨$j=hi-1$⟩ **show** *?thesis*
by (*simp add*: *A assms'*)
next
assume *A*: ⟨$j<hi-1$⟩ **show** *?thesis*
apply (*rule trans*)
apply (*rule sorted-sublist-wrt-nth-le*[*OF assms*(*2*), **where** $i=j$, **where** $j=⟨hi-1⟩$])
prefer *6*
apply (*rule assms*(*5*))
apply *auto*
subgoal using *A assms'(5)* **by** *linarith*
subgoal using *assms'(3) less-imp-diff-less* **by** *blast*
subgoal using *assms'(5)* **by** *auto*
subgoal using *A* **by** *linarith*
done
qed
qed
done
qed

lemma *sorted-sublist-map-snoc*:
⟨$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
sorted-sublist-map R h xs lo $(hi-1) \Longrightarrow$
$lo \le hi \Longrightarrow hi < length\ xs \Longrightarrow (R\ (h\ (xs!(hi-1)))\ (h\ (xs!hi))) \Longrightarrow$ *sorted-sublist-map R h xs lo hi*⟩
by (*blast intro*: *sorted-sublist-wrt-snoc*)

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

lemma *sublist-split-part*: ⟨$lo \le hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow$ *sublist xs lo* $(p-1)$ @ $xs!p$ # *sublist xs* $(p+1)$ $hi =$ *sublist xs lo hi*⟩
apply (*auto simp add*: *sublist-split*[*symmetric*])
apply (*rewrite sublist-snoc*[**where** $xs=xs$,**where** $lo=lo$,**where** $hi=p$])
by *auto*

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

lemma *isPartition-wrt-trans*:
⟨$(\bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z) \Longrightarrow$
isPartition-wrt R xs lo hi p $\Longrightarrow$
$(\forall\, i\ j.\ lo \le i \land i < p \land p < j \land j \le hi \longrightarrow R\ (xs!i)\ (xs!j))$⟩
by (*auto simp add*: *isPartition-wrt-def*)

lemma *isPartition-map-trans*:
‹($\bigwedge$ $x\ y\ z$. $[\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$) $\Longrightarrow$
$hi < length\ xs \Longrightarrow$
$isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p \Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (h\ (xs!i))\ (h\ (xs!j)))$›
by (*auto simp add*: *isPartition-wrt-def*)

lemma *merge-sorted-wrt-partitions-between′*:
‹$lo \le hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (p{-}1) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ (p{+}1)\ hi \Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
apply (*auto simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-def sorted-sublist-wrt-def sublist-map*)
apply (*simp add*: *sublist-split-part*[*symmetric*])
apply (*auto simp add*: *List.sorted-wrt-append*)
subgoal by (*auto simp add*: *sublist-el*)
subgoal by (*auto simp add*: *sublist-el*)
subgoal by (*auto simp add*: *sublist-el′*)
done

lemma *merge-sorted-wrt-partitions-between*:
‹($\bigwedge$ $x\ y\ z$. $[\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z$) $\Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (p{-}1) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ (p{+}1)\ hi \Longrightarrow$
$lo \le hi \Longrightarrow hi < length\ xs \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
by (*simp add*: *merge-sorted-wrt-partitions-between′ isPartition-wrt-trans*)

The main theorem to merge sorted lists

lemma *merge-sorted-wrt-partitions*:
‹$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (p - Suc\ 0) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ (Suc\ p)\ hi \Longrightarrow$
$lo \le hi \Longrightarrow lo \le p \Longrightarrow p \le hi \Longrightarrow hi < length\ xs \Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
subgoal premises *assms*
proof −
have *C*: ‹$lo{=}p \wedge p{=}hi \vee lo{=}p \wedge p{<}hi \vee lo{<}p \wedge p{=}hi \vee lo{<}p \wedge p{<}hi$›
using *assms* **by** *linarith*
show *?thesis*
using *C* **apply** *auto*
subgoal — lo=p=hi
apply (*rule sorted-sublist-wrt-refl*)
using *assms* **by** *auto*
subgoal — lo=p<hi
using *assms* **by** (*simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-wrt-cons′*)
subgoal — lo<p=hi
using *assms* **by** (*simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-wrt-snoc′*)
subgoal — lo<p<hi
using *assms*
apply (*rewrite merge-sorted-wrt-partitions-between′*[**where** *p*=*p*])
by *auto*
done

qed
done

theorem *merge-sorted-map-partitions*:
‹$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
$isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ (p - Suc\ 0) \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ (Suc\ p)\ hi \Longrightarrow$
$lo \le hi \Longrightarrow lo \le p \Longrightarrow p \le hi \Longrightarrow hi < length\ xs \Longrightarrow$
$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi$›
apply (*rule merge-sorted-wrt-partitions*) **apply** *auto*
by (*simp add*: *merge-sorted-wrt-partitions isPartition-map-trans*)

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

lemma *partition-map-extend*:
‹$isPartition\text{-}map\ R\ h\ xs\ lo'\ hi'\ p \Longrightarrow$
$hi < length\ xs \Longrightarrow$
$lo \le lo' \Longrightarrow lo' \le hi \Longrightarrow hi' \le hi \Longrightarrow$
$lo' \le p \Longrightarrow p \le hi' \Longrightarrow$
$(\bigwedge i.\ lo{\le}i \Longrightarrow i < lo' \Longrightarrow R\ (h\ (xs!i))\ (h\ (xs!p))) \Longrightarrow$
$(\bigwedge j.\ hi'{<}j \Longrightarrow j{\le}hi \Longrightarrow R\ (h\ (xs!p))\ (h\ (xs!j))) \Longrightarrow$
$isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p$›
by (*auto simp add*: *partition-wrt-extend*)

lemma *isPartition-empty*:
‹$(\bigwedge j.\ [\![lo < j;\ j \le hi]\!] \Longrightarrow R\ (xs\ !\ lo)\ (xs\ !\ j)) \Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ lo$›
by (*auto simp add*: *isPartition-wrt-def*)

lemma *take-ext*:
‹$(\forall i{<}k.\ xs'!i{=}xs!i) \Longrightarrow$
$k < length\ xs \Longrightarrow k < length\ xs' \Longrightarrow$
$take\ k\ xs' = take\ k\ xs$›
by (*simp add*: *nth-take-lemma*)

lemma *drop-ext'*:
‹$(\forall i.\ i{\ge}k \wedge i{<}length\ xs \longrightarrow xs'!i{=}xs!i) \Longrightarrow$
$0{<}k \Longrightarrow xs{\ne}[] \Longrightarrow$ — These corner cases will be dealt with in the next lemma
$length\ xs'{=}length\ xs \Longrightarrow$
$drop\ k\ xs' = drop\ k\ xs$›

```
  apply (rewrite in ‹drop - ¤ = -› List.rev-rev-ident[symmetric])
  apply (rewrite in ‹- = drop - ¤› List.rev-rev-ident[symmetric])
  apply (rewrite in ‹¤ = -› List.drop-rev)
  apply (rewrite in ‹- = ¤› List.drop-rev)
  apply simp
  apply (rule take-ext)
  by (auto simp add: nth-rev)

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


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


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

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

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

lemma sorted-wrt-lower-sublist-still-sorted:
  assumes ‹sorted-sublist-wrt R xs lo (lo' − Suc 0)› and
    ‹lo ≤ lo'› and ‹lo' < length xs› and
    ‹(∀  i. lo≤i∧i<lo' ⟶ xs'!i=xs!i)› and ‹length xs' = length xs›
  shows ‹sorted-sublist-wrt R xs' lo (lo' − Suc 0)›
proof −
  have l: ‹lo < lo' − 1 ∨ lo ≥ lo'−1›
    by linarith
  show ?thesis
    using l apply auto
```

subgoal — lo < lo' - 1
apply (*auto simp add*: *sorted-sublist-wrt-def*)
apply (*rewrite sublist-ext*[**where** *xs*=*xs*])
using *assms* **by** (*auto simp add*: *sorted-sublist-wrt-def*)
subgoal — lo >= lo' - 1
using *assms* **by** (*auto simp add*: *sorted-sublist-wrt-le*)
done
qed

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

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

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

The specification of the partition function

definition *partition-spec* :: ⟨(*′b* ⇒ *′b* ⇒ *bool*) ⇒ (*′a* ⇒ *′b*) ⇒ *′a list* ⇒ *nat* ⇒ *nat* ⇒ *′a list* ⇒ *nat* ⇒ *bool*⟩ **where**
⟨*partition-spec R h xs lo hi xs′ p* ≡
mset xs′ = *mset xs* ∧ — The list is a permutation
isPartition-map R h xs′ lo hi p ∧ — We have a valid partition on the resulting list
lo ≤ *p* ∧ *p* ≤ *hi* ∧ — The partition index is in bounds
(∀ *i. i*<*lo* ⟶ *xs′*!*i*=*xs*!*i*) ∧ (∀ *i. hi*<*i*∧*i*<*length xs′* ⟶ *xs′*!*i*=*xs*!*i*)⟩ — Everything else is unchanged.

lemma *mathias*:
assumes
Perm: ⟨*mset xs′* = *mset xs*⟩

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

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

But we only need that the sets are equal.

lemma *mset-sublist-incl*:
assumes *Perm*: ‹$mset\ xs' = mset\ xs$›
and *Fix*: ‹$\bigwedge i.\ i<lo \Longrightarrow xs'!i = xs!i$› ‹$\bigwedge j.\ [\![hi<j; j<length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
and *bounds*: ‹$lo \leq hi$› ‹$hi < length\ xs$›
shows ‹$set\ (sublist\ xs'\ lo\ hi) \subseteq set\ (sublist\ xs\ lo\ hi)$›
proof
fix *x*
assume ‹$x \in set\ (sublist\ xs'\ lo\ hi)$›
then have ‹$\exists i.\ lo \leq i \wedge i \leq hi \wedge xs'!i = x$›
by (*metis assms*(*1*) *bounds*(*1*) *bounds*(*2*) *size-mset sublist-el′*)
then obtain *i* **where** *I*: ‹$lo \leq i$› ‹$i \leq hi$› ‹$xs'!i = x$› **by** *blast*
have ‹$\exists j.\ lo \leq j \wedge j \leq hi \wedge xs!j = x$›
using *Perm I bounds*(*2*) *Fix* **by** (*rule mathias*, *auto*)
then show ‹$x \in set\ (sublist\ xs\ lo\ hi)$›

by (*simp add*: *bounds*(*1*) *bounds*(*2*) *sublist-el'*)
qed

lemma *mset-sublist-eq*:
assumes ‹$mset\ xs' = mset\ xs$›
and ‹$\bigwedge\ i.\ i{<}lo \Longrightarrow xs'!i = xs!i$›
and ‹$\bigwedge\ j.\ [\![hi{<}j;\ j{<}length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
and *bounds*: ‹$lo \le hi$› ‹$hi < length\ xs$›
shows ‹$set\ (sublist\ xs'\ lo\ hi) = set\ (sublist\ xs\ lo\ hi)$›
proof
show ‹$set\ (sublist\ xs'\ lo\ hi) \subseteq set\ (sublist\ xs\ lo\ hi)$›
apply (*rule mset-sublist-incl*)
using *assms* **by** *auto*
show ‹$set\ (sublist\ xs\ lo\ hi) \subseteq set\ (sublist\ xs'\ lo\ hi)$›
apply (*rule mset-sublist-incl*)
by (*metis assms size-mset*)+
qed

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

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

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

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

definition *quicksort-post* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$›
where
‹*quicksort-post R h lo hi xs xs'* ≡
mset xs' = *mset xs* ∧
sorted-sublist-map R h xs' lo hi ∧
($\forall\ i.\ i{<}lo \longrightarrow xs'!i = xs!i$) ∧
($\forall\ j.\ hi{<}j \wedge j{<}length\ xs \longrightarrow xs'!j = xs!j$)›

Convert Pure to HOL

lemma *quicksort-postI*:
‹$[\![mset\ xs' = mset\ xs;$ *sorted-sublist-map R h xs' lo hi*; $(\bigwedge\ i.\ [\![i{<}lo]\!] \Longrightarrow xs'!i = xs!i);\ (\bigwedge\ j.\ [\![hi{<}j;\ j{<}length\ xs]\!] \Longrightarrow xs'!j = xs!j)]\!] \Longrightarrow$ *quicksort-post R h lo hi xs xs'*›
by (*auto simp add*: *quicksort-post-def*)

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

lemma *quicksort-correct-case1*:

assumes *trans*: ‹$\bigwedge$ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹$\bigwedge$*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
and *ifs*: ‹*p*−*1* ≤ *lo*› ‹*hi* ≤ *p*+*1*›
shows ‹*quicksort-post R h lo hi xs xs′*›
proof −

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

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

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


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

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

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

show ?thesis
proof (intro quicksort-postI)
  show ‹mset xs′ = mset xs›
    by (simp add: part(1))
```

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

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

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

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

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

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

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

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

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

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

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

This holds because this part hasn't changed

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

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

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

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

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

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

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

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

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

Analogous to case 2.

lemma *quicksort-correct-case4*:
  assumes
      pre: ‹*quicksort-pre R h xs0 lo hi xs*›
    and *part*: ‹*partition-spec R h xs lo hi xs' p*›
    and *ifs*: ‹$\neg\ p - Suc\ 0 \leq lo$ ›
  shows ‹*quicksort-pre R h xs0 lo* ($p-Suc\ 0$) *xs'*›
proof −

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

have *pre*: ‹$lo \leq hi$› ‹$hi < length\ xs$› ‹*mset xs0* = *mset xs*›
  using *pre* **by** (*auto simp add*: *quicksort-pre-def*)
have *part*: ‹*mset xs'* = *mset xs*› *True*
  ‹*isPartition-map R h xs' lo hi p*› ‹$lo \leq p$› ‹$p \leq hi$›
  ‹$\bigwedge\ i.\ i<lo \Longrightarrow xs'!i=xs!i$› ‹$\bigwedge\ i.\ [\![hi<i;\ i<length\ xs']\!] \Longrightarrow xs'!i=xs!i$›
  using *part* **by** (*auto simp add*: *partition-spec-def*)

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

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

lemma *quicksort-correct-case5*:
assumes *trans*: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *pre*: ‹$quicksort\text{-}pre\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$partition\text{-}spec\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$\neg\ p - Suc\ 0 \leq lo$› ‹$hi \leq Suc\ p$›
and *IH1'*: ‹$quicksort\text{-}post\ R\ h\ lo\ (p - Suc\ 0)\ xs'\ xs''$›
shows ‹$quicksort\text{-}post\ R\ h\ lo\ hi\ xs\ xs''$›
proof −

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

have *pre*: ‹$lo \leq hi$› ‹$hi < length\ xs$›
using *pre* **by** (*auto simp add*: *quicksort-pre-def*)
have *part*: ‹$mset\ xs' = mset\ xs$› *True*
‹$isPartition\text{-}map\ R\ h\ xs'\ lo\ hi\ p$› ‹$lo \leq p$› ‹$p \leq hi$›
‹$\bigwedge\ i.\ i<lo \Longrightarrow xs'!i=xs!i$› ‹$\bigwedge\ i.\ [\![hi<i;\ i<length\ xs']\!] \Longrightarrow xs'!i=xs!i$›
using *part* **by** (*auto simp add*: *partition-spec-def*)
have *IH1*: ‹$mset\ xs'' = mset\ xs'$› ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs''\ lo\ (p - Suc\ 0)$›
‹$\bigwedge\ i.\ i<lo \Longrightarrow xs''!i = xs'!i$› ‹$\bigwedge\ j.\ [\![p-Suc\ 0<j;\ j<length\ xs']\!] \Longrightarrow xs''!j = xs'!j$›
using *IH1'* **by** (*auto simp add*: *quicksort-post-def*)
note *IH1-perm* = *quicksort-post-set*[*OF IH1'*]

have *still-partition*: ‹$isPartition\text{-}map\ R\ h\ xs''\ lo\ hi\ p$›
proof(*intro isPartition-wrtI*)
fix *i* **assume** ‹$lo \leq i$› ‹$i < p$›

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

have ‹$xs''!i \in set\ (sublist\ xs''\ lo\ (p-Suc\ 0))$›
by (*metis* (*no-types*, *lifting*) *IH1(1) Suc-leI Suc-pred* ‹$i < p$› ‹$lo \leq i$› *le-less-trans less-imp-diff-less mset-eq-length not-le not-less-zero part(1) part(5) pre(2) sublist-el'*)
then have ‹$xs''!i \in set\ (sublist\ xs'\ lo\ (p-Suc\ 0))$›
by (*metis IH1-perm ifs(1) le-less-trans less-imp-diff-less mset-eq-length nat-le-linear part(1) part(5) pre(2)*)
then have ‹$\exists\ posI.\ lo\leq posI \wedge posI\leq p-Suc\ 0 \wedge xs''!i = xs'!posI$›
proof − — sledgehammer
have $p - Suc\ 0 < length\ xs$
by (*meson diff-le-self le-less-trans part(5) pre(2)*)
then show *?thesis*
by (*metis* (*no-types*) ‹$xs''\ !\ i \in set\ (sublist\ xs'\ lo\ (p - Suc\ 0))$› *ifs(1) mset-eq-length nat-le-linear part(1) sublist-el'*)

qed
then obtain *posI* :: *nat* **where** *PosI*: ‹$lo \le posI$› ‹$posI \le p - Suc\ 0$› ‹$xs''!i = xs'!posI$› **by** *blast*
then show ‹$R\ (h\ (xs''\ !\ i))\ (h\ (xs''\ !\ p))$›
by (*metis* (*no-types*, *lifting*) *IH1*(*4*) ‹$i < p$› *diff-Suc-less isPartition-wrt-def le-less-trans mset-eq-length not-le not-less-eq part*(*1*) *part*(*3*) *part*(*5*) *pre*(*2*) *zero-less-Suc*)
next
fix *j* **assume** ‹$p < j$› ‹$j \le hi$›
then show ‹$R\ (h\ (xs''\ !\ p))\ (h\ (xs''\ !\ j))$›

This holds because this part hasn't changed

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

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

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

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

show *?thesis*
proof (*intro quicksort-postI*)
show ‹*mset xs''* = *mset xs*›
by (*simp add*: *IH1*(*1*) *part*(*1*))
next
show ‹*sorted-sublist-map R h xs'' lo hi*›
by (*rule sorted-middle*)
next
show ‹$\bigwedge i.\ i < lo \Longrightarrow xs''\ !\ i = xs\ !\ i$›
by (*simp add*: *IH1*(*3*) *part*(*6*))
next
show ‹$\bigwedge j.\ hi < j \Longrightarrow j < length\ xs \Longrightarrow xs''\ !\ j = xs\ !\ j$›

    by (*metis IH1*(*4*) *diff-le-self dual-order.strict-trans2 mset-eq-length part*(*1*) *part*(*5*) *part*(*7*))
  qed
qed

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

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

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

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

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

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

lemma *quicksort-correct-case7*:
  assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
    and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
    and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
    and *ifs*: ‹$\neg$ *p* − *Suc 0* ≤ *lo*› ‹$\neg$ *hi* ≤ *Suc p*›
    and *IH1′*: ‹*quicksort-post R h lo* (*p* − *Suc 0*) *xs′ xs′′*›
    and *IH2′*: ‹*quicksort-post R h* (*Suc p*) *hi xs′′ xs′′′*›
  shows ‹*quicksort-post R h lo hi xs xs′′′*›
proof −

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

  have *pre*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›

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

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

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

Obtain the position *posI* where *xs″* ! *i* was stored in *xs′*.

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

This holds because this part hasn't changed

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

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

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

This holds because this part hasn't changed

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

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

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

then show ‹$R\ (h\ (xs'''\ !\ p))\ (h\ (xs'''\ !\ j))$›
proof — — sledgehammer
have $\forall\, n\ na\ as\ p.\ (p\ (as\ !\ na{::}'a)\ (as\ !\ posJ) \vee posJ \leq na) \vee \neg\ isPartition\text{-}wrt\ p\ as\ n\ hi\ na$
by (*metis (no-types) PosJ(2) isPartition-wrt-def not-less*)
then show *?thesis*
by (*metis IH2(3) PosJ(1) PosJ(3) lessI not-less-eq-eq still-partition1*)
qed
qed

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

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

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

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

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

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

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

have *sorted-middle*: ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs'''\ lo\ hi$›
proof –
show *?thesis*
apply (*rule merge-sorted-map-partitions*[**where** *p=p*])
subgoal by (*rule trans*)
subgoal by (*rule still-partition2*)
subgoal by (*rule sorted-lower2*)
subgoal by (*rule sorted-upper2*)
subgoal using *pre(1)* **by** *auto*
subgoal by (*simp add: part(4)*)

subgoal by (*simp add*: *part*(*5*))
subgoal by (*metis IH1*(*1*) *IH2*(*1*) *part*(*1*) *pre*(*2*) *size-mset*)
done
qed

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

qed

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

lemma *quicksort-correct*:
assumes *trans*: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *Pre*: ‹$lo0 \leq hi0$› ‹$hi0 < length\ xs0$›
shows ‹$quicksort\ R\ h\ (lo0,hi0,xs0) \leq \Downarrow Id\ (SPEC(\lambda xs.\ quicksort\text{-}post\ R\ h\ lo0\ hi0\ xs0\ xs))$›
proof −
have *wf*: ‹$wf\ (measure\ (\lambda(lo,\ hi,\ xs).\ Suc\ hi - lo))$›
by *auto*
define *pre* **where** ‹$pre = (\lambda(lo,hi,xs).\ quicksort\text{-}pre\ R\ h\ xs0\ lo\ hi\ xs)$›
define *post* **where** ‹$post = (\lambda(lo,hi,xs).\ quicksort\text{-}post\ R\ h\ lo\ hi\ xs)$›
have *pre*: ‹$pre\ (lo0,hi0,xs0)$›
unfolding *quicksort-pre-def pre-def* **by** (*simp add*: *Pre*)

We first generalize the goal a over all states.

have ‹$WB\text{-}Sort.quicksort\ R\ h\ (lo0,hi0,xs0) \leq \Downarrow Id\ (SPEC\ (post\ (lo0,hi0,xs0)))$›
unfolding *quicksort-def prod.case*
apply (*rule RECT-rule*)
apply (*refine-mono*)
apply (*rule wf*)
apply (*rule pre*)
subgoal premises *IH* **for** *f x*
apply (*refine-vcg ASSERT-leI*)
unfolding *pre-def post-def*

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

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

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

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

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

Show that the invariant holds for the second recursive call

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

Wellfoundness (easy)

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

Show that the postcondition holds

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

Case: At least the first recursive call

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

Show that the precondition holds for the first recursive call

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

Wellfoundness for first recursive call (easy)

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

Simplify some refinement suff...

apply (*simp add*: *Misc.subset-Collect-conv ASSERT-leI*, *intro allI impI conjI*, *elim conjE*)
apply (*rule ASSERT-leI*)
apply (*simp-all add*: *Misc.subset-Collect-conv ASSERT-leI*)
subgoal unfolding *quicksort-post-def pre-def post-def* **by** (*auto dest*: *mset-eq-setD*)

Only the first recursive call: show postcondition

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

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

Both recursive calls.

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

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

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

Wellfoundedness for second recursive call (easy)

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

Show that the postcondition holds (after both recursive calls)

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

Finally, apply the generalized lemma to show the thesis.

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

definition *partition-main-inv* :: ‹($'b \Rightarrow 'b \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow 'b$) $\Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow (nat \times nat \times 'a\ list) \Rightarrow bool$› **where**
‹*partition-main-inv R h lo hi xs0 p* $\equiv$
case p of $(i,j,xs) \Rightarrow$
$j < length\ xs \wedge j \leq hi \wedge i < length\ xs \wedge lo \leq i \wedge i \leq j \wedge mset\ xs = mset\ xs0\ \wedge$
$(\forall k.\ k \geq lo \wedge k < i \longrightarrow R\ (h\ (xs!k))\ (h\ (xs!hi)))\ \wedge$ — All elements from *lo* to $i - (1::'c)$ are smaller than the pivot
$(\forall k.\ k \geq i \wedge k < j \longrightarrow R\ (h\ (xs!hi))\ (h\ (xs!k)))\ \wedge$ — All elements from i to $j - (1::'c)$ are greater than the pivot
$(\forall k.\ k < lo \longrightarrow xs!k = xs0!k)\ \wedge$ — Everything below *lo* is unchanged
$(\forall k.\ k \geq j \wedge k < length\ xs \longrightarrow xs!k = xs0!k)$ — All elements from j are unchanged (including everyting above *hi*)
›

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

definition *partition-main* :: ‹($'b \Rightarrow 'b \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow 'b$) $\Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a\ list \times nat)\ nres$› **where**
‹*partition-main R h lo hi xs0* = *do* {
ASSERT($hi < length\ xs0$);
pivot $\leftarrow$ *RETURN* ($h\ (xs0\ !\ hi)$);
$(i,j,xs) \leftarrow WHILE_T{}^{partition\text{-}main\text{-}inv\ R\ h\ lo\ hi\ xs0}$ — We loop from $j = lo$ to $j = hi - (1::'c)$.

$(\lambda(i,j,xs).\ j < hi)$
$(\lambda(i,j,xs).\ do\ \{$
  $ASSERT(i < length\ xs \wedge j < length\ xs);$
  if $R\ (h\ (xs!j))\ pivot$
  then $RETURN\ (i+1,\ j+1,\ swap\ xs\ i\ j)$
  else $RETURN\ (i,\ \ j+1,\ xs)$
$\})$
$(lo,\ lo,\ xs0)$; — i and j are both initialized to lo
$ASSERT(i < length\ xs \wedge j = hi \wedge lo \leq i \wedge hi < length\ xs \wedge mset\ xs = mset\ xs0);$
$RETURN\ (swap\ xs\ i\ hi,\ i)$
$\}$›

lemma *partition-main-correct*:
assumes *bounds*: ‹$hi < length\ xs$› ‹$lo \leq hi$› **and**
trans: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹$partition\text{-}main\ R\ h\ lo\ hi\ xs \leq SPEC(\lambda(xs',\ p).\ mset\ xs = mset\ xs' \wedge$
$lo \leq p \wedge p \leq hi \wedge isPartition\text{-}map\ R\ h\ xs'\ lo\ hi\ p \wedge (\forall\ i.\ i<lo \longrightarrow xs'!i=xs!i) \wedge (\forall\ i.\ hi<i\wedge i<length\ xs' \longrightarrow xs'!i=xs!i))$›
proof –
have K: ‹$b \leq hi - Suc\ n \Longrightarrow n > 0 \Longrightarrow Suc\ n \leq hi \Longrightarrow Suc\ b \leq hi - n$› **for** $b\ hi\ n$
by *auto*
have L: ‹$\sim\ R\ (h\ x)\ (h\ y) \Longrightarrow R\ (h\ y)\ (h\ x)$› **for** $x\ y$ — Corollary of linearity
using *assms* **by** *blast*
have M: ‹$a < Suc\ b \equiv a = b \vee a < b$› **for** $a\ b$
by *linarith*
have N: ‹$(a::nat) \leq b \equiv a = b \vee a < b$› **for** $a\ b$
by *arith*

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

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

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

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

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

lemma *partition-between-correct*:
assumes ‹$hi < length\ xs$› **and** ‹$lo \leq hi$› **and**
‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹$partition\text{-}between\ R\ h\ lo\ hi\ xs \leq SPEC(uncurry\ (partition\text{-}spec\ R\ h\ xs\ lo\ hi))$›
proof −
have *K*: ‹$b \leq hi - Suc\ n \Longrightarrow n > 0 \Longrightarrow Suc\ n \leq hi \Longrightarrow Suc\ b \leq hi - n$› **for** *b hi n*
by *auto*
show *?thesis*
unfolding *partition-between-def choose-pivot-def*
apply (*refine-vcg partition-main-correct*)
using *assms* **apply** (*auto dest*: *mset-eq-length simp add*: *partition-spec-def*)
by (*metis dual-order.strict-trans2 less-imp-not-eq2 mset-eq-length swap-nth*)
qed

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

definition *choose-pivot3* **where**
‹*choose-pivot3 R h xs lo* (*hi*::*nat*) = *do* {
$ASSERT(lo < length\ xs)$;
$ASSERT(hi < length\ xs)$;
$let\ k' = (hi - lo)\ div\ 2$;
$let\ k = lo + k'$;
$ASSERT(k < length\ xs)$;
$let\ start = h\ (xs\ !\ lo)$;
$let\ mid = h\ (xs\ !\ k)$;
$let\ end = h\ (xs\ !\ hi)$;
$if\ (R\ start\ mid \wedge R\ mid\ end) \vee (R\ end\ mid \wedge R\ mid\ start)\ then\ RETURN\ k$
$else\ if\ (R\ start\ end \wedge R\ end\ mid) \vee (R\ mid\ end \wedge R\ end\ start)\ then\ RETURN\ hi$
$else\ RETURN\ lo$
}›

— We only have to show that this procedure yields a valid index between *lo* and *hi*.
lemma *choose-pivot3-choose-pivot*:
  assumes ‹$lo < length\ xs$› ‹$hi < length\ xs$› ‹$hi \geq lo$›
  shows ‹$choose\text{-}pivot3\ R\ h\ xs\ lo\ hi \leq \Downarrow\ Id\ (choose\text{-}pivot\ R\ h\ xs\ lo\ hi)$›
  unfolding *choose-pivot3-def choose-pivot-def*
  using *assms* **by** (*auto intro*!: *ASSERT-leI simp*: *Let-def*)

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

definition *partition-between-ref*
  :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a\ list \times nat)\ nres$›
where
  ‹*partition-between-ref R h lo hi xs0* = *do* {
    $ASSERT(hi < length\ xs0 \wedge hi < length\ xs0 \wedge lo \leq hi)$;
    $k \leftarrow choose\text{-}pivot3\ R\ h\ xs0\ lo\ hi$; — choice of pivot
    $ASSERT(k < length\ xs0)$;
    $xs \leftarrow RETURN\ (swap\ xs0\ k\ hi)$; — move the pivot to the last position, before we start the actual loop
    $ASSERT(length\ xs = length\ xs0)$;
    partition-main R h lo hi xs
  }›

lemma *partition-main-ref'*:
  ‹*partition-main R h lo hi xs*
    $\leq \Downarrow\ ((\lambda\ a\ b\ c\ d.\ Id)\ a\ b\ c\ d)\ (partition\text{-}main\ R\ h\ lo\ hi\ xs)$›
  by *auto*

lemma *partition-between-ref-partition-between*:
  ‹$partition\text{-}between\text{-}ref\ R\ h\ lo\ hi\ xs \leq (partition\text{-}between\ R\ h\ lo\ hi\ xs)$›
proof −
  have *swap*: ‹$(swap\ xs\ k\ hi,\ swap\ xs\ ka\ hi) \in Id$› **if** ‹$k = ka$›
    for *k ka*
    using *that* **by** *auto*
  have [*refine0*]: ‹$(h\ (xsa\ !\ hi),\ h\ (xsaa\ !\ hi)) \in Id$›
    if ‹$(xsa,\ xsaa) \in Id$›
    for *xsa xsaa*
    using *that* **by** *auto*

  show *?thesis*
    apply (*subst* (*2*) *Down-id-eq*[*symmetric*])
    unfolding *partition-between-ref-def*
      partition-between-def
      OP-def
    apply (*refine-vcg choose-pivot3-choose-pivot swap partition-main-correct*)
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    by (*auto intro*: *Refine-Basic.Id-refine dest*: *mset-eq-length*)

qed

Technical lemma for sepref

lemma *partition-between-ref-partition-between′*:
‹(*uncurry2* (*partition-between-ref R h*), *uncurry2* (*partition-between R h*)) ∈
nat-rel $\times_f$ *nat-rel* $\times_f$ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel* $\times_r$ *nat-rel*⟩*nres-rel*›
by (*intro frefI nres-relI*)
(*auto intro*: *partition-between-ref-partition-between*)

Example instantiation for pivot

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

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

term *quicksort*

Refined quicksort algorithm: We use the refined partition function.

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

lemma *quicksort-ref-quicksort*:
assumes *bounds*: ‹*hi* < *length xs*› ‹*lo* ≤ *hi*› **and**
trans: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*quicksort-ref R h x0* ≤ ⇓ *Id* (*quicksort R h x0*)›
proof −
have *wf*: ‹*wf* (*measure* (λ(*lo*, *hi*, *xs*). *Suc hi* − *lo*))›

by *auto*
have *pre*: ‹$x0 = x0' \Longrightarrow (x0, x0') \in Id \times_r Id \times_r \langle Id\rangle list\text{-}rel$› **for** *x0 x0'* :: ‹*nat* × *nat* × *'b list*›
by *auto*
have [*refine0*]: ‹$(x1e = x1d) \Longrightarrow (x1e,x1d) \in Id$› **for** *x1e x1d* :: ‹*'b list*›
by *auto*

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

First assertion (premise for partition)

subgoal
by *auto*

First assertion (premise for partition)

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

Correctness of the concrete partition function

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

by *simp+*
qed

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

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

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

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

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

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

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

Final correctness lemma

lemma *full-quicksort-correct-sorted*:
assumes
trans: ‹$\bigwedge$*x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹$\bigwedge$*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*full-quicksort R h xs* ≤ ⇓ *Id* (*SPEC*(λ*xs'*. *mset xs'* = *mset xs* ∧ *sorted-wrt* (λ *x y*. *R* (*h x*) (*h y*)) *xs'*))›
proof −
show *?thesis*
unfolding *full-quicksort-def*
apply (*refine-vcg*)
subgoal by *simp* — case xs=[]
subgoal by *simp* — case xs=[]

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

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

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

end
theory *WB-Sort-SML*
imports *WB-Sort WB-More-IICF-SML*
begin

named-theorems *isasat-codegen*

lemma *swap-match*[*isasat-codegen*]: ‹*WB-More-Refinement-List.swap* = *IICF-List.swap*›
by (*auto simp*: *WB-More-Refinement-List.swap-def IICF-List.swap-def intro*!: *ext*)

sepref-register *choose-pivot3*

Example instantiation code for pivot

sepref-definition *choose-pivot3-impl-code*
is ‹*uncurry2* (*choose-pivot3-impl*)›
:: ‹$(arl\text{-}assn\ nat\text{-}assn)^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a nat\text{-}assn$›
unfolding *choose-pivot3-impl-def choose-pivot3-def id-def*
by *sepref*

declare *choose-pivot3-impl-code.refine*[*sepref-fr-rules*]

Example instantiation for *partition-main*

definition *partition-main-impl* **where**
‹*partition-main-impl* = *partition-main* (≤) *id*›

sepref-register *partition-main-impl*

Example instantiation code for *partition-main*

sepref-definition *partition-main-code*
is ‹*uncurry2* (*partition-main-impl*)›
:: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
$arl\text{-}assn\ nat\text{-}assn *a\ nat\text{-}assn$›

unfolding *partition-main-impl-def partition-main-def*
id-def isasat-codegen
by *sepref*

declare *partition-main-code.refine*[*sepref-fr-rules*]

Example instantiation for partition

definition *partition-between-impl* **where**
$\langle$*partition-between-impl* = *partition-between-ref* $(\leq)$ *id*$\rangle$

sepref-register *partition-between-ref*

Example instantiation code for partition

sepref-definition *partition-between-code*
is $\langle$*uncurry2* (*partition-between-impl*)$\rangle$
:: $\langle nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
$arl\text{-}assn\ nat\text{-}assn * a\ nat\text{-}assn\rangle$
unfolding *partition-between-ref-def partition-between-impl-def*
choose-pivot3-impl-def[*symmetric*] *partition-main-impl-def*[*symmetric*]
unfolding *id-def isasat-codegen*
by *sepref*

declare *partition-between-code.refine*[*sepref-fr-rules*]

— Example implementation
definition *quicksort-impl* **where**
$\langle$*quicksort-impl a b c* $\equiv$ *quicksort-ref* $(\leq)$ *id* $(a,b,c)\rangle$

sepref-register *quicksort-impl*

— Example implementation code
sepref-definition
quicksort-code
is $\langle$*uncurry2 quicksort-impl*$\rangle$
:: $\langle nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
$arl\text{-}assn\ nat\text{-}assn\rangle$
unfolding *partition-between-impl-def*[*symmetric*]
quicksort-impl-def quicksort-ref-def
by *sepref*

declare *quicksort-code.refine*[*sepref-fr-rules*]

Executable code for the example instance

sepref-definition *full-quicksort-code*
is $\langle$*full-quicksort-impl*$\rangle$
:: $\langle (arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
$arl\text{-}assn\ nat\text{-}assn\rangle$
unfolding *full-quicksort-impl-def full-quicksort-ref-def quicksort-impl-def*[*symmetric*] *List.null-def*
by *sepref*

Export the code

export-code $\langle$*nat-of-integer*$\rangle$ $\langle$*integer-of-nat*$\rangle$ $\langle$*partition-between-code*$\rangle$ $\langle$*full-quicksort-code*$\rangle$ **in** *SML-imp*
module-name *IsaQuicksort* **file** *code/quicksort.sml*

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

Chapter 1

Two-Watched Literals

1.1 Rule-based system

1.1.1 Types and Transitions System

Types and accessing functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Transition System

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

inductive *cdcl-twl-cp* :: ‹'v twl-st ⇒ 'v twl-st ⇒ bool› **where**
pop:
‹*cdcl-twl-cp* (M, N, U, None, NE, UE, {#}, add-mset L Q)
(M, N, U, None, NE, UE, {#(L, C)|C ∈# N + U. L ∈# watched C#}, Q)› |
propagate:
‹*cdcl-twl-cp* (M, N, U, None, NE, UE, add-mset (L, D) WS, Q)

(*Propagated* L' (*clause* D) # M, N, U, *None*, *NE*, *UE*, *WS*, *add-mset* $(-L')$ Q)›
if
‹*watched* $D = \{\#L,\ L'\#\}$› **and** ‹*undefined-lit* M L'› **and** ‹$\forall$ $L \in\#$ *unwatched* D. $-L \in$ *lits-of-l* M› |
conflict:
‹*cdcl-twl-cp* (M, N, U, *None*, *NE*, *UE*, *add-mset* (L, D) *WS*, Q)
(M, N, U, *Some* (*clause* D), *NE*, *UE*, $\{\#\}$, $\{\#\}$)›
if ‹*watched* $D = \{\#L,\ L'\#\}$› **and** ‹$-L' \in$ *lits-of-l* M› **and** ‹$\forall$ $L \in\#$ *unwatched* D. $-L \in$ *lits-of-l* M› |
delete-from-working:
‹*cdcl-twl-cp* (M, N, U, *None*, *NE*, *UE*, *add-mset* (L, D) *WS*, Q) (M, N, U, *None*, *NE*, *UE*, *WS*, Q)›
if ‹$L' \in\#$ *clause* D› **and** ‹$L' \in$ *lits-of-l* M› |
update-clause:
‹*cdcl-twl-cp* (M, N, U, *None*, *NE*, *UE*, *add-mset* (L, D) *WS*, Q)
(M, N', U', *None*, *NE*, *UE*, *WS*, Q)›
if ‹*watched* $D = \{\#L,\ L'\#\}$› **and** ‹$-L \in$ *lits-of-l* M› **and** ‹$L' \notin$ *lits-of-l* M› **and**
‹$K \in\#$ *unwatched* D› **and** ‹*undefined-lit* M K $\vee$ $K \in$ *lits-of-l* M› **and**
‹*update-clauses* (N, U) D L K (N', U')›
— The condition $-$ $L \in$ *lits-of-l* M is already implied by *valid* invariant.

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

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

inductive *cdcl-twl-o* :: ‹$'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st* $\Rightarrow$ *bool*› **where**
decide:
‹*cdcl-twl-o* (M, N, U, *None*, *NE*, *UE*, $\{\#\}$, $\{\#\}$) (*Decided* L # M, N, U, *None*, *NE*, *UE*, $\{\#\}$, $\{\#-L\#\}$)›
if ‹*undefined-lit* M L› **and** ‹*atm-of* $L \in$ *atms-of-mm* (*clause* ‘# N + *NE*)›
| *skip*:
‹*cdcl-twl-o* (*Propagated* L C' # M, N, U, *Some* D, *NE*, *UE*, $\{\#\}$, $\{\#\}$)
(M, N, U, *Some* D, *NE*, *UE*, $\{\#\}$, $\{\#\}$)›
if ‹$-L \notin\#$ D› **and** ‹$D \neq \{\#\}$›
| *resolve*:
‹*cdcl-twl-o* (*Propagated* L C # M, N, U, *Some* D, *NE*, *UE*, $\{\#\}$, $\{\#\}$)
(M, N, U, *Some* ($cdcl_W$*-restart-mset.resolve-cls* L D C), *NE*, *UE*, $\{\#\}$, $\{\#\}$)›
if ‹$-L \in\#$ D› **and**
‹*get-maximum-level* (*Propagated* L C # M) (*remove1-mset* $(-L)$ D) = *count-decided* M›
| *backtrack-unit-clause*:
‹*cdcl-twl-o* (M, N, U, *Some* D, *NE*, *UE*, $\{\#\}$, $\{\#\}$)
(*Propagated* L $\{\#L\#\}$ # *M1*, N, U, *None*, *NE*, *add-mset* $\{\#L\#\}$ *UE*, $\{\#\}$, $\{\#-L\#\}$)›
if
‹$L \in\#$ D› **and**
‹(*Decided* K # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* M)› **and**
‹*get-level* M L = *count-decided* M› **and**
‹*get-level* M L = *get-maximum-level* M D'› **and**
‹*get-maximum-level* M $(D' - \{\#L\#\}) \equiv i$› **and**
‹*get-level* M $K = i + 1$›
‹$D' = \{\#L\#\}$› **and**
‹$D' \subseteq\#$ D› **and**
‹*clause* ‘# $(N + U)$ + *NE* + *UE* $\models pm$ D'›
| *backtrack-nonunit-clause*:
‹*cdcl-twl-o* (M, N, U, *Some* D, *NE*, *UE*, $\{\#\}$, $\{\#\}$)
(*Propagated* L D' # *M1*, N, *add-mset* (*TWL-Clause* $\{\#L,\ L'\#\}$ $(D' - \{\#L,\ L'\#\})$) U, *None*, *NE*, *UE*,
$\{\#\}$, $\{\#-L\#\}$)›
if
‹$L \in\#$ D› **and**
‹(*Decided* K # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* M)› **and**

⟨$get\text{-}level\ M\ L = count\text{-}decided\ M$⟩ **and**
⟨$get\text{-}level\ M\ L = get\text{-}maximum\text{-}level\ M\ D'$⟩ **and**
⟨$get\text{-}maximum\text{-}level\ M\ (D' - \{\#L\#\}) \equiv i$⟩ **and**
⟨$get\text{-}level\ M\ K = i + 1$⟩
⟨$D' \neq \{\#L\#\}$⟩ **and**
⟨$D' \subseteq\#\ D$⟩ **and**
⟨$clause\ `\#\ (N + U) + NE + UE \models pm\ D'$⟩ **and**
⟨$L \in\#\ D'$⟩
⟨$L' \in\#\ D'$⟩ **and** — L' is the new watched literal
⟨$get\text{-}level\ M\ L' = i$⟩

inductive-cases *cdcl-twl-oE*: ⟨*cdcl-twl-o S T*⟩

inductive *cdcl-twl-stgy* :: ⟨$'v\ twl\text{-}st \Rightarrow\ 'v\ twl\text{-}st \Rightarrow bool$⟩ **for** S :: ⟨$'v\ twl\text{-}st$⟩ **where**
cp: ⟨$cdcl\text{-}twl\text{-}cp\ S\ S' \Longrightarrow cdcl\text{-}twl\text{-}stgy\ S\ S'$⟩ |
other': ⟨$cdcl\text{-}twl\text{-}o\ S\ S' \Longrightarrow cdcl\text{-}twl\text{-}stgy\ S\ S'$⟩

inductive-cases *cdcl-twl-stgyE*: ⟨*cdcl-twl-stgy S T*⟩

1.1.2 Definition of the Two-watched Literals Invariants

Definitions

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

primrec *struct-wf-twl-cls* :: ⟨$'v\ multiset\ twl\text{-}clause \Rightarrow bool$⟩ **where**
⟨$struct\text{-}wf\text{-}twl\text{-}cls\ (TWL\text{-}Clause\ W\ UW) \longleftrightarrow$
  $size\ W = 2 \land distinct\text{-}mset\ (W + UW)$⟩

fun $state_W\text{-}of$ :: ⟨$'v\ twl\text{-}st \Rightarrow\ 'v\ cdcl_W\text{-}restart\text{-}mset$⟩ **where**
⟨$state_W\text{-}of\ (M, N, U, C, NE, UE, Q) =$
  $(M, clause\ `\#\ N + NE, clause\ `\#\ U + UE,\ C)$⟩

named-theorems *twl-st* ⟨*Conversions simp rules*⟩

lemma [*twl-st*]: ⟨$trail\ (state_W\text{-}of\ S') = get\text{-}trail\ S'$⟩
  by (*cases S'*) (*auto simp*: *trail.simps*)

lemma [*twl-st*]:
  ⟨$get\text{-}trail\ S' \neq [] \Longrightarrow cdcl_W\text{-}restart\text{-}mset.hd\text{-}trail\ (state_W\text{-}of\ S') = hd\ (get\text{-}trail\ S')$⟩
  by (*cases S'*) (*auto simp*: *trail.simps*)

lemma [*twl-st*]: ⟨$conflicting\ (state_W\text{-}of\ S') = get\text{-}conflict\ S'$⟩
  by (*cases S'*) (*auto simp*: *conflicting.simps*)

The invariant on the clauses is the following:

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

definition *twl-is-an-exception* :: ⟨$'a\ multiset\ twl\text{-}clause \Rightarrow\ 'a\ multiset \Rightarrow$
  $('b \times\ 'a\ multiset\ twl\text{-}clause)\ multiset \Rightarrow bool$⟩
where

‹*twl-is-an-exception* C Q WS $\longleftrightarrow$
$(\exists L.\ L \in\#\ Q \wedge L \in\#\ \textit{watched}\ C) \vee (\exists L.\ (L,\ C) \in\#\ WS)$›

definition *is-blit* :: ‹$('a,\ 'b)$ *ann-lits* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool*›**where**
[*simp*]: ‹*is-blit* M D L $\longleftrightarrow$ $(L \in\#\ D \wedge L \in \textit{lits-of-l}\ M)$›

definition *has-blit* :: ‹$('a,\ 'b)$ *ann-lits* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ $'a$ *literal* $\Rightarrow$ *bool*›**where**
‹*has-blit* M D L' $\longleftrightarrow$ $(\exists L.\ \textit{is-blit}\ M\ D\ L \wedge \textit{get-level}\ M\ L \leq \textit{get-level}\ M\ L')$›

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

fun *twl-lazy-update* :: ‹$('a,\ 'b)$ *ann-lits* $\Rightarrow$ $'a$ *twl-cls* $\Rightarrow$ *bool*› **where**
‹*twl-lazy-update* M $(\textit{TWL-Clause}\ W\ UW)$ $\longleftrightarrow$
$(\forall L.\ L \in\#\ W \longrightarrow -L \in \textit{lits-of-l}\ M \longrightarrow \neg\textit{has-blit}\ M\ (W{+}UW)\ L \longrightarrow$
$(\forall K \in\#\ UW.\ \textit{get-level}\ M\ L \geq \textit{get-level}\ M\ K \wedge -K \in \textit{lits-of-l}\ M))$›

If one watched literals has been assigned to false ($-\ L \in \textit{lits-of-l}\ M$) and the clause has not yet been updated ($L' \notin \textit{lits-of-l}\ M$: it should be removed either by updating L, propagating L', or marking the conflict), then the literals L is of maximal level.

fun *watched-literals-false-of-max-level* :: ‹$('a,\ 'b)$ *ann-lits* $\Rightarrow$ $'a$ *twl-cls* $\Rightarrow$ *bool*› **where**
‹*watched-literals-false-of-max-level* M $(\textit{TWL-Clause}\ W\ UW)$ $\longleftrightarrow$
$(\forall L.\ L \in\#\ W \longrightarrow -L \in \textit{lits-of-l}\ M \longrightarrow \neg\textit{has-blit}\ M\ (W{+}UW)\ L \longrightarrow$
$\textit{get-level}\ M\ L = \textit{count-decided}\ M)$›

This invariants talks about the enqueued literals:

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

fun *no-duplicate-queued* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*no-duplicate-queued* $(M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q)$ $\longleftrightarrow$
$(\forall C\ C'.\ C \in\#\ WS \longrightarrow C' \in\#\ WS \longrightarrow \textit{fst}\ C = \textit{fst}\ C') \wedge$
$(\forall C.\ C \in\#\ WS \longrightarrow \textit{add-mset}\ (\textit{fst}\ C)\ Q \subseteq\#\ \textit{uminus}\ `\#\ \textit{lit-of}\ `\#\ \textit{mset}\ M) \wedge$
$Q \subseteq\#\ \textit{uminus}\ `\#\ \textit{lit-of}\ `\#\ \textit{mset}\ M$›

lemma *no-duplicate-queued-alt-def*:
‹*no-duplicate-queued* S $=$
$((\forall C\ C'.\ C \in\#\ \textit{clauses-to-update}\ S \longrightarrow C' \in\#\ \textit{clauses-to-update}\ S \longrightarrow \textit{fst}\ C = \textit{fst}\ C') \wedge$
$(\forall C.\ C \in\#\ \textit{clauses-to-update}\ S \longrightarrow$
$\textit{add-mset}\ (\textit{fst}\ C)\ (\textit{literals-to-update}\ S) \subseteq\#\ \textit{uminus}\ `\#\ \textit{lit-of}\ `\#\ \textit{mset}\ (\textit{get-trail}\ S)) \wedge$
$\textit{literals-to-update}\ S \subseteq\#\ \textit{uminus}\ `\#\ \textit{lit-of}\ `\#\ \textit{mset}\ (\textit{get-trail}\ S))$›
by (*cases* S) *auto*

fun *distinct-queued* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*distinct-queued* $(M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q)$ $\longleftrightarrow$
$\textit{distinct-mset}\ Q \wedge$
$(\forall L\ C.\ \textit{count}\ WS\ (L,\ C) \leq \textit{count}\ (N + U)\ C)$›

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

fun *clauses-to-update-prop* **where**

‹*clauses-to-update-prop* Q M $(L, C) \longleftrightarrow$
$(L \in\# \textit{ watched } C \land -L \in \textit{lits-of-l } M \land L \notin\# Q \land \neg\textit{has-blit } M\ (\textit{clause } C)\ L)$›
declare *clauses-to-update-prop.simps*[*simp del*]

This invariants talks about the enqueued literals:

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

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

fun *clauses-to-update-inv* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*clauses-to-update-inv* $(M, N, U, \textit{None}, NE, UE, WS, Q) \longleftrightarrow$
$(\forall L\ C.\ ((L, C) \in\# WS \longrightarrow \{\#(L, C)|\ C \in\# N + U.\ \textit{clauses-to-update-prop } Q\ M\ (L, C)\#\} \subseteq\# WS)) \land$
$(\forall L.\ WS = \{\#\} \longrightarrow \{\#(L, C)|\ C \in\# N + U.\ \textit{clauses-to-update-prop } Q\ M\ (L, C)\#\} = \{\#\}) \land$
$(\forall L\ C.\ C \in\# N + U \longrightarrow L \in\# \textit{watched } C \longrightarrow -L \in \textit{lits-of-l } M \longrightarrow \neg\textit{has-blit } M\ (\textit{clause } C)\ L \longrightarrow$
$(L, C) \notin\# WS \longrightarrow L \in\# Q)$›
| ‹*clauses-to-update-inv* $(M, N, U, D, NE, UE, WS, Q) \longleftrightarrow \textit{True}$›

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

fun *twl-exception-inv* :: ‹$'v$ *twl-st* $\Rightarrow$ $'v$ *twl-cls* $\Rightarrow$ *bool*› **where**
‹*twl-exception-inv* $(M, N, U, \textit{None}, NE, UE, WS, Q)\ C \longleftrightarrow$
$(\forall L.\ L \in\# \textit{watched } C \longrightarrow -L \in \textit{lits-of-l } M \longrightarrow \neg\textit{has-blit } M\ (\textit{clause } C)\ L \longrightarrow$
$L \notin\# Q \longrightarrow (L, C) \notin\# WS \longrightarrow$
$(\forall K \in\# \textit{unwatched } C.\ -K \in \textit{lits-of-l } M))$›
| ‹*twl-exception-inv* $(M, N, U, D, NE, UE, WS, Q)\ C \longleftrightarrow \textit{True}$›

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

fun *twl-st-exception-inv* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*twl-st-exception-inv* $(M, N, U, D, NE, UE, WS, Q) \longleftrightarrow$
$(\forall C \in\# N + U.\ \textit{twl-exception-inv } (M, N, U, D, NE, UE, WS, Q)\ C)$›

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

fun *propa-cands-enqueued* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*propa-cands-enqueued* $(M, N, U, \textit{None}, NE, UE, WS, Q) \longleftrightarrow$
$(\forall L\ C.\ C \in\# N+U \longrightarrow L \in\# \textit{clause } C \longrightarrow M \models\!as\ \textit{CNot } (\textit{remove1-mset } L\ (\textit{clause } C)) \longrightarrow$
$\textit{undefined-lit } M\ L \longrightarrow$
$(\exists L'.\ L' \in\# \textit{watched } C \land L' \in\# Q) \lor (\exists L.\ (L, C) \in\# WS))$›
| ‹*propa-cands-enqueued* $(M, N, U, D, NE, UE, WS, Q) \longleftrightarrow \textit{True}$›

fun *confl-cands-enqueued* :: ‹$'v$ *twl-st* $\Rightarrow$ *bool*› **where**
‹*confl-cands-enqueued* $(M, N, U, \textit{None}, NE, UE, WS, Q) \longleftrightarrow$
$(\forall C \in\# N + U.\ M \models\!as\ \textit{CNot } (\textit{clause } C) \longrightarrow$
$(\exists L'.\ L' \in\# \textit{watched } C \land L' \in\# Q) \lor (\exists L.\ (L, C) \in\# WS))$›
| ‹*confl-cands-enqueued* $(M, N, U, \textit{Some -}, NE, UE, WS, Q) \longleftrightarrow$
True›

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

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

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

lemma *twl-st-inv-alt-def*:
⟨twl-st-inv S ⟷
(∀ C ∈# get-clauses S. struct-wf-twl-cls C) ∧
(∀ C ∈# get-clauses S. get-conflict S = None ⟶
¬twl-is-an-exception C (literals-to-update S) (clauses-to-update S) ⟶
(twl-lazy-update (get-trail S) C)) ∧
(∀ C ∈# get-clauses S. get-conflict S = None ⟶
watched-literals-false-of-max-level (get-trail S) C)⟩
by (*cases S*) (*auto simp*: *twl-st-inv.simps*)

All the unit clauses are all propagated initially except when we have found a conflict of level *0*.

fun *entailed-clss-inv* :: ⟨'v twl-st ⇒ bool⟩ **where**
⟨entailed-clss-inv (M, N, U, D, NE, UE, WS, Q) ⟷
(∀ C ∈# NE + UE.
(∃ L. L ∈# C ∧ (D = None ∨ count-decided M > 0 ⟶ get-level M L = 0 ∧ L ∈ lits-of-l M)))⟩

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

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

Putting invariants together:

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

past-invs S)
⟩

definition *twl-stgy-invs* :: ⟨′*v twl-st* ⇒ *bool*⟩ **where**
⟨*twl-stgy-invs S* ⟷
$cdcl_W$*-restart-mset*.$cdcl_W$*-stgy-invariant* ($state_W$*-of S*) ∧
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0* ($state_W$*-of S*)⟩

Initial properties

lemma *twl-is-an-exception-add-mset-to-queue*: ⟨*twl-is-an-exception C* (*add-mset L Q*) *WS* ⟷
(*twl-is-an-exception C Q WS* ∨ (*L* ∈# *watched C*))⟩
unfolding *twl-is-an-exception-def* **by** *auto*

lemma *twl-is-an-exception-add-mset-to-clauses-to-update*:
⟨*twl-is-an-exception C Q* (*add-mset* (*L*, *D*) *WS*) ⟷ (*twl-is-an-exception C Q WS* ∨ *C* = *D*)⟩
unfolding *twl-is-an-exception-def* **by** *auto*

lemma *twl-is-an-exception-empty*[*simp*]: ⟨¬*twl-is-an-exception C* {#} {#}⟩
unfolding *twl-is-an-exception-def* **by** *auto*

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

lemma *clauses-to-update-inv-cases*[*case-names WS-nempty WS-empty Q*]:
assumes
⟨⋀*L C*. (*L*, *C*) ∈# *WS* ⟹ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*⟩ **and**
⟨⋀*L*. *WS* = {#} ⟹ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} = {#}⟩ **and**
⟨⋀*L C*. *C* ∈# *N* + *U* ⟹ *L* ∈# *watched C* ⟹ −*L* ∈ *lits-of-l M* ⟹ ¬*has-blit M* (*clause C*) *L* ⟹
(*L*, *C*) ∉# *WS* ⟹ *L* ∈# *Q*⟩
shows
⟨*clauses-to-update-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*)⟩
using *assms* **unfolding** *clauses-to-update-inv.simps* **by** *blast*

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

lemma
shows
no-duplicate-queued-no-queued: ⟨*no-duplicate-queued* (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, {#})⟩ **and**
no-distinct-queued-no-queued: ⟨*distinct-queued* ([], *N*, *U*, *D*, *NE*, *UE*, {#}, {#})⟩
by *auto*

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

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

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

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

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

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

  show ‹*twl-lazy-update* *M1* *C*›
    unfolding *C* *twl-lazy-update.simps*
  proof (*intro allI impI*)
    fix *L*
    assume

```
    W: ‹L ∈# W› and
    uL: ‹− L ∈ lits-of-l M1› and
    L': ‹¬has-blit M1 (W+UW) L›

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

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

show ‹∀ K∈#UW. get-level M1 K ≤ get-level M1 L ∧ −K ∈ lits-of-l M1›
proof clarify
  fix K''
  assume ‹K'' ∈# UW›
  then have
    lev-K'-L: ‹get-level M K'' ≤ get-level M L› and
    uK'-M: ‹−K'' ∈ lits-of-l M›
    using lazy W uL L'M unfolding C MM' by auto
  then have uK'-M1: ‹− K'' ∈ lits-of-l M1›
    using uK'-M unfolding M apply (auto simp: get-level-append-if
        split: if-splits)
    using M' MM' n-d uL count-decided-ge-get-level[of M1 L]
    by (auto dest: defined-lit-no-dupD in-lits-of-l-defined-litD
        simp: get-level-cons-if atm-of-eq-atm-of
        split: if-splits)
  have ‹get-level M K'' = get-level M1 K''›
  proof (rule ccontr, cases ‹defined-lit M' K''›)
    case False
    moreover assume ‹get-level M K'' ≠ get-level M1 K''›
    ultimately show False unfolding MM' by auto
  next
    case True
    assume K'': ‹get-level M K'' ≠ get-level M1 K''›
    have ‹get-level M' K'' = 0›
    proof −
      have a1: ‹get-level M' K'' + count-decided M1 ≤ get-level M1 L›
        using lev-K'-L unfolding lev-L-M1 unfolding MM' get-level-skip-end[OF True] .
```

then have ‹*count-decided M1* ≤ *get-level M1 L*›
by *linarith*
then have ‹*get-level M1 L* = *count-decided M1*›
using *count-decided-ge-get-level le-antisym* **by** *blast*
then show *?thesis*
using *a1* **by** *linarith*
qed
moreover have ‹*Decided K* ∈ *set* (*dropWhile* (λ*S*. *atm-of* (*lit-of S*) ≠ *atm-of K''*) *M'*)›
unfolding *M' append-assoc*[*symmetric*] **by** (*rule last-in-set-dropWhile*)
(*use True* **in** ‹*auto simp*: *lits-of-def M' MM' defined-lit-map*›)
ultimately show *False*
by (*auto simp*: *M' filter-empty-conv get-level-def*)
qed
then show ‹*get-level M1 K''* ≤ *get-level M1 L* ∧ −*K''* ∈ *lits-of-l M1*›
using *lev-M-M1*[*OF uL*] *lev-K'-L uK'-M uK'-M1* **by** *auto*
qed
qed
qed

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

lemma *has-blit-Cons*[*simp*]:
assumes *blit*: ‹*has-blit M C L*› **and** *n-d*: ‹*no-dup* (*K* # *M*)›
shows ‹*has-blit* (*K* # *M*) *C L*›
proof −
obtain *L'* **where**
‹*is-blit M C L'*› **and**
‹*get-level M L'* ≤ *get-level M L*›
using *blit* **unfolding** *has-blit-def* **by** *auto*
then have
‹*is-blit* (*K* # *M*) *C L'*› **and**
‹*get-level* (*K* # *M*) *L'* ≤ *get-level* (*K* # *M*) *L*›
using *n-d* **by** (*auto simp add*: *has-blit-def get-level-cons-if atm-of-eq-atm-of*
dest: *in-lits-of-l-defined-litD*)
then show *?thesis*
unfolding *has-blit-def* **by** *blast*
qed

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

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

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

by (*cases C*) *auto*

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

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

lemma *twl-lazy-update-Propagated*:
assumes
W: ‹*L* ∈# *W*› **and** *n-d*: ‹*no-dup* (*Propagated L D* # *M*)› **and**
lazy: ‹*twl-lazy-update M* (*TWL-Clause W UW*)›
shows
‹*twl-lazy-update* (*Propagated L D* # *M*) (*TWL-Clause W UW*)›
unfolding *twl-lazy-update.simps*
proof (*intro conjI impI allI*)
fix *La*
assume
La: ‹*La* ∈# *W*› **and**
uL-M: ‹− *La* ∈ *lits-of-l* (*Propagated L D* # *M*)› **and**
b: ‹¬ *has-blit* (*Propagated L D* # *M*) (*W* + *UW*) *La*›
have *b'*: ‹¬*has-blit M* (*W* + *UW*) *La*›
apply (*rule no-has-blit-propagate*[*OF b*])
using *assms* **by** *auto*

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

dest!: *multi-member-split*)
next
case *2*
have [*simp*]: ‹*has-blit* (*Propagated L D # M*) (*W* + *UW*) (−*L*)›
using *2 La W* **unfolding** *has-blit-def* **apply** −
by (*rule exI*[*of* - *L*])
(*auto simp*: *get-level-cons-if atm-of-eq-atm-of*)
show *?thesis*
using *2 b count-decided-ge-get-level*[*of* ‹*Propagated L D # M*›]
by (*auto simp*: *uminus-lit-swap split*: *if-splits*)
qed
qed

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

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

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

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

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

lemma
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**

valid: ⟨*valid-enqueued S*⟩ **and**
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)⟩ **and**
no-dup: ⟨*no-duplicate-queued S*⟩ **and**
dist-q: ⟨*distinct-queued S*⟩ **and**
ws: ⟨*clauses-to-update-inv S*⟩
shows *twl-cp-twl-st-exception-inv*: ⟨*twl-st-exception-inv T*⟩ **and**
twl-cp-clauses-to-update: ⟨*clauses-to-update-inv T*⟩
using *cdcl twl twl-excep valid inv no-dup ws*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE L Q*)
case *1* **note** - = *this*(*2*)
then show *?case* **unfolding** *twl-st-inv.simps twl-is-an-exception-def*
by (*fastforce simp add*: *pair-in-image-Pair image-constant-conv uminus-lit-swap twl-exception-inv.simps*)
case *2* **note** *twl* = *this*(*1*) **and** *ws* = *this*(*6*)
have *struct*: ⟨*struct-wf-twl-cls C*⟩ **if** ⟨$C \in\# N + U$⟩ **for** *C*
using *twl that* **by** (*simp add*: *twl-st-inv.simps*)
have *H*: ⟨*count* (*watched C*) $L \le 1$⟩ **if** ⟨$C \in\# N + U$⟩ **for** *C L*
using *struct*[*OF that*] **by** (*cases C*) (*auto simp add*: *twl-st-inv.simps size-2-iff*)
have *sum-le-count*: ⟨($\sum x\in\#N+U.$ *count* $\{\#(L, x).\ L \in\#$ *watched* $x\#\}$ $(a, b)) \le$ *count* $(N+U)$ b⟩
for *a b*
apply (*subst* (*2*) *count-sum-mset-if-1-0*)
apply (*rule sum-mset-mono*)
using *H* **apply** (*auto simp*: *count-image-mset-Pair2*)
done
define *NU* **where** *NU*[*symmetric*]: ⟨$NU = N + U$⟩
show *?case*
using *ws* **by** (*fastforce simp add*: *pair-in-image-Pair multiset-filter-mono2 image-Pair-subset-mset clauses-to-update-prop.simps NU filter-mset-empty-conv*)
next
case (*propagate D L L′ M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *undef* = *this*(*2*) **and** *unw* = *this*(*3*)

case *1*
note *twl* = *this*(*1*) **and** *twl-excep* = *this*(*2*) **and** *valid* = *this*(*3*) **and** *inv* = *this*(*4*) **and** *no-dup* = *this*(*5*) **and** *ws* = *this*(*6*)
have [*simp*]: ⟨$- L' \notin$ *lits-of-l M*⟩
using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *blast*
have *D-N-U*: ⟨$D \in\# N + U$⟩ **and** *lev-L*: ⟨*get-level M L* = *count-decided M*⟩
using *valid* **by** *auto*
then have *wf-D*: ⟨*struct-wf-twl-cls D*⟩
using *twl* **by** (*simp add*: *twl-st-inv.simps*)
have ⟨$\forall s\in\#$*clause* ‘# $U.\ \neg$ *tautology s*⟩
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* **by** (*simp-all add*: $cdcl_W$*-restart-mset-state*)
have *n-d*: ⟨*no-dup M*⟩
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
have [*simp*]: ⟨$L \neq L'$⟩
using *wf-D watched* **by** (*cases D*) *auto*
have [*simp*]: ⟨$- L \in$ *lits-of-l M*⟩
using *valid* **by** *auto*
then have [*simp*]: ⟨$L \notin$ *lits-of-l M*⟩
using *n-d no-dup-consistentD* **by** *blast*
obtain *NU* **where** *NU*: ⟨$N + U$ = *add-mset D NU*⟩
by (*metis D-N-U insert-DiffM*)

have [*simp*]: ‹*has-blit* (*Propagated* L' (*add-mset* L (*add-mset* L' *x2*)) # M)
(*add-mset* L (*add-mset* L' *x2*)) L› **for** *x2*
unfolding *has-blit-def*
by (*rule exI*[*of* - L'])
(*use lev-L* **in** ‹*auto simp*: *get-level-cons-if*›)
have *HH*: ‹¬*clauses-to-update-prop* (*add-mset* ($-L'$) Q) (*Propagated* L' (*clause* D) # M) (L, D)›
using *watched* **unfolding** *clauses-to-update-prop.simps* **by** (*cases* D) (*auto simp*: *watched*)
have ‹*add-mset* L Q ⊆# {#− *lit-of* x. x ∈# *mset* M#}›
using *no-dup* **by** (*auto*)
moreover have ‹*distinct-mset* {#− *lit-of* x. x ∈# *mset* M#}›
by (*subst distinct-image-mset-inj*)
(*use n-d* **in** ‹*auto simp*: *lit-of-inj-on-no-dup distinct-map no-dup-def*›)
ultimately have [*simp*]: ‹L ∉# Q›
by (*metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add*)
have ‹¬*has-blit* M (*clause* D) L›
using *watched undef unw n-d* **by** (*cases* D)
(*auto simp*: *has-blit-def Decided-Propagated-in-iff-in-lits-of-l dest*: *no-dup-consistentD*)
then have *w-q-p-D*: ‹*clauses-to-update-prop* Q M (L, D)›
by (*auto simp*: *clauses-to-update-prop.simps watched*)
have ‹*Pair* L ‘# {#C ∈# *add-mset* D *NU*. *clauses-to-update-prop* Q M (L, C)#} ⊆# *add-mset* (L, D) *WS*›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
then have *IH*: ‹*Pair* L ‘# {#C ∈# *NU*. *clauses-to-update-prop* Q M (L, C)#} ⊆# *WS*›
using *w-q-p-D* **by** *auto*
have *IH-Q*: ‹∀ *La* C. C ∈# *add-mset* D *NU* ⟶ *La* ∈# *watched* C ⟶ − *La* ∈ *lits-of-l* M ⟶
¬ *has-blit* M (*clause* C) *La* ⟶ (*La*, C) ∉# *add-mset* (L, D) *WS* ⟶ *La* ∈# Q›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)

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

moreover have *CD*: *False* **if** ‹C = D›
using J J' *watched-C watched that J-notin*
by (*cases* D) (*auto simp*: *add-mset-eq-add-mset*)
ultimately show ‹− K ∈ *lits-of-l* (*Propagated* L' (*clause* D) # M)›
by *blast*
qed
case *2*
show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty* L'' C)

then have [*simp*]: ‹$L'' = L$›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU* **by** (*auto simp*: *all-conj-distrib*)

have ∗: ‹*Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊇#
Pair L ‘# {#*C* ∈# *NU*.
clauses-to-update-prop (*add-mset* (− *L*′) *Q*) (*Propagated L*′ (*clause D*) # *M*) (*L*′′, *C*)#}›
using *undef n-d*
unfolding *image-Pair-subset-mset multiset-filter-mono2 clauses-to-update-prop.simps*
by (*auto dest*!: *no-has-blit-propagate*′)
show *?case*
using *subset-mset.dual-order.trans*[*OF IH* ∗] *HH*
unfolding *NU* ‹$L'' = L$›
by *simp*
next
case (*WS-empty K*)
then show *?case*
using *IH IH-Q watched undef n-d* **unfolding** *NU*
by (*cases D*) (*auto simp*: *filter-mset-empty-conv*
clauses-to-update-prop.simps watched add-mset-eq-add-mset
dest!: *no-has-blit-propagate*′)
next
case (*Q LC*′ *C*)
then show *?case*
using *watched 1.prems*(*6*) *HH Q.hyps HH IH-Q undef n-d*
apply (*cases D*)
apply (*cases C*)
apply (*auto simp*: *add-mset-eq-add-mset NU*)
by (*metis HH Q.IH*(*2*) *Q.IH*(*3*) *Q.hyps clauses-to-update-prop.simps insert-iff*
no-has-blit-propagate′ *set-mset-add-mset-insert*)
qed
next
case (*conflict D L L*′ *M N U NE UE WS Q*)
case *1*
note *twl* = *this*(*5*)
show *?case* **by** (*auto simp*: *twl-st-inv.simps twl-exception-inv.simps*)

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

case *1* **note** *twl* = *this*(*1*) **and** *twl-excep* = *this*(*2*) **and** *valid* = *this*(*3*) **and** *inv* = *this*(*4*) **and**
no-dup = *this*(*5*) **and** *ws* = *this*(*6*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
have *D-N-U*: ‹*D* ∈# *N* + *U*›
using *valid* **by** *auto*
then have *wf-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*simp add*: *twl-st-inv.simps*)
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
have *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
have [*simp*]: ‹*has-blit M* (*clause D*) *L*›

unfolding *has-blit-def*
by (*rule exI*[*of - L'*])
(*use watched L' lev-L* **in** ‹*auto simp*: *count-decided-ge-get-level*›)
have [*simp*]: ‹¬*clauses-to-update-prop Q M* (*L*, *D*)›
using *L'* **by** (*auto simp*: *clauses-to-update-prop.simps watched*)
have *IH-WS*: ‹*Pair L* ‘# {#*C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*, *D*) *WS*›
using *ws* **by** (*auto simp del*: *filter-union-mset simp*: *NU*)
then have *IH-WS-NU*: ‹*Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*, *D*) *WS*›
using *ws* **by** (*auto simp del*: *filter-union-mset simp*: *NU*)

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

show *?case*
unfolding *Ball-def twl-st-exception-inv.simps twl-exception-inv.simps*
proof (*intro allI conjI impI*)
fix *C J K*
assume *C*: ‹*C* ∈# *N* + *U*› **and**
watched-C: ‹*J* ∈# *watched C*› **and**
J: ‹− *J* ∈ *lits-of-l M*› **and**
J': ‹¬*has-blit M* (*clause C*) *J*› **and**
J-notin: ‹*J* ∉# *Q*› **and**
C-WS: ‹(*J*, *C*) ∉# *WS*› **and**
‹*K* ∈# *unwatched C*›
then have ‹− *K* ∈ *lits-of-l M*› **if** ‹*C* ≠ *D*›
using *twl-excep that* **by** (*simp add*: *uminus-lit-swap twl-exception-inv.simps*)

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

case *2*
show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty K C*) **note** *KC* = *this*
have *LK*: ‹*L* = *K*›
using *no-dup KC* **by** *auto*
from *subset-add-mset-notin-subset-mset*[*OF IH-WS*]
have *1*: ‹*Pair K* ‘# {#*C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
using *L' LK* ‹*has-blit M* (*clause D*) *L*›
by (*auto simp del*: *filter-union-mset simp*: *pair-in-image-Pair watched add-mset-eq-add-mset*

all-conj-distrib clauses-to-update-prop.simps)
show *?case*
by (*metis* (*no-types*, *lifting*) *1 LK*)
next
case (*WS-empty K*) **note** [*simp*] = *this*(*1*)
have [*simp*]: ‹¬*clauses-to-update-prop Q M* (*K*, *D*)›
using *IH-Q WS-empty.IH watched* ‹*has-blit M* (*clause D*) *L*›
using *IH-WS′ IH-Q watched* **by** (*auto simp*: *add-mset-eq-add-mset NU filter-mset-empty-conv*
all-conj-distrib clauses-to-update-prop.simps)
show *?case*
using *IH-WS′ IH-Q watched* **by** (*auto simp*: *add-mset-eq-add-mset NU filter-mset-empty-conv*
all-conj-distrib clauses-to-update-prop.simps)
next
case (*Q K C*)
then show *?case*
using ‹¬ *clauses-to-update-prop Q M* (*L*, *D*)› *ws*
unfolding *clauses-to-update-inv.simps*(*1*) *clauses-to-update-prop.simps member-add-mset*
is-blit-def
by *blast*
qed
next
case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*)
and
L′ = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N′U′* = *this*(*6*)

case *1* **note** *twl* = *this*(*1*) **and** *twl-excep* = *this*(*2*) **and** *valid* = *this*(*3*) **and** *inv* = *this*(*4*) **and**
no-dup = *this*(*5*) **and** *ws* = *this*(*6*)
obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided*
M›
using *valid* **by** *auto*
then have *struct-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *L′-UWD*: ‹*L* ∉# *remove1-mset L′ UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD L* ≥ *1*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *L* ≥ *2*›
using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have *L′-L′-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD K* ≥ *2*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *K* ≥ *2*›
using *D L′* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)

moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N′* + *U′*› **for** *C*
using *C N′U′ that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *trail.simps*)
then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
distinct-consistent-interp **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
have *L-M*: ‹*L* ∉ *lits-of-l M*›
using *n-d uL* **by** (*fastforce dest*!: *distinct-consistent-interp*
simp: *consistent-interp-def lits-of-def uminus-lit-swap*)
have *w-max-D*: ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *lev-L′*: ‹*get-level M L′* = *count-decided M*›
if ‹− *L′* ∈ *lits-of-l M*› ‹¬*has-blit M* (*clause D*) *L′*›
using *L-M w-max-D D watched L′ uL that* **by** *auto*
have *D-ne-D*: ‹*D* ≠ *update-clause D L K*›
using *D add-remove-WD* **by** *auto*
have *N′U′*: ‹*N′* + *U′* = *add-mset ?D* (*remove1-mset D* (*N* + *U*))›
using *N′U′ D-N-U* **by** (*auto elim*!: *update-clausesE*)
define *NU* **where** ‹*NU* = *remove1-mset D* (*N* + *U*)›
then have *NU*: ‹*N* + *U* = *add-mset D NU*›
using *D-N-U* **by** *auto*
have *watched-D*: ‹*watched ?D* = {#*K*, *L′*#}›
using *D add-remove-WD watched* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
have *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
have ‹*has-blit* (*Propagated L′ C* # *M*)
(*add-mset L* (*add-mset L′ x2*)) *L*› **for** *C x2*
unfolding *has-blit-def*
by (*rule exI*[*of - L′*])
(*use lev-L* **in** ‹*auto simp*: *count-decided-ge-get-level get-level-cons-if*›)
then have *HH*: ‹¬*clauses-to-update-prop* (*add-mset* (−*L′*) *Q*) (*Propagated L′* (*clause D*) # *M*) (*L*, *D*)›
using *watched* **unfolding** *clauses-to-update-prop.simps* **by** (*cases D*) (*auto simp*: *watched*)
have ‹*add-mset L Q* ⊆# {#− *lit-of x*. *x* ∈# *mset M*#}›
using *no-dup* **by** (*auto*)
moreover have ‹*distinct-mset* {#− *lit-of x*. *x* ∈# *mset M*#}›
by (*subst distinct-image-mset-inj*)
(*use n-d* **in** ‹*auto simp*: *lit-of-inj-on-no-dup distinct-map no-dup-def*›)

ultimately have *LQ*: ‹$L \notin\# Q$›
by (*metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add*)
have *w-q-p-D*: ‹¬*has-blit M* (*clause D*) $L \Longrightarrow$ *clauses-to-update-prop Q M* (*L*, *D*)›
using *watched uL L′* **by** (*cases D*) (*auto simp*: *LQ clauses-to-update-prop.simps*)
have ‹*Pair L* ‘# {#*C* ∈# *add-mset D NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*, *D*) *WS*›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
then have *IH*: ‹¬*has-blit M* (*clause D*) $L \Longrightarrow$ *Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
using *w-q-p-D* **by** *auto*
have *IH-Q*: ‹⋀*La C*. *C* ∈# *add-mset D NU* ⟹ *La* ∈# *watched C* ⟹ − *La* ∈ *lits-of-l M* ⟹
¬*has-blit M* (*clause C*) *La* ⟹ (*La*, *C*) ∉# *add-mset* (*L*, *D*) *WS* ⟹ *La* ∈# *Q*›
using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
have *blit-clss-to-upd*: ‹*has-blit M* (*clause D*) *L* ⟹ ¬ *clauses-to-update-prop Q M* (*L*, *D*)›
by (*auto simp*: *clauses-to-update-prop.simps*)
have
‹*Pair L* ‘# {#*C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*, *D*) *WS*›
using *ws* **by** (*auto simp del*: *filter-union-mset*)
moreover have ‹*has-blit M* (*clause D*) *L* ⟹
(*L*, *D*) ∉# *Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#}›
by (*auto simp*: *clauses-to-update-prop.simps*)
ultimately have *Q-M-L-WS*:
‹*Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
by (*auto simp del*: *filter-union-mset simp*: *NU w-q-p-D blit-clss-to-upd*
intro: *subset-add-mset-notin-subset-mset split*: *if-splits*)
have *L-ne-L′*: ‹*L* ≠ *L′*›
using *struct-D D watched* **by** *auto*
have *clss-upd-D*[*simp*]: ‹*clause ?D* = *clause D*›
using *D K watched* **by** *auto*
show *?case*
unfolding *Ball-def twl-st-exception-inv.simps twl-exception-inv.simps*
proof (*intro allI conjI impI*)
fix *C J K″*
assume *C*: ‹*C* ∈# *N′* + *U′*› **and**
watched-C: ‹*J*∈# *watched C*› **and**
J: ‹− *J* ∈ *lits-of-l M*› **and**
J′: ‹¬*has-blit M* (*clause C*) *J*› **and**
J-notin: ‹*J* ∉# *Q*› **and**
C-WS: ‹(*J*, *C*) ∉# *WS*› **and**
K″: ‹*K″* ∈# *unwatched C*›
then have ‹− *K″* ∈ *lits-of-l M*› **if** ‹*C* ≠ *D*› ‹*C* ≠ *?D*›
using *twl-excep that* ∗[*OF* - *C*] *N′U′* **by** (*simp add*: *uminus-lit-swap twl-exception-inv.simps*)
moreover have ‹− *K″* ∈ *lits-of-l M*› **if** *CD*: ‹*C* = *D*›
proof (*rule ccontr*)
assume *uK″-M*: ‹− *K″* ∉ *lits-of-l M*›
have ‹*Pair L* ‘# {#*C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *add-mset* (*L*, *D*) *WS*›
using *ws* **by** (*auto simp*: *all-conj-distrib*
simp del: *filter-union-mset*)
show *False*
proof *cases*
assume [*simp*]: ‹*J* = *L*›
have *w-q-p-L*: ‹*clauses-to-update-prop Q M* (*L*, *C*)›
unfolding *clauses-to-update-prop.simps watched-C J J′ K″ uK″-M*

apply (*auto simp add*: *add-mset-eq-add-mset conj-disj-distribR ex-disj-distrib*)
using *watched watched-C CD J J′ J-notin K″ uK″-M uL L′ L-M*
by (*auto simp*: *clauses-to-update-prop.simps add-mset-eq-add-mset*)
then have ‹*Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
using *ws* **by** (*auto simp*: *all-conj-distrib NU CD simp del*: *filter-union-mset*)
moreover have ‹(*L*, *C*) ∈# *Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#}›
using *C w-q-p-L D-ne-D* **by** (*auto simp*: *pair-in-image-Pair N′U′ NU CD*)
ultimately have ‹(*L*, *C*) ∈# *WS*›
by *blast*
then show ‹*False*›
using *C-WS* **by** *simp*
next
assume ‹*J* ≠ *L*›
then have ‹*clauses-to-update-prop Q M* (*L*, *C*)›
unfolding *clauses-to-update-prop.simps watched-C J J′ K″ uK″-M*
apply (*auto simp add*: *add-mset-eq-add-mset conj-disj-distribR ex-disj-distrib*)
using *watched watched-C CD J J′ J-notin K″ uK″-M uL L′ L-M*
apply (*auto simp*: *clauses-to-update-prop.simps add-mset-eq-add-mset*)
using *C-WS D-N-U clauses-to-update-prop.simps ws* **by** *auto*
then show ‹*False*›
using *C-WS D-N-U J J′ J-notin* ‹*J* ≠ *L*› *that watched-C ws* **by** *auto*
qed
qed
moreover {
assume *CD*: ‹*C* = *?D*›
have *JL*[*simp*]: ‹*J* = *L′*›
using *CD J J′ watched-C watched L′ D uK-M undef*
by (*auto simp*: *add-mset-eq-add-mset*)
have ‹*K″* ≠ *K*›
using *K″ uK-M uL D L′-L′-UWD* **unfolding** *CD*
by (*cases D*) *auto*
have *K″-unwatched-L*: ‹*K″* ∈# *remove1-mset K* (*unwatched D*) ∨ *K″* = *L*›
using *K″* **unfolding** *CD* **by** (*cases D*) *auto*
have ‹*clause C* = *clause D*›
using *D K watched* **unfolding** *CD* **by** *auto*
then have *blit*: ‹¬ *has-blit M* (*clause D*) *L′*›
using *J′* **unfolding** *CD* **by** *simp*
have *False* **if** ‹− *L′* ∈ *lits-of-l M*› ‹*L′* ∈ *lits-of-l M*›
using *n-d that consistent-interp-def distinct-consistent-interp* **by** *blast*
have *H*: ‹⋀*x La xa*. *x* ∈# *N* + *U* ⟹
La ∈# *watched x* ⟹ − *La* ∈ *lits-of-l M* ⟹
¬*has-blit M* (*clause x*) *La* ⟹ *La* ∉# *Q* ⟹ (*La*, *x*) ∉# *add-mset* (*L*, *D*) *WS* ⟹
xa ∈# *unwatched x* ⟹ − *xa* ∈ *lits-of-l M*›
using *twl-excep*[*unfolded twl-st-exception-inv.simps Ball-def twl-exception-inv.simps*]
unfolding *has-blit-def is-blit-def*
by *blast*
have *LL′*: ‹*L* ≠ *L′*›
using *struct-D watched* **by** (*cases D*) *auto*
have *L′D-WS*: ‹(*L′*, *D*) ∉# *WS*›
using *no-dup LL′* **by** (*auto dest*: *multi-member-split*)
have ‹*xa* ∈# *unwatched D* ⟹ − *xa* ∈ *lits-of-l M*›
if ‹− *L′* ∈ *lits-of-l M*› **and** ‹*L′* ∉# *Q*› **and** ‹¬ *has-blit M* (*clause D*) *L′*› **for** *xa*
by (*rule H*[*of D L′*])
(*use D-N-U watched LL′ that L′D-WS K″ that* **in** ‹*auto simp*: *add-mset-eq-add-mset L-M*›)
consider
(*unwatched-unqueued*) ‹*K″* ∈# *remove1-mset K* (*unwatched D*)› |

(*KL*) ⟨*K″ = L*⟩
using *K″-unwatched-L* **by** *blast*
then have ⟨− *K″* ∈ *lits-of-l M*⟩
proof *cases*
case *KL*
then show *?thesis*
using *uL* **by** *simp*
next
case *unwatched-unqueued*
moreover have ⟨*L′* ∉# *Q*⟩
using *JL J-notin* **by** *blast*
ultimately show *?thesis*
using *blit H*[*of D L′*] *D-N-U watched LL′ L′D-WS K″ J J′*
by (*auto simp*: *add-mset-eq-add-mset L-M dest*: *in-diffD*)
qed
}
ultimately show ⟨− *K″* ∈ *lits-of-l M*⟩
by *blast*
qed

case *2*
show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty K″ C*) **note** *KC* = *this*(*1*)
have *LK*: ⟨*L* = *K″*⟩
using *no-dup KC* **by** *auto*
have [*simp*]: ⟨¬*clauses-to-update-prop Q M* (*K″*, *update-clause D K″ K*)⟩
using *watched uK-M struct-D*
by (*cases D*) (*auto simp*: *clauses-to-update-prop.simps add-mset-eq-add-mset LK*)
have *1*: ⟨*Pair L* ‘# {#*C* ∈# *N′* + *U′*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆#
Pair L ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#}⟩
unfolding *image-Pair-subset-mset LK*
using *LK N′U′* **by** (*auto simp del*: *filter-union-mset simp*: *pair-in-image-Pair watched NU*
add-mset-eq-add-mset all-conj-distrib)
then show ⟨*Pair K″* ‘# {#*C* ∈# *N′* + *U′*. *clauses-to-update-prop Q M* (*K″*, *C*)#} ⊆# *WS*⟩
using *Q-M-L-WS* **unfolding** *LK* **by** *auto*
next
case (*WS-empty K″*)
then show *?case*
using *IH IH-Q uL uK-M L-M watched L-ne-L′* **unfolding** *N′U′ NU*
by (*force simp*: *filter-mset-empty-conv clauses-to-update-prop.simps*
add-mset-eq-add-mset watched-D all-conj-distrib)
next
case (*Q K′ C*) **note** *C* = *this*(*1*) **and** *uK′-M* = *this*(*2*) **and** *uK″-M* = *this*(*3*) **and** *KC-WS* =
this(*4*)
and *watched-C* = *this*(*5*)
have *?case* **if** *CD*: ⟨*C* ≠ *D*⟩ ⟨*C* ≠ *?D*⟩
using *IH-Q*[*of C K′*] *CD watched uK-M L′ L-ne-L′ L-M uK′-M uK″-M*
Q **unfolding** *N′U′ NU*
by *auto*
moreover have *?case* **if** *CD*: ⟨*C* = *D*⟩
proof −
consider
(*KL*) ⟨*K′* = *L*⟩ |
(*K′L′*) ⟨*K′* = *L′*⟩
using *watched watched-C CD* **by** (*auto simp*: *add-mset-eq-add-mset*)

then show *?thesis*
proof *cases*
case *KL* **note** [*simp*] = *this*
have ‹(*L*, *C*) ∈# *Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#}›
using *CD C w-q-p-D uK′′-M* **unfolding** *NU N′U′* **by** (*auto simp*: *pair-in-image-Pair D-ne-D*)
then have ‹(*L*, *C*) ∈# *WS*›
using *Q-M-L-WS* **by** *blast*
then have *False* **using** *KC-WS* **unfolding** *CD* **by** *simp*
then show *?thesis* **by** *fast*
next
case *K′L′* **note** [*simp*] = *this*
show *?thesis*
by (*rule IH-Q*[*of C*]) (*use CD watched-C uK′-M uK′′-M KC-WS L-ne-L′* **in** *auto*)
qed
qed
moreover {
have ‹(*L′*, *D*) ∉# *WS*›
using *no-dup L-ne-L′* **by** (*auto simp*: *all-conj-distrib*)
then have *?case* **if** *CD*: ‹*C* = *?D*›
using *IH-Q*[*of D L*] *IH-Q*[*of D L′*] *CD watched watched-D watched-C watched uK-M L′*
L-ne-L′ L-M uK′-M uK′′-M D-ne-D C **unfolding** *NU N′U′*
by (*auto simp*: *add-mset-eq-add-mset all-conj-distrib imp-conjR*)
}
ultimately show *?case*
by *blast*
qed
qed

lemma *twl-cp-twl-inv*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
wq: ‹*clauses-to-update-inv S*›
shows ‹*twl-st-inv T*›
using *cdcl twl valid inv twl-excep no-dup wq*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE L Q*) **note** *inv* = *this*(*1*)
then show *?case* **unfolding** *twl-st-inv.simps twl-is-an-exception-def*
by (*fastforce simp add*: *pair-in-image-Pair*)
next
case (*propagate D L L′ M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *undef* = *this*(*2*) **and**
unw = *this*(*3*) **and** *twl* = *this*(*4*) **and** *valid* = *this*(*5*) **and** *inv* = *this*(*6*) **and** *exception* = *this*(*7*)
have *uL′-M*[*simp*]: ‹− *L′* ∉ *lits-of-l M*›
using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *blast*
have *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
then have *wf-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp add*: *twl-st-inv.simps*)
have [*simp*]: ‹− *L* ∈ *lits-of-l M*›
using *valid* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*

$cdcl_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
show *?case* **unfolding** *twl-st-simps Ball-def*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N* + *U*›
show ‹*struct-wf-twl-cls C*›
using *twl C* **by** (*auto simp*: *twl-st-inv.simps*)[]
have *watched-max*: ‹*watched-literals-false-of-max-level M C*›
using *twl C* **by** (*auto simp*: *twl-st-inv.simps*)
then show ‹*watched-literals-false-of-max-level* (*Propagated L′* (*clause D*) # *M*) *C*›
using *undef n-d*
by (*cases C*) (*auto simp*: *get-level-cons-if dest*!: *no-has-blit-propagate′*)

assume *excep*: ‹¬*twl-is-an-exception C* (*add-mset* (− *L′*) *Q*) *WS*›
have *excep-C*: ‹¬ *twl-is-an-exception C Q* (*add-mset* (*L*, *D*) *WS*)› **if** ‹*C* ≠ *D*›
using *excep that* **by** (*auto simp add*: *twl-is-an-exception-def*)

then
have ‹*twl-lazy-update M C*› **if** ‹*C* ≠ *D*›
using *twl C D-N-U that* **by** (*cases* ‹*C* = *D*›) (*auto simp add*: *twl-st-inv.simps*)
then show ‹*twl-lazy-update* (*Propagated L′* (*clause D*) # *M*) *C*›
using *twl C excep uL′-M twl undef n-d uL′-M unw watched-max*
apply (*cases C*)
apply (*auto simp*: *get-level-cons-if count-decided-ge-get-level*
twl-is-an-exception-add-mset-to-queue atm-of-eq-atm-of
dest!: *no-has-blit-propagate′ no-has-blit-propagate*)
apply (*metis twl-clause.sel*(*2*) *uL′-M unw*)
apply (*metis twl-clause.sel*(*2*) *uL′-M unw*)
apply (*metis twl-clause.sel*(*2*) *uL′-M unw*)
apply (*metis twl-clause.sel*(*2*) *uL′-M unw*)
done
qed
next
case (*conflict D L L′ M N U NE UE WS Q*) **note** *twl* = *this*(*4*)
then show *?case*
by (*auto simp*: *twl-st-inv.simps*)
next
case (*delete-from-working L′ D M N U NE UE L WS Q*) **note** *watched* = *this*(*1*) **and** *L′* = *this*(*2*)
and
twl = *this*(*3*) **and** *valid* = *this*(*4*) **and** *inv* = *this*(*5*) **and** *tauto* = *this*(*6*)
show *?case* **unfolding** *twl-st-simps Ball-def*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N* + *U*›
show ‹*struct-wf-twl-cls C*›
using *twl C* **by** (*auto simp*: *twl-st-inv.simps*)[]
show ‹*watched-literals-false-of-max-level M C*›
using *twl C* **by** (*auto simp*: *twl-st-inv.simps*)

assume *excep*: ‹¬*twl-is-an-exception C Q WS*›
have ‹*get-level M L* = *count-decided M*› **and** *L*: ‹−*L* ∈ *lits-of-l M*› **and** *D*: ‹*D* ∈# *N* + *U*›
using *valid* **by** *auto*
have ‹*watched-literals-false-of-max-level M D*›
using *twl D* **by** (*auto simp*: *twl-st-inv.simps*)
have ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*

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

show ‹*twl-lazy-update M C*›
using *twl-C twl-D* **by** *blast*
qed
next
case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*) **and**
L' = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** $N'U'$ = *this*(*6*) **and** *twl* = *this*(*7*) **and**
valid = *this*(*8*) **and** *inv* = *this*(*9*) **and** *twl-excep* = *this*(*10*) **and**
no-dup = *this*(*11*) **and** *wq* = *this*(*12*)
obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
then have *struct-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *L′-UWD*: ‹*L* ∉# *remove1-mset L′ UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD L* ≥ *1*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *L* ≥ *2*›
using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have *L′-L′-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD K* ≥ *2*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
then have ‹*count* (*clause D*) *K* ≥ *2*›
using *D L′* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*]
split: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*

unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹$C \in\# N + U$› **if** ‹$C \neq ?D$› **and** C: ‹$C \in\# N' + U'$› **for** C
using C $N'U'$ *that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
then have *uK-M*: ‹$- K \notin$ *lits-of-l M*›
using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
distinct-consistent-interp **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) $\neq$ *WD*›
using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
have *cls-D-D*: ‹*clause ?D* = *clause D*›
by (*cases D*) (*use watched K* **in** *auto*)

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

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

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

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

assume *excep*: ‹$\neg$*twl-is-an-exception C Q WS*›

have ‹*get-level M L = count-decided M*› **and** *L*: ‹$-L \in$ *lits-of-l M*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*›
  using *valid* **by** *auto*

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

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

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

qed

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

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

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

dest: *mset-subset-eq-insertD*)
qed *auto*

lemma *distinct-mset-Pair*: ⟨*distinct-mset* (*Pair L* ‘# *C*) ⟷ *distinct-mset C*⟩
by (*induction C*) *auto*

lemma *distinct-image-mset-clause*:
⟨*distinct-mset* (*clause* ‘# *C*) ⟹ *distinct-mset C*⟩
by (*induction C*) *auto*

lemma *twl-cp-distinct-queued*:
assumes
cdcl: ⟨*cdcl-twl-cp S T*⟩ **and**
twl: ⟨*twl-st-inv S*⟩ **and**
valid: ⟨*valid-enqueued S*⟩ **and**
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)⟩ **and**
no-dup: ⟨*no-duplicate-queued S*⟩ **and**
dist: ⟨*distinct-queued S*⟩
shows ⟨*distinct-queued T*⟩
using *cdcl twl valid inv no-dup dist*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE L Q*) **note** *c-dist* = *this*(*4*) **and** *dist* = *this*(*5*)
show *?case*
using *dist* **by** (*auto simp*: *distinct-mset-Pair count-image-mset-Pair simp del*: *image-mset-union*)
next
case (*propagate D L L′ M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *undef* = *this*(*2*) **and**
twl = *this*(*4*) **and** *valid* = *this*(*5*) **and** *inv* = *this*(*6*) **and** *no-dup* = *this*(*7*)
and *dist* = *this*(*8*)
have ⟨*L′* ∉ *lits-of-l M*⟩
using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *auto*
then have ⟨−*L′* ∉# *Q*⟩
using *no-dup* **by** (*fastforce simp*: *lits-of-def dest*!: *mset-subset-eqD*)
then show *?case*
using *dist* **by** (*auto simp*: *all-conj-distrib split*: *if-splits dest*!: *Suc-leD*)
next
case (*conflict D L L′ M N U NE UE WS Q*) **note** *dist* = *this*(*8*)
then show *?case*
by *auto*
next
case (*delete-from-working D L L′ M N U NE UE WS Q*) **note** *dist* = *this*(*7*)
show *?case* **using** *dist* **by** (*auto simp*: *all-conj-distrib split*: *if-splits dest*!: *Suc-leD*)
next
case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*)
and
L′ = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N′U′* = *this*(*6*) **and** *twl* = *this*(*7*) **and**
valid = *this*(*8*) **and** *inv* = *this*(*9*) **and** *no-dup* = *this*(*10*) **and** *dist* = *this*(*11*)

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

fix *K″ C*
have *LD*: ⟨*Suc* (*count WS* (*L*, *D*)) ≤ *count N D* + *count U D*⟩
using *dist N′U′* **by** (*auto split*: *if-splits*)

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

lemma *twl-cp-valid*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
dist: ‹*distinct-queued S*›
shows ‹*valid-enqueued T*›
using *cdcl twl valid inv no-dup dist*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE L Q*) **note** *valid* = *this*(*2*)
then show *?case*
by (*auto simp del*: *filter-union-mset*)
next
case (*propagate D L L′ M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *twl* = *this*(*4*) **and** *valid* = *this*(*5*) **and** *inv* = *this*(*6*) **and** *no-taut* = *this*(*7*)
show *?case*
using *valid* **by** (*auto dest*: *mset-subset-eq-insertD simp*: *get-level-cons-if*)
next
case (*conflict D L L′ M N U NE UE WS Q*) **note** *valid* = *this*(*5*)
then show *?case*
by *auto*
next
case (*delete-from-working D L L′ M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *L′* = *this*(*2*) **and**
twl = *this*(*3*) **and** *valid* = *this*(*4*) **and** *inv* = *this*(*5*)
show *?case* **unfolding** *twl-st-simps Ball-def*
using *valid* **by** (*auto dest*: *mset-subset-eq-insertD*)
next
case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*) **and**
L′ = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N′U′* = *this*(*6*) **and** *twl* = *this*(*7*) **and** *valid* = *this*(*8*) **and** *inv* = *this*(*9*) **and** *no-dup* = *this*(*10*) **and** *dist* = *this*(*11*)
show *?case*
unfolding *valid-enqueued.simps Ball-def*

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

lemma *twl-cp-propa-cands-enqueued*:
  assumes

cdcl: ⟨*cdcl-twl-cp S T*⟩ **and**
twl: ⟨*twl-st-inv S*⟩ **and**
valid: ⟨*valid-enqueued S*⟩ **and**
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)⟩ **and**
twl-excep: ⟨*twl-st-exception-inv S*⟩ **and**
no-dup: ⟨*no-duplicate-queued S*⟩ **and**
cands: ⟨*propa-cands-enqueued S*⟩ **and**
ws: ⟨*clauses-to-update-inv S*⟩
shows ⟨*propa-cands-enqueued T*⟩
using *cdcl twl valid inv twl-excep no-dup cands ws*
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE L Q*) **note** *inv* = *this*(*1*) **and** *valid* = *this*(*2*) **and** *cands* = *this*(*6*)
show *?case* **unfolding** *propa-cands-enqueued.simps*
proof (*intro allI conjI impI*)
fix *C K*
assume *C*: ⟨*C* ∈# *N* + *U*⟩ **and**
⟨*K* ∈# *clause C*⟩ **and**
⟨*M* ⊨*as CNot* (*remove1-mset K* (*clause C*))⟩ **and**
⟨*undefined-lit M K*⟩
then have ⟨(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *add-mset L Q*)⟩
using *cands* **by** *auto*
then show
⟨(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *Q*) ∨
(∃ *La*. (*La*, *C*) ∈# *Pair L* ‘# {#*C* ∈# *N* + *U*. *L* ∈# *watched C*#})⟩
using *C* **by** *auto*
qed
next
case (*propagate D L L′ M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *undef* = *this*(*2*) **and**
false = *this*(*3*) **and**
twl = *this*(*4*) **and** *valid* = *this*(*5*) **and** *inv* = *this*(*6*) **and** *excep* = *this*(*7*)
and *no-dup* = *this*(*8*) **and** *cands* = *this*(*9*) **and** *to-upd* = *this*(*10*)
have *uL′-M*: ⟨− *L′* ∉ *lits-of-l M*⟩
using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *blast*
have *D-N-U*: ⟨*D* ∈# *N* + *U*⟩
using *valid* **by** *auto*
then have *wf-D*: ⟨*struct-wf-twl-cls D*⟩
using *twl* **by** (*simp add*: *twl-st-inv.simps*)
show *?case* **unfolding** *propa-cands-enqueued.simps*
proof (*intro allI conjI impI*)
fix *C K*
assume *C*: ⟨*C* ∈# *N* + *U*⟩ **and**
K: ⟨*K* ∈# *clause C*⟩ **and**
L′-M-C: ⟨*Propagated L′* (*clause D*) # *M* ⊨*as CNot* (*remove1-mset K* (*clause C*))⟩ **and**
undef-K: ⟨*undefined-lit* (*Propagated L′* (*clause D*) # *M*) *K*⟩
then have *wf-C*: ⟨*struct-wf-twl-cls C*⟩
using *twl* **by** (*simp add*: *twl-st-inv.simps*)
have *undef-K-M*: ⟨*undefined-lit M K*⟩
using *undef-K* **by** (*simp add*: *Decided-Propagated-in-iff-in-lits-of-l*)
consider
(*no-L′*) ⟨*M* ⊨*as CNot* (*remove1-mset K* (*clause C*))⟩ |
(*L′*) ⟨−*L′* ∈# *remove1-mset K* (*clause C*)⟩
using *L′-M-C* ⟨− *L′* ∉ *lits-of-l M*⟩
by (*metis insertE list.simps*(*15*) *lit-of.simps*(*2*) *lits-of-insert*
true-annots-CNot-lit-of-notin-skip true-annots-true-cls-def-iff-negation-in-model)
then show ⟨(∃ *L′a*. *L′a* ∈# *watched C* ∧ *L′a* ∈# *add-mset* (− *L′*) *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)⟩
proof *cases*

case *no-L′*
then have ⟨(∃ L′. L′ ∈# *watched* C ∧ L′ ∈# Q) ∨ (∃ La. (La, C) ∈# *add-mset* (L, D) WS)⟩
  using *cands C K undef-K-M* **by** *auto*
moreover {
  have ⟨K = L′⟩ **if** ⟨C = D⟩
    by (*metis* ⟨− L′ ∉ *lits-of-l* M⟩ *add-mset-add-single clause.simps in-CNot-implies-uminus*(2)
      in-remove1-mset-neq multi-member-this no-L′ that twl-clause.exhaust twl-clause.sel(1)
      union-iff watched)
  then have *False* **if** ⟨C = D⟩
    using *undef-K* **by** (*simp add*: *Decided-Propagated-in-iff-in-lits-of-l that*)
}
ultimately show *?thesis* **by** *auto*
next
case *L′*
have *?thesis* **if** ⟨L′ ∈# *watched* C⟩
proof −
  have ⟨K = L′⟩
    using *that L′-M-C* ⟨− L′ ∉ *lits-of-l* M⟩ *L′ undef*
    by (*metis clause.simps in-CNot-implies-uminus*(2) *in-lits-of-l-defined-litD*
      in-remove1-mset-neq insert-iff list.simps(15) *lits-of-insert*
      twl-clause.exhaust-sel uminus-not-id′ uminus-of-uminus-id union-iff)
  then have *False*
    using *Decided-Propagated-in-iff-in-lits-of-l undef-K* **by** *force*
  then show *?thesis*
    by *fastforce*
qed

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

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

  have ⟨*no-dup* M⟩
    using *inv* **unfolding** cdcl_W*-restart-mset.*cdcl_W*-all-struct-inv-def*
    cdcl_W*-restart-mset.*cdcl_W*-M-level-inv-def* **by** (*simp add*: *trail.simps*)

then have [*dest*]: *False* **if** ‹*a* ∈ *lits-of-l M*› **and** ‹−*a* ∈ *lits-of-l M*› **for** *a*
using *consistent-interp-def distinct-consistent-interp that*(*1*) *that*(*2*) **by** *blast*
have *uab*: ‹*a* ∉ *lits-of-l M*› **if** ‹−*b* ∈ *lits-of-l M*›
using *L′-M-C C-W-UW W that undef-K-M uL′-M*
by (*cases* ‹*K* = *a*›) (*fastforce simp*: *Decided-Propagated-in-iff-in-lits-of-l*
simp del: *uL′-M*)+
have *uba*: ‹*b* ∉ *lits-of-l M*› **if** ‹−*a* ∈ *lits-of-l M*›
using *L′-M-C C-W-UW W that undef-K-M uL′-M*
by (*cases* ‹*K* = *b*›) (*fastforce simp*: *Decided-Propagated-in-iff-in-lits-of-l*
add-mset-commute[*of a b*])+
have [*simp*]: ‹−*a* ≠ *L′*› ‹−*b* ≠ *L′*›
using *Q W C-W-UW* **by** *fastforce*+
have *H′*: ‹∀ *La L′*. *watched C* = {#*La*, *L′*#} ⟶ − *La* ∈ *lits-of-l M* ⟶
¬*has-blit M* (*clause C*) *La* ⟶ *L′* ∉ *lits-of-l M* ⟶
(∀ *K*∈#*unwatched C*. − *K* ∈ *lits-of-l M*)›
using *excep C CD Q W WS uab uba* **by** (*auto simp*: *twl-exception-inv.simps simp del*: *set-mset-union*
dest: *multi-member-split*)
moreover have ‹*watched C* = {#*La*, *L″*#} ⟶− *La* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*clause C*) *La*› **for** *La L″*
using *in-CNot-implies-uminus*[*OF - L′-M-C*] *wf-C L′ uL′-M undef-K-M undef uab uba*
unfolding *C-W-UW has-blit-def* **apply** −
apply (*cases* ‹*La* = *K*›)
apply (*auto simp*: *has-blit-def Decided-Propagated-in-iff-in-lits-of-l W*
add-mset-eq-add-mset in-remove1-mset-neq)
apply (*metis* ‹⋀*a*. ⟦*a* ∈ *lits-of-l M*; − *a* ∈ *lits-of-l M*⟧ ⟹ *False*› *add-mset-remove-trivial*
defined-lit-uminus in-lits-of-l-defined-litD in-remove1-mset-neq undef)
apply (*metis* ‹⋀*a*. ⟦*a* ∈ *lits-of-l M*; − *a* ∈ *lits-of-l M*⟧ ⟹ *False*› *add-mset-remove-trivial*
defined-lit-uminus in-lits-of-l-defined-litD in-remove1-mset-neq undef)
done
ultimately have ‹∀ *K*∈#*unwatched C*. − *K* ∈ *lits-of-l M*›
using *uab uba W C-W-UW ua-or-ub wf-C* **unfolding** *C-W-UW*
by (*auto simp*: *add-mset-eq-add-mset*)
then show *False*
by (*metis Decided-Propagated-in-iff-in-lits-of-l L′ uminus-lit-swap*
Q clause.simps in-diffD propagate.hyps(*2*) *twl-clause.collapse union-iff*)
qed

ultimately show *?thesis* **by** *fast*
qed
qed
next
case (*conflict D L L′ M N U NE UE WS Q*) **note** *cands* = *this*(*10*)
then show *?case*
by *auto*
next
case (*delete-from-working L′ D M N U NE UE L WS Q*) **note** *watched* = *this*(*1*) **and** *L′* = *this*(*2*) **and**
twl = *this*(*3*) **and** *valid* = *this*(*4*) **and** *inv* = *this*(*5*) **and** *cands* = *this*(*8*) **and** *ws* = *this*(*9*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps*)
show *?case* **unfolding** *propa-cands-enqueued.simps*
proof (*intro allI conjI impI*)
fix *C K*
assume *C*: ‹*C* ∈# *N* + *U*› **and**

K: ‹*K* ∈# *clause C*› **and**
L′-M-C: ‹*M* ⊨*as CNot* (*remove1-mset K* (*clause C*))› **and**
undef-K: ‹*undefined-lit M K*›
then have ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *Q*) ∨ (∃ *La*. *La* = *L* ∧ *C* = *D* ∨ (*La*, *C*) ∈# *WS*)›
using *cands* **by** *auto*
moreover have *False* **if** [*simp*]: ‹*C* = *D*›
using *L′ L′-M-C undef-K watched*
using *Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def distinct-consistent-interp local.K n-d K*
by (*cases D*)
(*auto 5 5 simp*: *true-annots-true-cls-def-iff-negation-in-model add-mset-eq-add-mset dest*: *in-lits-of-l-defined-litD no-dup-consistentD dest*!: *multi-member-split*)
ultimately show ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)›
by *auto*
qed
next
case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *uL* = *this*(*2*) **and**
L′ = *this*(*3*) **and** *K* = *this*(*4*) **and** *undef* = *this*(*5*) **and** *N′U′* = *this*(*6*) **and** *twl* = *this*(*7*) **and** *valid* = *this*(*8*) **and** *inv* = *this*(*9*) **and** *twl-excep* = *this*(*10*) **and** *no-dup* = *this*(*11*) **and** *cands* = *this*(*12*) **and** *ws* = *this*(*13*)
obtain *WD UWD* **where** *D*: ‹*D* = *TWL-Clause WD UWD*› **by** (*cases D*)
have *L*: ‹*L* ∈# *watched D*› **and** *D-N-U*: ‹*D* ∈# *N* + *U*› **and** *lev-L*: ‹*get-level M L* = *count-decided M*›
using *valid* **by** *auto*
then have *struct-D*: ‹*struct-wf-twl-cls D*›
using *twl* **by** (*auto simp*: *twl-st-inv.simps*)
have *L′-UWD*: ‹*L* ∉# *remove1-mset L′ UWD*› **if** ‹*L* ∈# *WD*› **for** *L*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD L* ≥ *1*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
then have ‹*count* (*clause D*) *L* ≥ *2*›
using *D that* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have *L′-L′-UWD*: ‹*K* ∉# *remove1-mset K UWD*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*count UWD K* ≥ *2*›
by (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
then have ‹*count* (*clause D*) *K* ≥ *2*›
using *D L′* **by** (*auto simp del*: *count-greater-zero-iff simp*: *count-greater-zero-iff*[*symmetric*] *split*: *if-splits*)
moreover have ‹*distinct-mset* (*clause D*)›
using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
ultimately show *False*
unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›

  using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N′* + *U′*› **for** *C*
  using *C N′U′ that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
  using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
    $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
  using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
    distinct-consistent-interp **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
  using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
have *D-N-U*: ‹*D* ∈# *N* + *U*›
  using *N′U′ D uK-M uL D-N-U* **by** (*auto simp*: *add-mset-remove-trivial-iff split*: *if-splits*)
have *D-ne-D*: ‹*D* ≠ *update-clause D L K*›
  using *D add-remove-WD* **by** *auto*

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

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

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

    assume ‹¬ ((∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))›
    then have [*simp*]: ‹*L′* ∉# *Q*› **and** *L′-C-WS*: ‹(*L′*, *C*) ∉# *WS*›

using *watched D* **by** *auto*
have ‹$C \in\#$ *add-mset* (*L*, *TWL-Clause WD UWD*) *WS* $\longrightarrow$
$C' \in\#$ *add-mset* (*L*, *TWL-Clause WD UWD*) *WS* $\longrightarrow$
fst C = *fst C'*› **for** *C C'*
using *no-dup* **unfolding** *D no-duplicate-queued.simps*
by *blast*
from *this*[*of* ‹(*L*, *TWL-Clause WD UWD*)› ‹(*L'*, *TWL-Clause* {#*L*, *L'*#} *UWD*)›]
have *notin*: ‹*False*› **if** ‹(*L'*, *TWL-Clause* {#*L*, *L'*#} *UWD*) $\in\#$ *WS*›
using *struct-D watched that* **unfolding** *D*
by *auto*
have ‹*?D* $\neq$ *D*›
using *C D watched L K uK-M uL* **by** *auto*
then have *excep*: ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *D*) *WS*, *Q*) *D*›
using *twl-excep* ∗[*of D*] *D-N-U* **by** (*auto simp*: *twl-st-inv.simps*)
moreover have ‹*D* = *TWL-Clause* {#*L*, *L'*#} *UWD* $\Longrightarrow$
WD = {#*L*, *L'*#} $\Longrightarrow$
$\forall$ *L*$\in\#$*remove1-mset K UWD*.
− *L* $\in$ *lits-of-l M* $\Longrightarrow$
¬*has-blit M* (*add-mset L* (*add-mset L' UWD*)) *L'*›
using *uL uL'-M n-d* ‹*K* $\notin$ *lits-of-l M*› **unfolding** *has-blit-def*
apply (*auto dest*:*no-dup-consistentD simp*: *in-remove1-mset-neq Ball-def*)
by (*metis in-remove1-mset-neq no-dup-consistentD*)
ultimately have ‹$\forall$ *K* $\in\#$ *unwatched D*. −*K* $\in$ *lits-of-l M*›
using *D watched L'-M-C L'-C-WS*
by (*auto simp*: *add-mset-eq-add-mset uL'-M L-M uL twl-exception-inv.simps*
true-annots-true-cls-def-iff-negation-in-model dest: *in-diffD notin*)
then show *False*
using *uK-M update-clause.hyps*(*4*) **by** *blast*
qed
moreover have ‹($\exists$ *L'*. *L'* $\in\#$ *watched C* $\wedge$ *L'* $\in\#$ *Q*) $\vee$ ($\exists$ *L*. (*L*, *C*) $\in\#$ *WS*)› **if** [*simp*]: ‹*C* = *D*›
unfolding *that*
proof −
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
have *N'U'*: ‹*N'* + *U'* = *add-mset ?D* (*remove1-mset D* (*N* + *U*))›
using *N'U' D-N-U* **by** (*auto elim*!: *update-clausesE*)

have ‹*add-mset L Q* $\subseteq\#$ {#− *lit-of x*. *x* $\in\#$ *mset M*#}›
using *no-dup* **by** (*auto*)
moreover have ‹*distinct-mset* {#− *lit-of x*. *x* $\in\#$ *mset M*#}›
by (*subst distinct-image-mset-inj*)
(*use n-d* **in** ‹*auto simp*: *lit-of-inj-on-no-dup distinct-map no-dup-def*›)
ultimately have [*simp*]: ‹*L* $\notin\#$ *Q*›
by (*metis distinct-mset-add-mset distinct-mset-union subset-mset.le-iff-add*)
have ‹*has-blit M* (*clause D*) *L* $\Longrightarrow$ *False*›
by (*smt K L'-M-C has-blit-def in-lits-of-l-defined-litD insert-DiffM insert-iff*
is-blit-def n-d no-dup-consistentD set-mset-add-mset-insert that
true-annots-true-cls-def-iff-negation-in-model undef-K)
then have *w-q-p-D*: ‹*clauses-to-update-prop Q M* (*L*, *D*)›
by (*auto simp*: *clauses-to-update-prop.simps watched*)
(*use uL undef L'* **in** ‹*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*›)
have ‹*Pair L* ‘# {#*C* $\in\#$ *add-mset D NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} $\subseteq\#$
add-mset (*L*, *D*) *WS*›

using *ws no-dup* **unfolding** *clauses-to-update-inv.simps NU*
by (*auto simp*: *all-conj-distrib*)
then have *IH*: ‹*Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*›
using *w-q-p-D* **by** *auto*
moreover have ‹(*L*, *D*) ∈# *Pair L* ‘# {#*C* ∈# *NU*. *clauses-to-update-prop Q M* (*L*, *C*)#}›
using *C D-ne-D w-q-p-D* **unfolding** *NU N'U'* **by** (*auto simp*: *pair-in-image-Pair*)
ultimately show ‹(∃ *L'*. *L'* ∈# *watched D* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *D*) ∈# *WS*)›
by *blast*
qed
ultimately show ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*)›
by *auto*
qed
qed

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

L'-M-C: ‹*Propagated* L' (*clause* D) # M ⊨*as CNot* (*clause* C)›
consider
(*no*-L') ‹M ⊨*as CNot* (*clause* C)›
| (L') ‹$-L'$ ∈# *clause* C›
using L'-M-C ‹$- L' \notin$ *lits-of-l* M›
by (*metis insertE list.simps*(*15*) *lit-of.simps*(*2*) *lits-of-insert*
true-annots-CNot-lit-of-notin-skip true-annots-true-cls-def-iff-negation-in-model)
then show ‹($\exists L'a$. $L'a$ ∈# *watched* C ∧ $L'a$ ∈# *add-mset* ($- L'$) Q) ∨ ($\exists L$. (L, C) ∈# WS)›
proof *cases*
case *no*-L'
then have ‹($\exists L'$. L' ∈# *watched* C ∧ L' ∈# Q) ∨ ($\exists La$. (La, C) ∈# *add-mset* (L, D) WS)›
using *cands* C **by** (*auto simp*: S)
moreover {
have ‹$C \neq D$›
by (*metis* ‹$- L' \notin$ *lits-of-l* M› *add-mset-add-single clause.simps in-CNot-implies-uminus*(*2*)
multi-member-this no-L' *twl-clause.exhaust twl-clause.sel*(*1*)
union-iff watched)
}
ultimately show *?thesis* **by** *auto*
next
case L'
have L'-C: ‹L' ∉# *watched* C›
using L'-M-C ‹$- L' \notin$ *lits-of-l* M›
by (*metis* (*no-types*, *hide-lams*) *Decided-Propagated-in-iff-in-lits-of-l* L' *clause.simps*
in-CNot-implies-uminus(*2*) *insertE list.simps*(*15*) *lits-of-insert twl-clause.exhaust-sel*
uminus-not-id' *uminus-of-uminus-id undef union-iff*)
moreover have *?thesis*
proof (*rule ccontr*, *clarsimp*)
assume
Q: ‹$\forall L'a$. $L'a$ ∈# *watched* C ⟶ $L'a \neq - L'$ ∧ $L'a$ ∉# Q› **and**
WS: ‹$\forall L$. (L, C) ∉# WS›
then have ‹¬ *twl-is-an-exception* C (*add-mset* ($- L'$) Q) WS›
by (*auto simp*: *twl-is-an-exception-def*)
moreover have
‹*twl-st-inv* (*Propagated* L' (*clause* D) # M, N, U, *None*, NE, UE, WS, *add-mset* ($- L'$) Q)›
using *twl-cp-twl-inv*[*OF - twl valid inv excep no-dup ws*] *cdcl* **unfolding** S T **by** *fast*
ultimately have ‹*twl-lazy-update* (*Propagated* L' (*clause* D) # M) C›
using C **by** (*auto simp*: *twl-st-inv.simps*)

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

have ‹*no-dup* M›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps* S)

then have [*dest*]: *False* **if** ‹$a \in$ *lits-of-l M*› **and** ‹$-a \in$ *lits-of-l M*› **for** *a*
using *consistent-interp-def distinct-consistent-interp that*(*1*) *that*(*2*) **by** *blast*
have *uab*: ‹$a \notin$ *lits-of-l M*› **if** ‹$-b \in$ *lits-of-l M*›
using *L′-M-C C-W-UW W that uL′-M* **by** (*cases* ‹$K = a$›) *auto*
have *uba*: ‹$b \notin$ *lits-of-l M*› **if** ‹$-a \in$ *lits-of-l M*›
using *L′-M-C C-W-UW W that uL′-M* **by** (*cases* ‹$K = b$›) *auto*
have [*simp*]: ‹$-a \neq L'$› ‹$-b \neq L'$›
using ‹$\forall L'a.\ L'a \in\#$ *watched C* $\longrightarrow L'a \neq -\ L' \wedge L'a \notin\# Q$› *W C-W-UW*
by *fastforce*+
have H': ‹$\forall La\ L'.$ *watched C* $= \{\#La,\ L'\#\} \longrightarrow -\ La \in$ *lits-of-l M* $\longrightarrow L' \notin$ *lits-of-l M* $\longrightarrow$
$\neg$ *has-blit M* (*clause C*) *La* $\longrightarrow$($\forall K{\in}\#$*unwatched C*. $-\ K \in$ *lits-of-l M*)›
using *excep C CD Q W WS uab uba*
by (*auto simp*: *twl-exception-inv.simps S dest*: *multi-member-split*)
moreover have ‹$\neg$ *has-blit M* (*clause C*) *a*› ‹$\neg$ *has-blit M* (*clause C*) *b*›
using *multi-member-split*[*OF C*]
using *watched L′ undef L′-M-C*
unfolding *has-blit-def*
by (*metis* (*no-types*, *lifting*) *Clausal-Logic.uminus-lit-swap*
‹$\bigwedge a.\ [\![a \in$ *lits-of-l M*; $-\ a \in$ *lits-of-l M*$]\!] \Longrightarrow$ *False*› *in-CNot-implies-uminus*(*2*)
in-lits-of-l-defined-litD insert-iff is-blit-def list.set(*2*) *lits-of-insert uL′-M*)+
ultimately have ‹$\forall K{\in}\#$*unwatched C*. $-\ K \in$ *lits-of-l M*›
using *uab uba W C-W-UW ua-ub struct*
by (*auto simp*: *add-mset-eq-add-mset*)
then show *False*
by (*metis Decided-Propagated-in-iff-in-lits-of-l L′ uminus-lit-swap*
Q clause.simps undef twl-clause.collapse union-iff)
qed
ultimately show *?thesis* **by** *fast*
qed
qed
next
case (*conflict D L L′ M N U NE UE WS Q*)
then show *?case*
by *auto*
next
case (*delete-from-working L′ D M N U NE UE L WS Q*) **note** $S =$ *this*(*1*) **and** $T =$ *this*(*2*) **and**
watched $=$ *this*(*3*) **and** $L' =$ *this*(*4*)
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*simp add*: *trail.simps S*)
show *?case* **unfolding** *confl-cands-enqueued.simps Ball-def S T*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹$C \in\# N + U$› **and**
L′-M-C: ‹$M \models as$ *CNot* (*clause C*)›
then have ‹($\exists L'.\ L' \in\#$ *watched C* $\wedge L' \in\# Q$) $\vee$ ($\exists La.\ La = L \wedge C = D \vee (La,\ C) \in\# WS$)›
using *cands S* **by** *auto*
moreover have *False* **if** [*simp*]: ‹$C = D$›
using *L′-M-C watched L′ n-d* **by** (*cases D*) (*auto dest*!: *distinct-consistent-interp*
simp: *consistent-interp-def dest*!: *multi-member-split*)
ultimately show ‹($\exists L'.\ L' \in\#$ *watched C* $\wedge L' \in\# Q$) $\vee$ ($\exists L.\ (L,\ C) \in\# WS$)›
by *auto*
qed
next
case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** $S =$ *this*(*1*) **and** $T =$ *this*(*2*) **and**
watched $=$ *this*(*3*) **and** $uL =$ *this*(*4*) **and** $L' =$ *this*(*5*) **and** $K =$ *this*(*6*) **and** *undef* $=$ *this*(*7*) **and**

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

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

have *L-M*: ‹*L* ∉ *lits-of-l M*›
using *n-d uL* **by** (*fastforce dest*!: *distinct-consistent-interp*

simp: *consistent-interp-def lits-of-def uminus-lit-swap*)
have *w-max-D*: ‹*watched-literals-false-of-max-level M D*›
using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps S*)

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

show *?case* **unfolding** *confl-cands-enqueued.simps Ball-def S T*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹$C \in\# N' + U'$› **and**
L'-M-C: ‹*M* ⊨*as CNot* (*clause C*)›
then have ‹$(\exists L'.\ L' \in\#$ *watched C* $\wedge L' \in\# Q) \vee (\exists La.\ (La, C) \in\# WS)$› **if** ‹$C \neq ?D$› ‹$C \neq D$›
using *cands* ∗[*OF that*(*1*) *C*] *that*(*2*) *S* **by** *auto*
moreover have ‹$C \neq ?D$›
by (*metis D L'-M-C add-diff-cancel-left' clause.simps clause-D in-diffD true-annots-true-cls-def-iff-negation-in-model twl-clause.sel*(*2*) *uK-M K*)
moreover have ‹$(\exists L'.\ L' \in\#$ *watched C* $\wedge L' \in\# Q) \vee (\exists La.\ (La, C) \in\# WS)$› **if** [*simp*]: ‹$C = D$›
unfolding *that*
proof −
obtain *NU* **where** *NU*: ‹*N* + *U* = *add-mset D NU*›
by (*metis D-N-U insert-DiffM*)
have *N'U'*: ‹*N'* + *U'* = *add-mset ?D* (*remove1-mset D* (*N* + *U*))›
using *N'U' D-N-U* **by** (*auto elim*!: *update-clausesE*)

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

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

lemma *twl-cp-past-invs*:
  assumes
    cdcl: ‹*cdcl-twl-cp S T*› **and**
    twl: ‹*twl-st-inv S*› **and**
    valid: ‹*valid-enqueued S*› **and**
    inv: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
    twl-excep: ‹*twl-st-exception-inv S*› **and**
    no-dup: ‹*no-duplicate-queued S*› **and**
    past-invs: ‹*past-invs S*›
  shows ‹*past-invs T*›
  using *cdcl twl valid inv twl-excep no-dup past-invs*
proof (*induction rule*: *cdcl-twl-cp.induct*)
  case (*pop M N U NE UE L Q*) **note** *past-invs* = *this*(*6*)
  then show *?case*
    by (*subst past-invs-enqueud*, *subst* (*asm*) *past-invs-enqueud*)
next
  case (*propagate D L L' M N U NE UE WS Q*) **note** *watched* = *this*(*1*) **and** *twl* = *this*(*4*) **and**
    valid = *this*(*5*) **and** *inv* = *this*(*6*) **and** *past-invs* = *this*(*9*)
  have [*simp*]: ‹− *L'* ∉ *lits-of-l M*›
    using *Decided-Propagated-in-iff-in-lits-of-l propagate.hyps*(*2*) **by** *blast*
  have *D-N-U*: ‹*D* ∈# *N* + *U*›
    using *valid* **by** *auto*
  then have *wf-D*: ‹*struct-wf-twl-cls D*›
    using *twl* **by** (*simp add*: *twl-st-inv.simps*)
  show *?case* **unfolding** *past-invs.simps Ball-def*
  proof (*intro allI conjI impI*)
    fix *C*
    assume *C*: ‹*C* ∈# *N* + *U*›

    fix *M1 M2* :: ‹(*'a*, *'a clause*) *ann-lits*› **and** *K*
    assume ‹*Propagated L'* (*clause D*) # *M* = *M2* @ *Decided K* # *M1*›
    then have *M*: ‹*M* = *tl M2* @ *Decided K* # *M1*›
      by (*meson* $cdcl_W$*-restart-mset.propagated-cons-eq-append-decide-cons*)
    then show
      ‹*twl-lazy-update M1 C*› **and**
      ‹*watched-literals-false-of-max-level M1 C*› **and**
      ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
      using *C past-invs* **by** (*auto simp add*: *past-invs.simps*)
  next
    fix *M1 M2* :: ‹(*'a*, *'a clause*) *ann-lits*› **and** *K*
    assume ‹*Propagated L'* (*clause D*) # *M* = *M2* @ *Decided K* # *M1*›
    then have *M*: ‹*M* = *tl M2* @ *Decided K* # *M1*›
      by (*meson* $cdcl_W$*-restart-mset.propagated-cons-eq-append-decide-cons*)
    then show ‹*confl-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})› **and**
      ‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})› **and**
      ‹*clauses-to-update-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})›
      using *past-invs* **by** (*auto simp add*: *past-invs.simps*)
  qed
next
  case (*conflict D L L' M N U NE UE WS Q*) **note** *twl* = *this*(*9*)
  then show *?case*
    by (*auto simp*: *past-invs.simps*)
next
  case (*delete-from-working L' D M N U NE UE L WS Q*) **note** *watched* = *this*(*1*) **and** *L'* = *this*(*2*)
and

twl = *this*(*3*) **and** *valid* = *this*(*4*) **and** *inv* = *this*(*5*) **and** *past-invs* = *this*(*8*)
show *?case* **unfolding** *past-invs.simps Ball-def*
proof (*intro allI conjI impI*)
fix *C*
assume *C*: ‹*C* ∈# *N* + *U*›

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

    using *struct-D D* **by** (*auto simp*: *distinct-mset-union*)
  ultimately show *False*
    unfolding *distinct-mset-count-less-1* **by** (*metis Suc-1 not-less-eq-eq*)
qed
have ‹*watched-literals-false-of-max-level M D*›
  using *D-N-U twl* **by** (*auto simp*: *twl-st-inv.simps*)
let *?D* = ‹*update-clause D L K*›
have ∗: ‹*C* ∈# *N* + *U*› **if** ‹*C* ≠ *?D*› **and** *C*: ‹*C* ∈# *N'* + *U'*› **for** *C*
  using *C N'U' that* **by** (*auto elim*!: *update-clausesE dest*: *in-diffD*)
have *n-d*: ‹*no-dup M*›
  using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
    $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
then have *uK-M*: ‹− *K* ∉ *lits-of-l M*›
  using *undef Decided-Propagated-in-iff-in-lits-of-l consistent-interp-def*
    distinct-consistent-interp **by** *blast*
have *add-remove-WD*: ‹*add-mset K* (*remove1-mset L WD*) ≠ *WD*›
  using *uK-M uL* **by** (*auto simp*: *add-mset-remove-trivial-iff trivial-add-mset-remove-iff*)
have *cls-D-D*: ‹*clause ?D* = *clause D*›
  by (*cases D*) (*use watched K* **in** *auto*)

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

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

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

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

  have ‹*twl-lazy-update M1 D*›
    using *past-invs D-N-U* **unfolding** *past-invs.simps M twl-lazy-update.simps C*
    by *fast*
  then have
    lazy-L': ‹− *L'* ∈ *lits-of-l M1* ⟹ ¬ *has-blit M1* (*add-mset L* (*add-mset L' UWD*)) *L'* ⟹
        (∀ *K*∈#*UWD*. *get-level M1 K* ≤ *get-level M1 L'* ∧ − *K* ∈ *lits-of-l M1*)›
    using *watched* **unfolding** *D twl-lazy-update.simps*
    by (*simp-all add*: *all-conj-distrib*)
  have *excep-inv*: ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*› **if** ‹*C* ≠ *?D*›
    using ∗ *C past-invs that M* **by** (*auto simp add*: *past-invs.simps*)
  then have ‹*twl-exception-inv* (*M1*, *N'*, *U'*, *None*, *NE*, *UE*, {#}, {#}) *C*› **if** ‹*C* ≠ *?D*›
    using *N'U' that* **by** (*auto simp add*: *twl-st-inv.simps twl-exception-inv.simps*)
  moreover have ‹*twl-lazy-update M1 C*› ‹*watched-literals-false-of-max-level M1 C*›
    if ‹*C* ≠ *?D*›

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

then have *N'U'*: ‹$N'+U'$ = *add-mset ?D NU*›
  unfolding *NU-remove* **.**
have *watched-D*: ‹*watched ?D* = {#*K*, *L'*#}›
  using *D add-remove-WD watched* **by** *auto*

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

1.1.3 Invariants and the Transition System

Conflict and propagate

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

lemma *twl-cp-propagate-or-conflict*:
  assumes

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

      apply (*cases D*; *auto simp add*: *clauses-def*
        trail.simps twl-st-inv.simps; *fail*)
    by (*simp add*: $cdcl_W$*-restart-mset-state del*: $cdcl_W$*-restart-mset.state-simp*)
  then show *?case* **by** *fast*
next
  case (*delete-from-working D L L′ M N U NE UE WS Q*)
  then show *?case* **by** (*simp add*: *lexn2-conv*)
next
  case (*update-clause D L L′ M K N U N′ U′ NE UE WS Q*) **note** *unwatched* = *this*(*4*) **and**
    valid = *this*(*8*)
  have ‹*D* ∈# *N* + *U*›
    using *valid* **by** *auto*
  have [*simp*]: ‹*clause* (*update-clause D L K*) = *clause D*›
    using *valid unwatched* **by** (*cases D*) (*auto simp*: *diff-union-swap2*[*symmetric*]
        simp del: *diff-union-swap2*)
  have ‹$state_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *D*) *WS*, *Q*) =
    $state_W$*-of* (*M*, *N′*, *U′*, *None*, *NE*, *UE*, *WS*, *Q*)›
    ‹(*literals-to-update-measure* (*M*, *N′*, *U′*, *None*, *NE*, *UE*, *WS*, *Q*),
      literals-to-update-measure (*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *D*) *WS*, *Q*))
    ∈ *lexn less-than 2*›
    using *update-clause* ‹*D* ∈# *N* + *U*› **by** (*cases* ‹*D* ∈# *N*›)
      (*fastforce simp*: *image-mset-remove1-mset-if elim*!: *update-clausesE*
        simp add: *lexn2-conv*)+
  then show *?case* **by** *fast*
qed

lemma *cdcl-twl-o-*$cdcl_W$*-o*:
  assumes
    cdcl: ‹*cdcl-twl-o S T*› **and**
    twl: ‹*twl-st-inv S*› **and**
    valid: ‹*valid-enqueued S*› **and**
    inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)›
  shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-o* ($state_W$*-of S*) ($state_W$*-of T*)›
  using *cdcl twl valid inv*
proof (*induction rule*: *cdcl-twl-o.induct*)
  case (*decide M L N NE U UE*) **note** *undef* = *this*(*1*) **and** *atm* = *this*(*2*)
  have ‹$cdcl_W$*-restart-mset.decide* ($state_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}))
    ($state_W$*-of* (*Decided L* # *M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#−*L*#}))›
    apply (*rule* $cdcl_W$*-restart-mset.decide-rule*)
       apply (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
      using *undef* **apply** (*simp add*: *trail.simps*; *fail*)
     using *atm* **apply** (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
    by (*simp add*: *state-eq-def* $cdcl_W$*-restart-mset-state del*: $cdcl_W$*-restart-mset.state-simp*)
  then show *?case*
    by (*blast dest*: $cdcl_W$*-restart-mset.*$cdcl_W$*-o.intros*)
next
  case (*skip L D C′ M N U NE UE*) **note** *LD* = *this*(*1*) **and** *D* = *this*(*2*)
  show *?case*
    apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-o.bj*)
    apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-bj.skip*)
    apply (*rule* $cdcl_W$*-restart-mset.skip-rule*)
        apply (*simp add*: *trail.simps*; *fail*)
       apply (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
      using *LD* **apply** (*simp*; *fail*)
     using *D* **apply** (*simp*; *fail*)
    by (*simp add*: *state-eq-def* $cdcl_W$*-restart-mset-state del*: $cdcl_W$*-restart-mset.state-simp*)

next
  case (*resolve L D C M N U NE UE*) **note** *LD* = *this*(*1*) **and** *lev* = *this*(*2*) **and** *inv* = *this*(*5*)
  have ⟨∀ *La mark a b. a* @ *Propagated La mark* # *b* = *Propagated L C* # *M* ⟶
    b ⊨*as CNot* (*remove1-mset La mark*) ∧ *La* ∈# *mark*⟩
    using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
     cdcl$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
    by (*auto simp*: *trail.simps*)
  then have *LC*: ⟨*L* ∈# *C*⟩
    by *blast*
  show *?case*
    apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*-o.bj*)
    apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*-bj.resolve*)
    apply (*rule cdcl*$_W$*-restart-mset.resolve-rule*)
          apply (*simp add*: *trail.simps*; *fail*)
         apply (*simp add*: *trail.simps*; *fail*)
        using *LC* **apply** (*simp add*: *trail.simps*; *fail*)
       apply (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
      using *LD* **apply** (*simp*; *fail*)
     using *lev* **apply** (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
    by (*simp add*: *state-eq-def cdcl*$_W$*-restart-mset-state del*: *cdcl*$_W$*-restart-mset.state-simp*)
next
  case (*backtrack-unit-clause L D K M1 M2 M D′ i N U NE UE*) **note** *L-D* = *this*(*1*) **and**
     decomp = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *max-D′-L* = *this*(*4*) **and** *lev-D* = *this*(*5*) **and**
     lev-K = *this*(*6*) **and** *D′-D* = *this*(*8*) **and** *NU-D′* = *this*(*9*) **and** *inv* = *this*(*12*) **and**
     D′[*simp*] = *this*(*7*)
  let *?S* = ⟨*state*$_W$*-of* (*M*, *N*, *U*, *Some* {#*L*#}, *NE*, *UE*, {#}, {#})⟩
   let *?T* = ⟨*state*$_W$*-of* (*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, {#},
{#*L*#})⟩
  have *n-d*: ⟨*no-dup M*⟩
    using *inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
      cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
    by (*simp add*: *cdcl*$_W$*-restart-mset-state*)
  have ⟨*undefined-lit M1 L*⟩
    apply (*rule cdcl*$_W$*-restart-mset.backtrack-lit-skiped*[*of ?S - K - M2 i*])
    subgoal using *lev-L inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
      cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
      by (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
    subgoal using *decomp* **by** (*simp add*: *trail.simps*; *fail*)
    subgoal
     using *lev-L inv* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
       by (*simp add*: *cdcl*$_W$*-restart-mset-state*; *fail*)
    subgoal using *lev-K* **by** (*simp add*: *trail.simps*; *fail*)
    done
  obtain *M3* **where** *M3*: ⟨*M* = *M3* @ *M2* @ *Decided K* # *M1*⟩
    using *decomp* **by** (*blast dest*!: *get-all-ann-decomposition-exists-prepend*)
  have *D*: ⟨*D* = *add-mset L* (*remove1-mset L D*)⟩
    using *L-D* **by** *auto*
  have ⟨*undefined-lit* (*M3* @ *M2*) *K*⟩
    using *n-d* **unfolding** *M3* **by** *auto*
  then have [*simp*]: ⟨*count-decided M1* = *0*⟩
    using *lev-D lev-K* **by** (*auto simp*: *M3 image-Un*)
  show *?case*
    apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*-o.bj*)
    apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*-bj.backtrack*)
    apply (*rule cdcl*$_W$*-restart-mset.backtrack-rule*[*of - L* ⟨*remove1-mset L D*⟩ *K M1 M2*
        ⟨*remove1-mset L D′*⟩ *i*])

subgoal using *L-D* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *decomp* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *lev-L* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *max-D'-L L-D* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *lev-D L-D* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *lev-K* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *D'-D* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *NU-D'* **by** (*simp add*: $cdcl_W$*-restart-mset-state clauses-def ac-simps*)
subgoal using *decomp* **unfolding** *state-eq-def state-def prod.inject*
by (*simp add*: $cdcl_W$*-restart-mset-state*)
done
next
case (*backtrack-nonunit-clause L D K M1 M2 M D' i N U NE UE L'*) **note** *LD* = *this*(*1*) **and** *decomp* = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *max-lev* = *this*(*4*) **and** *i* = *this*(*5*) **and** *lev-K* = *this*(*6*) **and** *D'-D* = *this*(*8*) **and** *NU-D'* = *this*(*9*) **and** *L-D'* = *this*(*10*) **and** *L'* = *this*(*11*−*12*) **and** *inv* = *this*(*15*)
let *?S* = ‹$state_W$*-of* (*M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})›
let *?T* = ‹$state_W$*-of* (*Propagated L D* # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, {#}, {#*L*#})›
have *n-d*: ‹*no-dup M*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*)
have ‹*undefined-lit M1 L*›
apply (*rule* $cdcl_W$*-restart-mset.backtrack-lit-skiped*[*of ?S - K - M2 i*])
subgoal
using *lev-L inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
subgoal using *decomp* **by** (*simp add*: *trail.simps*; *fail*)
subgoal using *lev-L inv*
unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
subgoal using *lev-K* **by** (*simp add*: *trail.simps*; *fail*)
done
obtain *M3* **where** *M3*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** (*blast dest*!: *get-all-ann-decomposition-exists-prepend*)

have ‹*undefined-lit* (*M3* @ *M2*) *K*›
using *n-d* **unfolding** *M3* **by** (*auto simp*: *lits-of-def*)
then have *count-M1*: ‹*count-decided M1* = *i*›
using *lev-K* **unfolding** *M3* **by** (*auto simp*: *image-Un*)
have ‹$L \neq L'$›
using *L' lev-L lev-K count-decided-ge-get-level*[*of M K*] *L'* **by** *auto*
then have *D*: ‹*add-mset L* (*add-mset L'* (*D'* − {#*L*, *L'*#})) = *D'*›
using *L' L-D'*
by (*metis add-mset-diff-bothsides diff-single-eq-union insert-noteq-member mset-add*)
have *D'*: ‹*remove1-mset L D'* = *add-mset L'* (*D'* − {#*L*, *L'*#})›
by (*subst D*[*symmetric*]) *auto*
show *?case*
apply (*subst D*[*symmetric*])
apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-o.bj*)
apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-bj.backtrack*)
apply (*rule* $cdcl_W$*-restart-mset.backtrack-rule*[*of - L* ‹*remove1-mset L D*› *K M1 M2* ‹*remove1-mset L D'*› *i*])
subgoal using *LD* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *decomp* **by** (*simp add*: *trail.simps*)

subgoal using *lev-L* **by** (*simp add*: $cdcl_W$*-restart-mset-state*; *fail*)
subgoal using *max-lev L-D′* **by** (*simp add*: $cdcl_W$*-restart-mset-state get-maximum-level-add-mset*)
subgoal using *i* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
subgoal using *lev-K i* **unfolding** *D′* **by** (*simp add*: *trail.simps*)
subgoal using *D′-D* **by** (*simp add*: *mset-le-subtract*)
subgoal using *NU-D′ L-D′* **by** (*simp add*: *mset-le-subtract clauses-def ac-simps*)
subgoal
using *decomp* **unfolding** *state-eq-def state-def prod.inject*
using *i lev-K count-M1 L-D′* **by** (*simp add*: $cdcl_W$*-restart-mset-state D*)
done
qed

lemma *cdcl-twl-cp-*$cdcl_W$*-stgy*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹
$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* (*state*$_W$*-of S*) (*state*$_W$*-of T*) ∨
(*state*$_W$*-of S* = *state*$_W$*-of T* ∧ (*literals-to-update-measure T*, *literals-to-update-measure S*)
∈ *lexn less-than 2*)›
by (*auto dest*!: *twl-cp-propagate-or-conflict*
$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.conflict′*
$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.propagate′*
simp: *twl-struct-invs-def*)

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

lemma *cdcl-twl-cp-entailed-clss-inv*:
‹*cdcl-twl-cp S T* ⟹ *entailed-clss-inv S* ⟹ *entailed-clss-inv T*›
proof (*induction rule*: *cdcl-twl-cp.induct*)
case (*pop M N U NE UE L Q*)
then show *?case* **by** *auto*
next
case (*propagate D L L′ M N U NE UE WS Q*) **note** *undef* = *this*(*2*) **and** - = *this*
then have *unit*: ‹*entailed-clss-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *D*) *WS*, *Q*)›
by *auto*
show *?case*
unfolding *entailed-clss-inv.simps Ball-def*
proof (*intro allI impI conjI*)
fix *C*
assume ‹*C* ∈# *NE* + *UE*›
then obtain *L* **where**
C: ‹*L* ∈# *C*› **and** *lev-L*: ‹*get-level M L* = *0*› **and** *L-M*: ‹*L* ∈ *lits-of-l M*›
using *unit* **by** *auto*
have ‹*atm-of L′* ≠ *atm-of L*›
using *undef L-M* **by** (*auto simp*: *defined-lit-map lits-of-def*)
then show ‹∃*L*. *L* ∈# *C* ∧ (*None* = *None* ∨ *0* < *count-decided* (*Propagated L′* (*clause D*) # *M*)
⟶
get-level (*Propagated L′* (*clause D*) # *M*) *L* = *0* ∧
L ∈ *lits-of-l* (*Propagated L′* (*clause D*) # *M*))›
using *lev-L L-M C* **by** *auto*
qed
next
case (*conflict D L L′ M N U NE UE WS Q*)
then show *?case* **by** *auto*
next

case (*delete-from-working D L L' M N U NE UE WS Q*)
then show *?case* by *auto*
next
case (*update-clause D L L' M K N' U' N U NE UE WS Q*)
then show *?case* by *auto*
qed

lemma *cdcl-twl-cp-init-clss*:
‹*cdcl-twl-cp S T* $\Longrightarrow$ *twl-struct-invs S* $\Longrightarrow$ *init-clss* (*state*$_W$*-of T*) = *init-clss* (*state*$_W$*-of S*)›
by (*metis cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-no-more-init-clss cdcl-twl-cp-cdcl*$_W$*-stgy*)

lemma *cdcl-twl-cp-twl-struct-invs*:
‹*cdcl-twl-cp S T* $\Longrightarrow$ *twl-struct-invs S* $\Longrightarrow$ *twl-struct-invs T*›
apply (*subst twl-struct-invs-def*)
apply (*intro conjI*)
subgoal by (*rule twl-cp-twl-inv*; *auto simp add*: *twl-struct-invs-def twl-cp-twl-inv*)
subgoal by (*simp add*: *twl-cp-valid twl-struct-invs-def*)
subgoal by (*metis cdcl-twl-cp-cdcl*$_W$*-stgy cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-cdcl*$_W$*-all-struct-inv twl-struct-invs-def*)
subgoal by (*metis cdcl-twl-cp-cdcl*$_W$*-stgy twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-no-smaller-propa*)
subgoal by (*rule twl-cp-twl-st-exception-inv*; *auto simp add*: *twl-struct-invs-def*; *fail*)
subgoal by (*use twl-struct-invs-def twl-cp-no-duplicate-queued* **in** *blast*)
subgoal by (*rule twl-cp-distinct-queued*; *auto simp add*: *twl-struct-invs-def*)
subgoal by (*rule twl-cp-confl-cands-enqueued*; *auto simp add*: *twl-struct-invs-def*; *fail*)
subgoal by (*rule twl-cp-propa-cands-enqueued*; *auto simp add*: *twl-struct-invs-def*; *fail*)
subgoal by (*simp add*: *cdcl-twl-cp-conflict*; *fail*)
subgoal by (*simp add*: *cdcl-twl-cp-entailed-clss-inv twl-struct-invs-def*; *fail*)
subgoal by (*simp add*: *twl-struct-invs-def twl-cp-clauses-to-update*; *fail*)
subgoal by (*simp add*: *twl-cp-past-invs twl-struct-invs-def*; *fail*)
done

lemma *twl-struct-invs-no-false-clause*:
assumes ‹*twl-struct-invs S*›
shows ‹*cdcl*$_W$*-restart-mset.no-false-clause* (*state*$_W$*-of S*)›
proof –
obtain *M N U D NE UE WS Q* **where**
S: ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)›
by (*cases S*) *auto*
have *wf*: ‹$\bigwedge C.\ C \in\#\ N + U \Longrightarrow$ *struct-wf-twl-cls C*› **and** *entailed*: ‹*entailed-clss-inv S*›
using *assms* **unfolding** *twl-struct-invs-def twl-st-inv.simps S* **by** *fast+*
have ‹$\{\#\} \notin\#\ NE + UE$›
using *entailed* **unfolding** *S entailed-clss-inv.simps*
by (*auto simp del*: *set-mset-union*)
moreover have ‹*clause C* = {#} $\Longrightarrow C \in\#\ N + U \Longrightarrow$ *False*› **for** *C*
using *wf*[*of C*] **by** (*cases C*) (*auto simp del*: *set-mset-union*)
ultimately show *?thesis*
by (*fastforce simp*: *S clauses-def cdcl*$_W$*-restart-mset.no-false-clause-def*)
qed

lemma *cdcl-twl-cp-twl-stgy-invs*:
‹*cdcl-twl-cp S T* $\Longrightarrow$ *twl-struct-invs S* $\Longrightarrow$ *twl-stgy-invs S* $\Longrightarrow$ *twl-stgy-invs T*›
using *cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-cdcl*$_W$*-stgy-invariant*[*of* ‹*state*$_W$*-of S*› ‹*state*$_W$*-of S*›]
unfolding *twl-stgy-invs-def*
by (*metis cdcl*$_W$*-restart-mset.cdcl*$_W$*-restart-conflict-non-zero-unless-level-0*

cdcl$_W$*-restart-mset.cdcl*$_W$*-stgy-cdcl*$_W$*-stgy-invariant*
cdcl-twl-cp-cdcl$_W$*-stgy cdcl*$_W$*-restart-mset.conflict*
cdcl$_W$*-restart-mset.propagate twl-cp-propagate-or-conflict*
twl-struct-invs-def twl-struct-invs-no-false-clause)

The other rules

lemma
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-struct-invs S*›
shows
cdcl-twl-o-twl-st-inv: ‹*twl-st-inv T*› **and**
cdcl-twl-o-past-invs: ‹*past-invs T*›
using *cdcl twl*
proof (*induction rule*: *cdcl-twl-o.induct*)
case (*decide M K N NE U UE*) **note** *undef* = *this*(*1*) **and** *atm* = *this*(*2*)

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

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

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

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

have *H*: *False* **if**
W: ‹*L* ∈# *W*› **and**
uL: ‹− *L* ∈ *lits-of-l* (*Decided K* # *M*)› **and**
L': ‹¬*has-blit* (*Decided K* # *M*) (*W* + *UW*) *L*› **and**
False: ‹−*L* ≠ *K*› **for** *L*
proof −
have *H*: ‹− *L* ∈ *lits-of-l M* ⟹ ¬ *has-blit M* (*W* + *UW*) *L* ⟹ *get-level M L* = *count-decided M* ›
using *watched W* **unfolding** *C-W*
by *auto*

obtain L′ where W′: ‹W = {#L, L′#}›
using struct W size-2-iff[of W] unfolding C-W
by (auto simp: add-mset-eq-single add-mset-eq-add-mset dest!: multi-member-split)
have no-has-blit: ‹¬has-blit M (W + UW) L›
using no-has-blit-decide′[of K M C] L′ n-d C-W W undef by auto
then have ‹∀K ∈# UW. −K ∈ lits-of-l M›
using uL L′ False excep C W C-W L′ W n-d undef
by (auto simp: twl-exception-inv.simps all-conj-distrib
dest!: multi-member-split[of - N])
then have M-CNot-C: ‹M ⊨as CNot (remove1-mset L′ (clause C))›
using uL False W′ unfolding true-annots-true-cls-def-iff-negation-in-model
by (auto simp: C-W W)
moreover have L′-C: ‹L′ ∈# clause C›
unfolding C-W W′ by auto
ultimately have ‹defined-lit M L′›
using propa-cands C by auto

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

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

{
assume exception: ‹¬ twl-is-an-exception C {#−K#} {#}›
have ‹twl-lazy-update M C›
using C inv by (auto simp: twl-st-inv.simps)
have lev-le-Suc: ‹get-level M Ka ≤ Suc (count-decided M)› for Ka
using count-decided-ge-get-level le-Suc-eq by blast
show ‹twl-lazy-update (Decided K # M) C›
unfolding C-W twl-lazy-update.simps Ball-def
proof (intro allI impI)
fix L K′ :: ‹′a literal›
assume
W: ‹L ∈# W› and
uL: ‹− L ∈ lits-of-l (Decided K # M)› and

L': ‹¬*has-blit* (*Decided K* # *M*) (*W* + *UW*) *L*› **and**
K': ‹$K' \in\#$ *UW*›
then have ‹$-L = K$›
using *H*[*OF W uL L'*] **by** *fast*
then have *False*
using *exception W*
by (*auto simp*: *C-W twl-is-an-exception-def*)
then show ‹*get-level* (*Decided K* # *M*) $K' \leq$ *get-level* (*Decided K* # *M*) $L \wedge$
$-K' \in$ *lits-of-l* (*Decided K* # *M*)›
by *fast*
qed
}
qed

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

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

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

have ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *excep inv C* **unfolding** *twl-st-inv.simps* **by** *auto*
then show ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *IH M* **by** *blast*
next
fix *M1 M2* :: ‹('*a*, '*a clause*) *ann-lits*› **and** K'
assume ‹*Decided K* # *M* = *M2* @ *Decided* K' # *M1*›
then have *M*: ‹*M* = *tl M2* @ *Decided* K' # *M1* ∨ *M* = *M1*›
by (*cases M2*) *auto*
then show ‹*confl-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})› **and**
‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})› **and**
‹*clauses-to-update-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})›
using *confl-cands past propa-cands w-q* **unfolding** *past-invs.simps* **by** *blast+*
qed
next
case (*skip L D* C' *M N U NE UE*)
case *1*
then show *?case*
by (*auto simp*: *twl-st-inv.simps twl-struct-invs-def*)

case *2*
then show *?case*
  by (*auto simp*: *past-invs.simps twl-struct-invs-def*)
next
  case (*resolve L D C M N U NE UE*)
  case *1*
  then show *?case*
    by (*auto simp*: *twl-st-inv.simps twl-struct-invs-def*)
  case *2*
  then show *?case*
    by (*auto simp*: *past-invs.simps twl-struct-invs-def*)
next
  case (*backtrack-unit-clause K′ D K M1 M2 M D′ i N U NE UE*) **note** *decomp* = *this*(*2*) **and**
    lev = *this*(*3*−*5*)

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

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

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

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

  have *propa-cands-M1*:
    ‹*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, {#}, {#− *K′*#})›
    unfolding *propa-cands-enqueued.simps*
  proof (*intro allI impI*)
    fix *L C*
    assume
      C: ‹*C* ∈# *N* + *U*› **and**

L: ‹*L* ∈# *clause C*› **and**
M1-CNot: ‹*M1* ⊨*as CNot* (*remove1-mset L* (*clause C*))› **and**
undef: ‹*undefined-lit M1 L*›
define *D* **where** ‹*D* = *remove1-mset L* (*clause C*)›
have ‹*add-mset L D* ∈# *clause* '# (*N* + *U*)› **and** ‹*M1* ⊨*as CNot D*›
using *C L M1-CNot* **unfolding** *D-def* **by** *auto*
moreover have ‹$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of ?S*)›
using *invs* **unfolding** *twl-struct-invs-def* **by** *blast*
ultimately have *False*
using *undef M′*
by (*fastforce simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def trail.simps clauses-def*)
then show ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# {#− *K′*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})›
by *fast*
qed

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

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

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

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

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

}

have ⟨*watched-literals-false-of-max-level M1 C*⟩
using *past C* **unfolding** *M′ past-invs.simps* **by** *blast*
then show ⟨*watched-literals-false-of-max-level ?M1 C*⟩
using *has-blit-Cons n-d′*
by (*auto simp*: *C-W get-level-cons-if*)
qed
case *2*
show *?case*
unfolding *past-invs.simps Ball-def*
proof (*intro allI impI conjI*)
fix *M1″ M2″ K″ C*
assume ⟨*?M1 = M2″ @ Decided K″ # M1″*⟩ **and** *C*: ⟨*C* ∈# *N + U*⟩
then have *M1*: ⟨*M1 = tl M2″ @ Decided K″ # M1″*⟩
by (*cases M2″*) *auto*
have ⟨*twl-lazy-update M1″ C*⟩⟨*watched-literals-false-of-max-level M1″ C*⟩
using *past C* **unfolding** *past-invs.simps M M1 twl-exception-inv.simps* **by** *auto*
moreover {
have ⟨*twl-exception-inv* (*M1″*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*⟩
using *past C* **unfolding** *past-invs.simps M M1* **by** *auto*
then have ⟨*twl-exception-inv* (*M1″*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, {#}, {#}) *C*⟩
using *C* **unfolding** *twl-exception-inv.simps* **by** *auto* }
ultimately show ⟨*twl-lazy-update M1″ C*⟩⟨*watched-literals-false-of-max-level M1″ C*⟩
⟨*twl-exception-inv* (*M1″*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, {#}, {#}) *C*⟩
by *fast+*
next
fix *M1″ M2″ K″*
assume ⟨*?M1 = M2″ @ Decided K″ # M1″*⟩
then have *M1*: ⟨*M1 = tl M2″ @ Decided K″ # M1″*⟩
by (*cases M2″*) *auto*
then show
⟨*confl-cands-enqueued* (*M1″*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, {#}, {#})⟩ **and**
⟨*propa-cands-enqueued* (*M1″*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, {#}, {#})⟩ **and**
⟨*clauses-to-update-inv* (*M1″*, *N*, *U*, *None*, *NE*, *add-mset* {#*K′*#} *UE*, {#}, {#})⟩
using *past* **by** (*auto simp add*: *past-invs.simps M*)
qed
next
case (*backtrack-nonunit-clause K′ D K M1 M2 M D′ i N U NE UE K″*) **note** *K′-D = this*(*1*) **and**
decomp = this(*2*) **and** *lev-K′ = this*(*3*) **and** *i = this*(*5*) **and** *lev-K = this*(*6*) **and** *K′-D′ = this*(*10*)
and *K″ = this*(*11*) **and** *lev-K″ = this*(*12*)
case *1* **note** *invs = this*(*1*)
let *?S* = ⟨(*M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})⟩
let *?M1* = ⟨*Propagated K′ D′ # M1*⟩
let *?T* = ⟨(*?M1*, *N*, *add-mset* (*TWL-Clause* {#*K′*, *K″*#} (*D′* − {#*K′*, *K″*#})) *U*, *None*, *NE*, *UE*,
{#},
{#− *K′*#})⟩
let *?D* = ⟨*TWL-Clause* {#*K′*, *K″*#} (*D′* − {#*K′*, *K″*#})⟩
have *bt-twl*: ⟨*cdcl-twl-o ?S ?T*⟩
using *cdcl-twl-o.backtrack-nonunit-clause*[*OF backtrack-nonunit-clause.hyps*] **.**
then have ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-o* ($state_W$*-of ?S*) ($state_W$*-of ?T*)⟩
by (*rule cdcl-twl-o-*$cdcl_W$*-o*) (*use invs* **in** ⟨*simp-all add*: *twl-struct-invs-def*⟩)
then have *struct-inv-T*: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of ?T*)⟩
using $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-restart-mset.other invs*
unfolding *twl-struct-invs-def* **by** *blast*
have *inv*: ⟨*twl-st-inv ?S*⟩ **and**

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

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

have ⟨*undefined-lit* (*M3 @ M2*) *K*⟩
  using *n-d* **unfolding** *M* **by** *auto*
then have *count-M1*: ⟨*count-decided M1 = i*⟩
  using *lev-K* **unfolding** *M* **by** (*auto simp*: *image-Un*)
then have *K''-ne-K*: ⟨$K' \neq K''$⟩
  using *lev-K lev-K' lev-K'' count-decided-ge-get-level*[*of M K''*] **unfolding** *M* **by** *auto*
then have *D*:
  ⟨*add-mset K'* (*add-mset K''* (*D'* − {#*K'*, *K''*#})) = *D'*⟩
  ⟨*add-mset K''* (*add-mset K'* (*D'* − {#*K'*, *K''*#})) = *D'*⟩
  using *K'' K'-D' multi-member-split* **by** *fastforce+*
have *propa-cands-M1*: ⟨*propa-cands-enqueued* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#− *K''*#})⟩
  unfolding *propa-cands-enqueued.simps*
proof (*intro allI impI*)
  fix *L C*
  assume
    C: ⟨*C* ∈# *N* + *U*⟩ **and**
    L: ⟨*L* ∈# *clause C*⟩ **and**
    M1-CNot: ⟨*M1* ⊨*as CNot* (*remove1-mset L* (*clause C*))⟩ **and**
    undef: ⟨*undefined-lit M1 L*⟩
  define *D* **where** ⟨*D = remove1-mset L* (*clause C*)⟩
  have ⟨*add-mset L D* ∈# *clause* '# (*N* + *U*)⟩ **and** ⟨*M1* ⊨*as CNot D*⟩
    using *C L M1-CNot* **unfolding** *D-def* **by** *auto*
  moreover have ⟨$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of ?S*)⟩
    using *invs* **unfolding** *twl-struct-invs-def* **by** *blast*
  ultimately have *False*
    using *undef M'*
    by (*fastforce simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def trail.simps clauses-def*)
  then show ⟨(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# {#− *K''*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})⟩
    by *fast*
qed
have ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting* ($state_W$*-of ?T*)⟩

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

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

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

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

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

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

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

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

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

case *2*

show *?case*
unfolding *past-invs.simps Ball-def*
proof (*intro allI impI conjI*)
fix *M1'' M2'' K''' C*
assume *M1*: ‹*?M1 = M2'' @ Decided K''' # M1''*› **and** *C*: ‹*C ∈# N + add-mset ?D U*›
then have *M1*: ‹*M1 = tl M2'' @ Decided K''' # M1''*›
by (*cases M2''*) *auto*
have ‹*twl-lazy-update M1'' C*›‹*watched-literals-false-of-max-level M1'' C*›
if ‹*C ≠ ?D*›
using *past C that* **unfolding** *past-invs.simps M M1 twl-exception-inv.simps* **by** *auto*
moreover {
have ‹*twl-exception-inv* (*M1''*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*› **if** ‹*C ≠ ?D*›
using *past C* **unfolding** *past-invs.simps M M1* **by** (*auto simp*: *that*)
then have ‹*twl-exception-inv* (*M1''*, *N*, *add-mset ?D U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
if ‹*C ≠ ?D*›
using *C* **unfolding** *twl-exception-inv.simps* **by** (*auto simp*: *that*) **}**
moreover {
have *n-d-M1*: ‹*no-dup ?M1*›
using *struct-inv-T* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)
then have ‹*undefined-lit M1'' K'*›
unfolding *M1* **by** *auto*
moreover {
have ‹*− K'' ∉ lits-of-l M1''*›
proof (*rule ccontr*)
assume ‹*¬ − K'' ∉ lits-of-l M1''*›
then have ‹*undefined-lit* (*tl M2'' @ Decided K''' #* []) *K''*›

using *n-d-M1* **unfolding** *M1* **by** (*auto simp*: *atm-lit-of-set-lits-of-l*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
defined-lit-map atm-of-eq-atm-of image-Un
dest: *no-dup-uminus-append-in-atm-notin*)
then show *False*
using *lev-M1-K''* *count-decided-ge-get-level*[*of M1'' K''*] **unfolding** *M1*
by (*auto simp*: *image-Un Int-Un-distrib*)
qed }
ultimately have ‹*twl-lazy-update M1'' ?D*› **and**
‹*watched-literals-false-of-max-level M1'' ?D*› **and**
‹*twl-exception-inv* (*M1''*, *N*, *add-mset* (*TWL-Clause* {#*K'*, *K''*#} (*D'* − {#*K'*, *K''*#})) *U*, *None*,
NE, *UE*, {#}, {#}) *?D*›
by (*auto simp*: *add-mset-eq-add-mset twl-exception-inv.simps get-level-cons-if*
Decided-Propagated-in-iff-in-lits-of-l) **}**
ultimately show ‹*twl-lazy-update M1'' C*›
‹*watched-literals-false-of-max-level M1'' C*›
‹*twl-exception-inv* (*M1''*, *N*, *add-mset* (*TWL-Clause* {#*K'*, *K''*#} (*D'* − {#*K'*, *K''*#})) *U*, *None*,
NE, *UE*, {#}, {#}) *C*›
by *blast+*
next
fix *M1'' M2'' K'''*
assume *M1*: ‹*?M1 = M2'' @ Decided K''' # M1''*›
then have *M1*: ‹*M1 = tl M2'' @ Decided K''' # M1''*›
by (*cases M2''*) *auto*
then have *confl-cands*: ‹*confl-cands-enqueued* (*M1''*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})› **and**
propa-cands: ‹*propa-cands-enqueued* (*M1''*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})› **and**
w-q: ‹*clauses-to-update-inv* (*M1''*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})›

using *past* **by** (*auto simp add*: *M M1 past-invs.simps simp del*: *propa-cands-enqueued.simps confl-cands-enqueued.simps*)
have *uK″-M1″*: ⟨− *K″* ∉ *lits-of-l M1″*⟩
proof (*rule ccontr*)
assume *K″-M1″*: ⟨¬ *?thesis*⟩
have ⟨*undefined-lit* (*tl M2″* @ *Decided K‴* # []) (−*K″*)⟩
apply (*rule no-dup-append-in-atm-notin*)
prefer *2* **using** *K″-M1″* **apply** (*simp*; *fail*)
by (*use n-d* **in** ⟨*auto simp*: *M M1 no-dup-def*; *fail*⟩)[]
then show *False*
using *lev-M1-K″ count-decided-ge-get-level*[*of M1″ K″*] **unfolding** *M M1*
by (*auto simp*: *image-Un*)
qed
have *uK′-M1″*: ⟨− *K′* ∉ *lits-of-l M1″*⟩
proof (*rule ccontr*)
assume *K′-M1″*: ⟨¬ *?thesis*⟩
have ⟨*undefined-lit* (*M3* @ *M2* @ *Decided K* # *tl M2″* @ *Decided K‴* # []) (−*K′*)⟩
apply (*rule no-dup-append-in-atm-notin*)
prefer *2* **using** *K′-M1″* **apply** (*simp*; *fail*)
by (*use n-d* **in** ⟨*auto simp*: *M M1*; *fail*⟩)[]
then show *False*
using *lev-K′ count-decided-ge-get-level*[*of M1″ K′*] **unfolding** *M M1*
by (*auto simp*: *image-Un*)
qed

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

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

lemma *cdcl-twl-o-twl-st-exception-inv*:
assumes
cdcl: ⟨*cdcl-twl-o S T*⟩ **and**
twl: ⟨*twl-struct-invs S*⟩
shows
⟨*twl-st-exception-inv T*⟩
using *cdcl twl*
proof (*induction rule*: *cdcl-twl-o.induct*)
case (*decide M L N U NE UE*) **note** *undef* = *this*(*1*) **and** *in-atms* = *this*(*2*) **and** *twl* = *this*(*3*)
then have *excep*: ⟨*twl-st-exception-inv* (*M*, *N*, *NE*, *None*, *U*, *UE*, {#}, {#})⟩

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

have *struct-inv-U*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of ?U*)›
using *cdcl-twl-o-*$cdcl_W$*-o*[*OF cdcl-twl-o.backtrack-unit-clause*[*OF backtrack-unit-clause.hyps*]
‹*twl-st-inv ?S*› *valid struct-inv-T*]
$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-inv* $cdcl_W$*-restart-mset.*$cdcl_W$*-restart.intros*(*3*)
struct-inv-T **by** *blast*
then have *undef*: ‹*undefined-lit M1 L*›
unfolding *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*simp add*: $cdcl_W$*-restart-mset-state*)

show *?case*
using *n-d excep undef*
unfolding *twl-struct-invs-def*
by (*auto simp*: *twl-exception-inv.simps dest*!: *no-has-blit-propagate′*)
next

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

have *struct-inv-U*: ‹cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv (state$_W$-of ?U)›
using cdcl-twl-o-cdcl$_W$-o[*OF cdcl-twl-o.backtrack-nonunit-clause*[*OF backtrack-nonunit-clause.hyps*]
‹*twl-st-inv ?S*› *valid struct-inv-T*]
cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-inv cdcl$_W$-restart-mset.cdcl$_W$-restart.intros(3)
struct-inv-T **by** *blast*
then have *undef*: ‹*undefined-lit M1 L*›
unfolding *twl-struct-invs-def* cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def **by** (*simp add*: cdcl$_W$-restart-mset-state)

have *n-d*: ‹*no-dup* (*Propagated L D'* # *M1*)›
using *struct-inv-U* **unfolding** cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def
by (*simp add*: *trail.simps*)
have ‹*i* = *count-decided M1*›
using *decomp lev-K n-d-M* **by** (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *get-level-append-if get-level-cons-if*
split: *if-splits*)
then have *lev-L'-M1*: ‹*get-level* (*Propagated L D'* # *M1*) *L'* = *count-decided M1*›
using *decomp lev-L' n-d-M* **by** (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *get-level-append-if get-level-cons-if*
split: *if-splits*)
have ‹− *L* ∉ *lits-of-l M1*›
using *n-d* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
moreover have ‹*has-blit* (*Propagated L D'* # *M1*) (*add-mset L* (*add-mset L'* (*D'* − {#*L*, *L'*#}))) *L'*›
unfolding *has-blit-def*
apply (*rule exI*[*of* - *L*])
using *lev-L' lev-L'-M1*
by *auto*
ultimately show *?case*
using *n-d excep undef*
unfolding *twl-struct-invs-def*
by (*auto simp*: *twl-exception-inv.simps dest*!: *no-has-blit-propagate'*)
qed

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

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

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

have *struct-C*: ⟨*struct-wf-twl-cls C*⟩
using *twl-st-inv C* **unfolding** *twl-st-inv.simps* **by** *blast*
then have *dist-C*: ⟨*distinct-mset* (*clause C*)⟩
by (*cases C*) *auto*
obtain *W UW K K′* **where**
C-W: ⟨*C* = *TWL-Clause W UW*⟩ **and**
W: ⟨*W* = {#*K*, *K′*#}⟩
using *struct-C* **by** (*cases C*) (*auto simp*: *size-2-iff*)

have ⟨¬*M* ⊨*as CNot* (*clause C*)⟩
using *confl-cand C* **by** *auto*
then have *uL-C*: ⟨−*L* ∈# *clause C*⟩ **and** *neg-C*: ⟨∀ *K* ∈# *clause C*. −*K* ∈ *lits-of-l* (*Decided L* #
M)⟩
using *LM-C* **unfolding** *true-annots-true-cls-def-iff-negation-in-model* **by** *auto*
have ⟨*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*⟩
using *excep C* **by** *auto*
then have *H*: ⟨*L* ∈# *watched* (*TWL-Clause* {#*K*, *K′*#} *UW*) ⟶
− *L* ∈ *lits-of-l M* ⟶ ¬ *has-blit M* (*clause* (*TWL-Clause* {#*K*, *K′*#} *UW*)) *L* ⟶
L ∉# {#} ⟶
(*L*, *TWL-Clause* {#*K*, *K′*#} *UW*) ∉# {#} ⟶
(∀ *K*∈#*unwatched* (*TWL-Clause* {#*K*, *K′*#} *UW*).

$-$ $K \in$ *lits-of-l* M)› **for** L
unfolding *twl-exception-inv.simps* C-W W **by** *blast*
have *excep*: ‹$L \in\#$ *watched* (*TWL-Clause* {#K, K'#} *UW*) $\longrightarrow$
$-$ $L \in$ *lits-of-l* M $\longrightarrow$ $\neg$ *has-blit* M (*clause* (*TWL-Clause* {#K, K'#} *UW*)) L $\longrightarrow$
($\forall$ $K\in\#$*unwatched* (*TWL-Clause* {#K, K'#} *UW*). $-$ $K \in$ *lits-of-l* M)› **for** L
using H[*of* L] **by** *simp*
have ‹$-L \in\#$ *watched* C›
proof (*rule ccontr*)
assume *uL-W*: ‹$-L \notin\#$ *watched* C›
then have *uL-UW*: ‹$-L \in\#$ *UW*›
using *uL-C* **unfolding** *C-W* **by** *auto*
have ‹$K \neq -L \vee K' \neq -L$›
using *dist-C C-W W* **by** *auto*
moreover have ‹$K \notin$ *lits-of-l* M› **and** ‹$K' \notin$ *lits-of-l* M› **and** *L-M*: ‹$L \notin$ *lits-of-l* M›
using *neg-C uL-W n-d* **unfolding** *C-W W* **by** (*auto simp*: *lits-of-def uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus Decided-Propagated-in-iff-in-lits-of-l)
ultimately have *disj*: ‹($-K \in$ *lits-of-l* M $\wedge$ $K' \notin$ *lits-of-l* M) $\vee$
($-K' \in$ *lits-of-l* M $\wedge$ $K \notin$ *lits-of-l* M)›
using *neg-C* **by** (*auto simp*: *C-W W*)
have ‹$\neg$*has-blit* M (*clause* C) K›
using ‹$K \notin$ *lits-of-l* M› ‹$K' \notin$ *lits-of-l* M›
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **by** (*auto dest*!: *multi-member-split*
dest!: *no-dup-consistentD*
dest!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹$\neg$*has-blit* M (*clause* C) K'›
using ‹$K' \notin$ *lits-of-l* M› ‹ $K \notin$ *lits-of-l* M›
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **by** (*auto dest*!: *multi-member-split*
dest!: *no-dup-consistentD*
dest!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
ultimately have ‹$\forall$ $K \in\#$ *unwatched* C. $-K \in$ *lits-of-l* M›
apply $-$
apply (*rule disjE*[*OF disj*])
subgoal
using *excep*[*of* K]
unfolding *C-W twl-clause.sel member-add-mset W*
by *auto*
subgoal
using *excep*[*of* K']
unfolding *C-W twl-clause.sel member-add-mset W*
by *auto*
done
then show *False*
using *uL-W uL-C L-M* **unfolding** *C-W W* **by** *auto*
qed
then show ‹($\exists$ L'. $L' \in\#$ *watched* C $\wedge$ $L' \in\#$ {#$-$ L#}) $\vee$ ($\exists$ L. (L, C) $\in\#$ {#})›
by *auto*
qed

case *2*
show *?case*
unfolding *propa-cands-enqueued.simps Ball-def*
proof (*intro allI impI*)
fix *FK* C
assume
C: ‹$C \in\#$ N + U› **and**
K: ‹$FK \in\#$ *clause* C› **and**

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

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

have ‹−*L* ∈# *watched C*›
proof (*rule ccontr*)
assume *uL-W*: ‹−*L* ∉# *watched C*›
then obtain *W UW K K′* **where**
C-W: ‹*C* = *TWL-Clause W UW*› **and**
W: ‹*W* = {#*K*, *K′*#}› **and**
uK-M: ‹−*K* ∈ *lits-of-l M*›
using *struct-C neg-C* **by** (*cases C*) (*auto simp*: *size-2-iff remove1-mset-add-mset-If*
add-mset-commute split: *if-splits*)
have *FK-F*: ‹*FK* ≠ *K*›
using *Decided-Propagated-in-iff-in-lits-of-l uK-M undef-M-K* **by** *blast*
have *L-M*: ‹*undefined-lit M L*›
using *neg-C uL-W n-d* **unfolding** *C-W W* **by** *auto*
then have ‹*K* ≠ −*L*›
using *uK-M* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
moreover have ‹*K* ∉ *lits-of-l M*›
using *neg-C uL-W n-d uK-M* **by** (*auto simp*: *lits-of-def uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus)
ultimately have ‹*K′* ∉ *lits-of-l M*›
apply (*cases* ‹*K′* = *FK*›)
using *Decided-Propagated-in-iff-in-lits-of-l undef-M-K* **apply** *blast*
using *neg-C C-W W FK-F n-d uL-W* **by** (*auto simp add*: *remove1-mset-add-mset-If uminus-lit-swap*
lits-of-def no-dup-cannot-not-lit-and-uminus)
moreover have ‹*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *excep C* **by** *auto*

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

then show *False*
using *C-W L-M(1)* ‹− *L* ∈# *clause C*› *uL-W*
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
then show ‹(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# {#− *L*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})›
by *auto*
qed

case *3*
show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty L C*)
then show *?case* **by** *simp*
next
case (*WS-empty K*)
then show *?case*
using *w-q n-d* **unfolding** *clauses-to-update-prop.simps*
by (*auto simp add*: *filter-mset-empty-conv*
dest!: *no-has-blit-decide'*)
next
case (*Q K C*)
then show *?case*
using *w-q n-d* **by** (*auto dest*!: *no-has-blit-decide'*)
qed
next
case (*skip L D C' M N U NE UE*)
case *1* **then show** *?case* **by** *auto*
case *2* **then show** *?case* **by** *auto*
case *3* **then show** *?case* **by** *auto*
next
case (*resolve L D C M N U NE UE*)
case *1* **then show** *?case* **by** *auto*
case *2* **then show** *?case* **by** *auto*
case *3* **then show** *?case* **by** *auto*
next
case (*backtrack-unit-clause L D K M1 M2 M D' i N U NE UE*) **note** *decomp* = *this*(*2*)
let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})›
let *?U* = ‹(*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, {#}, {#− *L*#})›
obtain *M3* **where**
M: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** *blast*

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

have *n-d*: ‹*no-dup M*›
using *struct-inv* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*

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

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

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

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

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

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

using *f2 neg-C* **unfolding** *C-W W* **by** (*metis* (*no-types*) *C-W W add-diff-cancel-right′*
atm-of-eq-atm-of clause.simps
consistent-interp-def in-diffD insertE list.simps(*15*) *lits-of-insert uL-C*
union-single-eq-member Decided-Propagated-in-iff-in-lits-of-l)
show ‹*K′* ∉ *lits-of-l M1*›
using *consistent-interp-def distinct-consistent-interp n-d-L-M1*
using *neg-C uL-W n-d* **unfolding** *C-W W* **by** *auto*
show ‹*L* ∉ *lits-of-l M1*›
using *undef-L* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
ultimately have ‹(−*K* ∈ *lits-of-l M1* ∧ *K′* ∉ *lits-of-l M1*) ∨
(−*K′* ∈ *lits-of-l M1* ∧ *K* ∉ *lits-of-l M1*)›
using *neg-C* **by** (*auto simp*: *C-W W*)
moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *excep-M1 C* **by** *auto*
have ‹¬*has-blit M1* (*clause C*) *K*›
using ‹*K* ∉ *lits-of-l M1*› ‹*K′* ∉ *lits-of-l M1*› ‹*L* ∉ *lits-of-l M1*› *uL-M1*
n-d-L-M1 no-dup-cons
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto dest*!: *multi-member-split*
dest!: *no-dup-consistentD*[*OF n-d-L-M1*]
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
using *n-d-L-M1 no-dup-cons no-dup-consistentD* **by** *blast*
moreover have ‹¬*has-blit M1* (*clause C*) *K′*›
using ‹*K′* ∉ *lits-of-l M1*› ‹ *K* ∉ *lits-of-l M1*› ‹*L* ∉ *lits-of-l M1*› *uL-M1*
n-d-L-M1 no-dup-cons no-dup-consistentD
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto 10 10 dest*!: *multi-member-split*
dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
using *n-d-L-M1 no-dup-cons no-dup-consistentD* **by** *auto*
ultimately have ‹∀ *K* ∈# *unwatched C*. −*K* ∈ *lits-of-l M1*›
using *C twl-clause.sel*(*1*) *union-single-eq-member w-q*
by (*fastforce simp*: *twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib L-M*)
then show *False*
using *uL-W uL-C L-M K-L uL-M1* **unfolding** *C-W W* **by** *auto*
qed
then show ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# {#− *L*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})›
by *auto*
qed
case *2*
then show *?case*
unfolding *propa-cands-enqueued.simps Ball-def*
proof (*intro allI impI*)
fix *FK C*
assume
C: ‹*C* ∈# *N* + *U*› **and**
K: ‹*FK* ∈# *clause C*› **and**
LM-C: ‹*Propagated L* {#*L*#} # *M1* ⊨*as CNot* (*remove1-mset FK* (*clause C*))› **and**
undef: ‹*undefined-lit* (*Propagated L* {#*L*#} # *M1*) *FK*›
have *undef-M-K*: ‹*undefined-lit* (*Propagated L D* # *M1*) *FK*›
using *undef* **by** (*auto simp*: *defined-lit-map*)
then have ‹¬ *M1* ⊨*as CNot* (*remove1-mset FK* (*clause C*))›
using *propa-cands C K undef* **by** (*auto simp*: *defined-lit-map*)
then have *uL-C*: ‹−*L* ∈# *clause C*› **and**
neg-C: ‹∀ *K* ∈# *remove1-mset FK* (*clause C*). −*K* ∈ *lits-of-l* (*Propagated L D* # *M1*)›
using *LM-C undef-M-K* **by** (*force simp*: *true-annots-true-cls-def-iff-negation-in-model*
dest: *in-diffD*)+

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

moreover have ‹$-L$ ∈# *watched C*›
proof (*rule ccontr*)
  assume *uL-W*: ‹$-L$ ∉# *watched C*›
  then obtain *W UW K K'* **where**
    C-W: ‹*C* = *TWL-Clause W UW*› **and**
    W: ‹*W* = {#*K*, *K'*#}› **and**
    uK-M: ‹$-K$ ∈ *lits-of-l M1*›
    using *struct-C neg-C* **by** (*cases C*) (*auto simp*: *size-2-iff remove1-mset-add-mset-If*
      add-mset-commute split: *if-splits*)
  have ‹*K* ∉ *lits-of-l M1*› **and** *L-M*: ‹*L* ∉ *lits-of-l M1*›
  proof −
    have *f2*: ‹*consistent-interp* (*lits-of-l M1*)›
      using *distinct-consistent-interp n-d-L-M1* **by** *auto*
    have *undef-L*: ‹*undefined-lit M1 L*›
      using *atm-lit-of-set-lits-of-l n-d-L-M1* **by** *force*
    then show ‹*K* ∉ *lits-of-l M1*›
      using *f2 neg-C* **unfolding** *C-W W*
      using *n-d-L-M1 no-dup-cons no-dup-consistentD uK-M* **by** *blast*
    show ‹*L* ∉ *lits-of-l M1*›
      using *undef-L* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
  qed
  have *FK-F*: ‹*FK* ≠ *K*›
    using *uK-M undef-M-K* **unfolding** *Decided-Propagated-in-iff-in-lits-of-l* **by** *auto*
  have ‹*K* ≠ $-L$›
    using *uK-M uL-M1* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
  moreover have ‹*K* ∉ *lits-of-l M1*›
    using *neg-C uL-W n-d uK-M n-d-L-M1* **by** (*auto simp*: *lits-of-def uminus-lit-swap*
      no-dup-cannot-not-lit-and-uminus dest: *no-dup-cannot-not-lit-and-uminus*)
  ultimately have ‹*K'* ∉ *lits-of-l M1*›
    apply (*cases* ‹*K'* = *FK*›)
    using *undef-M-K* **apply** (*force simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
    using *neg-C C-W W FK-F n-d uL-W n-d-L-M1* **by** (*auto simp add*: *remove1-mset-add-mset-If*
      uminus-lit-swap lits-of-def no-dup-cannot-not-lit-and-uminus
      dest: *no-dup-cannot-not-lit-and-uminus*)
  moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
    using *excep-M1 C* **by** *auto*
  moreover have ‹¬*has-blit M1* (*clause C*) *K*›
    using ‹*K* ∉ *lits-of-l M1*› ‹*K'* ∉ *lits-of-l M1*› ‹*L* ∉ *lits-of-l M1*› *uL-M1*
     n-d-L-M1 no-dup-cons K undef
    using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto dest*!: *multi-member-split*
      dest!: *no-dup-consistentD*[*OF n-d-L-M1*]
      dest!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
    by (*smt add-mset-commute add-mset-eq-add-mset defined-lit-uminus in-lits-of-l-defined-litD*
      insert-DiffM no-dup-consistentD set-subset-Cons true-annot-mono true-annot-singleton)+
  moreover have ‹¬*has-blit M1* (*clause C*) *K'*›
    using ‹*K'* ∉ *lits-of-l M1*› ‹ *K* ∉ *lits-of-l M1*› ‹*L* ∉ *lits-of-l M1*› *uL-M1*
    n-d-L-M1 no-dup-cons no-dup-consistentD K undef
    using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto 10 10 dest*!: *multi-member-split*
      dest!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
    by (*smt add-mset-commute add-mset-eq-add-mset defined-lit-uminus in-lits-of-l-defined-litD*
      insert-DiffM no-dup-consistentD set-subset-Cons true-annot-mono true-annot-singleton)+

ultimately have ‹∀ K ∈# *unwatched* C. $-K$ ∈ *lits-of-l M1*›
using *uK-M*
by (*auto simp*: *twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib*)
then show *False*
using *C-W uL-M1* ‹$-$ L ∈# *clause* C› *uL-W*
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
then show ‹(∃ L'. L' ∈# *watched* C ∧ L' ∈# {#$-$ L#}) ∨ (∃ L. (L, C) ∈# {#})›
by *auto*
qed

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

show *?case*
proof (*induction rule*: *clauses-to-update-inv-cases*)
case (*WS-nempty* L C)
then show *?case* **by** *simp*
next
case (*WS-empty* K)
then show *?case*
using *2*[*of* K] *n-d-L-M1*
apply (*simp only*: *filter-mset-empty-conv Ball-def image-mset-is-empty-iff*)
by (*auto simp add*: *clauses-to-update-prop.simps*)
next
case (Q K C)
then show *?case*
using *3*[*of* C K] *has-blit-Cons n-d-L-M1* **by** (*fastforce simp add*: *clauses-to-update-prop.simps*)
qed
next
case (*backtrack-nonunit-clause* L D K *M1 M2* M D' i N U *NE UE* L') **note** *LD* = *this*(*1*) **and** *decomp* = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *lev-max-L* = *this*(*4*) **and** i = *this*(*5*) **and** *lev-K* = *this*(*6*)
and LD' = *this*(*11*) **and** *lev-L'* = *this*(*12*)
let *?S* = ‹(M, N, U, *Some* D, *NE*, *UE*, {#}, {#})›
let *?D* = ‹*TWL-Clause* {#L, L'#} (D' $-$ {#L, L'#})›
let *?U* = ‹(*Propagated* L D' # *M1*, N, *add-mset ?D* U, *None*, *NE*,
UE, {#}, {#$-$ L#})›
obtain *M3* **where**
M: ‹M = *M3* @ *M2* @ *Decided* K # *M1*›
using *decomp* **by** *blast*

case *1*
then have *twl-st-inv*: ‹*twl-st-inv ?S*› **and**
struct-inv: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* ($state_W$*-of ?S*)› **and**
excep: ‹*twl-st-exception-inv ?S*› **and**
past: ‹*past-invs ?S*›
using *decomp* **unfolding** *twl-struct-invs-def* **by** *fast+*
then have
confl-cands: ‹*confl-cands-enqueued* (*M1*, N, U, *None*, *NE*, *UE*, {#}, {#})› **and**
propa-cands: ‹*propa-cands-enqueued* (*M1*, N, U, *None*, *NE*, *UE*, {#}, {#})› **and**

w-q: ‹*clauses-to-update-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})›
  using *decomp* **unfolding** *past-invs.simps* **by** *auto*

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

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

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

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

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

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

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

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

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

have ‹¬*M1* ⊨*as CNot* (*clause C*)›
using *confl-cands C* **by** *auto*
then have *uL-C*: ‹$-L \in\#$ *clause C*› **and** *neg-C*: ‹∀ $K \in\#$ *clause C*. $-K \in$ *lits-of-l* (*Decided L* # *M1*)›
using *LM-C* **unfolding** *true-annots-true-cls-def-iff-negation-in-model* **by** *auto*
have *K-L*: ‹$K \neq L$› **and** *K′-L*: ‹$K' \neq L$›
apply (*metis C-W LM-C W add-diff-cancel-right′ clause.simps consistent-interp-def distinct-consistent-interp in-CNot-implies-uminus*(*2*) *in-diffD n-d-L-M1 uL-C union-single-eq-member*)
using *C-W LM-C W uL-M1* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
have ‹$-L \in\#$ *watched C*›
proof (*rule ccontr*)
assume *uL-W*: ‹$-L \notin\#$ *watched C*›
have ‹$K \neq -L \vee K' \neq -L$›
using *dist-C C-W W* **by** *auto*
moreover have ‹$K \notin$ *lits-of-l M1*› **and** ‹$K' \notin$ *lits-of-l M1*› **and** *L-M*: ‹$L \notin$ *lits-of-l M1*›
proof −
have *f2*: ‹*consistent-interp* (*lits-of-l M1*)›
using *distinct-consistent-interp n-d-L-M1* **by** *auto*
have *undef-L*: ‹*undefined-lit M1 L*›
using *atm-lit-of-set-lits-of-l n-d-L-M1* **by** *force*
then show ‹$K \notin$ *lits-of-l M1*›
using *f2 neg-C* **unfolding** *C-W W* **by** (*metis* (*no-types*) *C-W W add-diff-cancel-right′ atm-of-eq-atm-of clause.simps consistent-interp-def in-diffD insertE list.simps*(*15*) *lits-of-insert uL-C union-single-eq-member Decided-Propagated-in-iff-in-lits-of-l*)
show ‹$K' \notin$ *lits-of-l M1*›
using *consistent-interp-def distinct-consistent-interp n-d-L-M1*
using *neg-C uL-W n-d* **unfolding** *C-W W* **by** *auto*
show ‹$L \notin$ *lits-of-l M1*›
using *undef-L* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
ultimately have ‹($-K \in$ *lits-of-l M1* ∧ $K' \notin$ *lits-of-l M1*) ∨
($-K' \in$ *lits-of-l M1* ∧ $K \notin$ *lits-of-l M1*)›
using *neg-C* **by** (*auto simp*: *C-W W*)
moreover have ‹¬*has-blit M1* (*clause C*) *K*›
using ‹$K \notin$ *lits-of-l M1*› ‹$K' \notin$ *lits-of-l M1*› ‹$L \notin$ *lits-of-l M1*› *uL-M1 n-d-L-M1 no-dup-cons*
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto dest*!: *multi-member-split dest*!: *no-dup-consistentD*[*OF n-d-L-M1*]
dest!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
using *n-d-L-M1 no-dup-cons no-dup-consistentD* **by** *blast*
moreover have ‹¬*has-blit M1* (*clause C*) *K′*›
using ‹$K' \notin$ *lits-of-l M1*› ‹ $K \notin$ *lits-of-l M1*› ‹$L \notin$ *lits-of-l M1*› *uL-M1 n-d-L-M1 no-dup-cons no-dup-consistentD*
using *uL-C neg-C n-d* **unfolding** *has-blit-def* **apply** (*auto 10 10 dest*!: *multi-member-split*

dest!: *in-lits-of-l-defined-litD*[*of* ‹−*L*›] *simp*: *add-mset-eq-add-mset*)
using *n-d-L-M1 no-dup-cons no-dup-consistentD* **by** *auto*
moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *excep-M1 C* **by** *auto*
ultimately have ‹∀ *K* ∈# *unwatched C*. −*K* ∈ *lits-of-l M1*›
using *C twl-clause.sel*(*1*) *union-single-eq-member w-q*
by (*fastforce simp*: *twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib L-M*)
then show *False*
using *uL-W uL-C L-M K-L uL-M1* **unfolding** *C-W W* **by** *auto*
qed
then show ‹(∃ *L′*. *L′* ∈# *watched C* ∧ *L′* ∈# {#− *L*#}) ∨ (∃ *L*. (*L*, *C*) ∈# {#})›
by *auto*
qed
qed

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

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

have ‹−*L* ∈# *watched C*›
proof (*rule ccontr*)
assume *uL-W*: ‹−*L* ∉# *watched C*›
then obtain *W UW K K′* **where**
C-W: ‹*C* = *TWL-Clause W UW*› **and**
W: ‹*W* = {#*K*, *K′*#}› **and**
uK-M: ‹−*K* ∈ *lits-of-l M1*›
using *struct-C neg-C* **by** (*cases C*) (*auto simp*: *size-2-iff remove1-mset-add-mset-If add-mset-commute split*: *if-splits*)
have *FK-F*: ‹*FK* ≠ *K*›
using *uK-M undef-M-K* **unfolding** *Decided-Propagated-in-iff-in-lits-of-l* **by** *auto*

have ‹$K \neq -L$›
using *uK-M uL-M1* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
moreover have ‹$K \notin$ *lits-of-l M1*›
using *neg-C uL-W n-d uK-M n-d-L-M1* **by** (*auto simp*: *lits-of-def uminus-lit-swap no-dup-cannot-not-lit-and-uminus dest*: *no-dup-cannot-not-lit-and-uminus*)
ultimately have ‹$K' \notin$ *lits-of-l M1*›
apply (*cases* ‹$K' = FK$›)
using *undef-M-K* **apply** (*force simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
using *neg-C C-W W FK-F n-d uL-W n-d-L-M1* **by** (*auto simp add*: *remove1-mset-add-mset-If uminus-lit-swap lits-of-def no-dup-cannot-not-lit-and-uminus dest*: *no-dup-cannot-not-lit-and-uminus*)
moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *excep-M1 C* **by** *auto*
moreover have ‹¬*has-blit M1* (*clause C*) *K*›
using ‹$K \notin$ *lits-of-l M1*› ‹$K' \notin$ *lits-of-l M1*› *uL-M1 n-d-L-M1 no-dup-cons*
using *n-d-L-M1 no-dup-cons no-dup-consistentD*
using *K in-lits-of-l-defined-litD undef*
using *neg-C n-d* **unfolding** *has-blit-def* **by** (*fastforce dest*!: *multi-member-split dest*!: *no-dup-consistentD*[*OF n-d-L-M1*] *dest*!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹¬*has-blit M1* (*clause C*) K'›
using ‹$K' \notin$ *lits-of-l M1*› ‹ $K \notin$ *lits-of-l M1*› *uL-M1 n-d-L-M1 no-dup-cons no-dup-consistentD*
using *n-d-L-M1 no-dup-cons no-dup-consistentD*
using *K in-lits-of-l-defined-litD undef*
using *neg-C n-d* **unfolding** *has-blit-def* **by** (*fastforce dest*!: *multi-member-split dest*!: *in-lits-of-l-defined-litD*[*of* ‹$-L$›] *simp*: *add-mset-eq-add-mset*)
moreover have ‹*twl-exception-inv* (*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*›
using *excep-M1 C* **by** *auto*
ultimately have ‹$\forall K \in\#$ *unwatched C*. $-K \in$ *lits-of-l M1*›
using *uK-M*
by (*auto simp*: *twl-exception-inv.simps C-W W add-mset-eq-add-mset all-conj-distrib*)
then show *False*
using *C-W uL-M1* ‹$- L \in\#$ *clause C*› *uL-W*
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
qed
then show ‹($\exists L'$. $L' \in\#$ *watched C* $\wedge$ $L' \in\#$ {#$- L$#}) $\vee$ ($\exists L$. (L, C) $\in\#$ {#})›
by *auto*
next
case *True*
then have ‹$\forall K\in\#$*remove1-mset L D'*. $- K \in$ *lits-of-l* (*Propagated L D'* # *M1*)›
using *M1-CNot-L-D* **by** (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model*)
then have ‹$\forall K\in\#$*remove1-mset L D'*. *defined-lit* (*Propagated L D'* # *M1*) *K*›
using *Decided-Propagated-in-iff-in-lits-of-l* **by** *blast*
moreover have ‹*defined-lit* (*Propagated L D'* # *M1*) *L*›
by (*auto simp*: *defined-lit-map*)
ultimately have ‹$\forall K\in\#D'$. *defined-lit* (*Propagated L D'* # *M1*) *K*›
by (*metis in-remove1-mset-neq*)
then have ‹$\forall K\in\#$*clause ?D*. *defined-lit* (*Propagated L D'* # *M1*) *K*›
using *LD'* ‹*defined-lit* (*Propagated L D'* # *M1*) *L*› **by** (*auto dest*: *in-diffD*)
then have *False*
using *K undef* **unfolding** *True* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*)
then show *?thesis* **by** *fast*
qed
qed

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

show *?case*
using *2*[*of K*] *3 n-d-L-M1 L-M1 blit-L′*
apply (*simp only*: *filter-mset-empty-conv Ball-def image-mset-is-empty-iff*)
by (*fastforce simp add*: *clauses-to-update-prop.simps*)
next
case (*Q K′ C*)
then show *?case*
using *3*[*of C K′*] *uL-M1 blit-L′ n-d-L-M1 has-blit-Cons*
by (*fastforce simp add*: *clauses-to-update-prop.simps*
add-mset-eq-add-mset Decided-Propagated-in-iff-in-lits-of-l)
qed
qed

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

lemma *cdcl-twl-o-entailed-clss-inv*:
assumes
cdcl: ⟨*cdcl-twl-o S T*⟩ **and**
unit: ⟨*twl-struct-invs S*⟩
shows ⟨*entailed-clss-inv T*⟩
using *cdcl unit*

proof (*induction rule*: *cdcl-twl-o.induct*)
  case (*decide M L N NE U UE*) **note** *undef* = *this*(*1*) **and** *twl* = *this*(*3*)
  then have *unit*: ‹*entailed-clss-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})›
    unfolding *twl-struct-invs-def* **by** *fast*
  show *?case*
    unfolding *entailed-clss-inv.simps Ball-def*
  proof (*intro allI impI*)
    fix *C*
    assume ‹*C* ∈# *NE* + *UE*›
    then obtain *K* **where** ‹*K* ∈# *C*› **and** *K*: ‹*K* ∈ *lits-of-l M*› **and** ‹*get-level M K* = *0*›
      using *unit* **by** *auto*
    moreover have ‹*atm-of L* ≠ *atm-of K*›
      using *undef K* **by** (*auto simp*: *defined-lit-map lits-of-def*)
    ultimately show ‹∃ *La*. *La* ∈# *C* ∧ (*None* = *None* ∨ *0* < *count-decided* (*Decided L* # *M*) ⟶
      get-level (*Decided L* # *M*) *La* = *0* ∧ *La* ∈ *lits-of-l* (*Decided L* # *M*))›
      by *auto*
  qed
next
  case (*skip L D C′ M N U NE UE*) **note** *twl* = *this*(*3*)
  let *?M* = ‹*Propagated L C′* # *M*›
  have *unit*: ‹*entailed-clss-inv* (*?M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})›
    using *twl* **unfolding** *twl-struct-invs-def* **by** *fast*
  show *?case*
    unfolding *entailed-clss-inv.simps Ball-def*
  proof (*intro allI impI*, *cases* ‹*count-decided M* = *0*›)
    case *True* **note** [*simp*] = *this*
    fix *C*
    assume ‹*C* ∈# *NE* + *UE*›
    then obtain *K* **where** ‹*K* ∈# *C*›
      using *unit* **by** *auto*
    then show ‹∃ *L*. *L* ∈# *C* ∧ (*Some D* = *None* ∨ *0* < *count-decided M* ⟶
        get-level M L = *0* ∧ *L* ∈ *lits-of-l M*)›
      by *auto*
  next
    case *False*
    fix *C*
    assume ‹*C* ∈# *NE* + *UE*›
    then obtain *K* **where** ‹*K* ∈# *C*› **and** *K*: ‹*K* ∈ *lits-of-l ?M*› **and** *lev-K*: ‹*get-level ?M K* = *0*›
      using *unit False* **by** *auto*
    moreover {
      have ‹*get-level ?M L* > *0*›
        using *False* **by** *auto*
      then have ‹*atm-of L* ≠ *atm-of K*›
        using *lev-K* **by** *fastforce* **}**
    ultimately show ‹∃ *L*. *L* ∈# *C* ∧ (*Some D* = *None* ∨ *0* < *count-decided M* ⟶
        get-level M L = *0* ∧ *L* ∈ *lits-of-l M*)›
      using *False* **by** *auto*
  qed
next
  case (*resolve L D C M N U NE UE*) **note** *twl* = *this*(*3*)
  let *?M* = ‹*Propagated L C* # *M*›
  let *?D* = ‹*Some* (*remove1-mset* (− *L*) *D* ∪# *remove1-mset L C*)›
  have *unit*: ‹*entailed-clss-inv* (*?M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})›
    using *twl* **unfolding** *twl-struct-invs-def* **by** *fast*
  show *?case*
    unfolding *entailed-clss-inv.simps Ball-def*

proof (*intro allI impI*, *cases* ‹*count-decided M* = *0*›)
  case *True* **note** [*simp*] = *this*
  fix *E*
  assume ‹*E* ∈# *NE* + *UE*›
  then obtain *K* **where** ‹*K* ∈# *E*›
    using *unit* **by** *auto*
  then show ‹∃ *La*. *La* ∈# *E* ∧ (*?D* = *None* ∨ *0* < *count-decided M* ⟶
      get-level M La = *0* ∧ *La* ∈ *lits-of-l M*)›
    by *auto*
next
  case *False*
  fix *E*
  assume ‹*E* ∈# *NE* + *UE*›
  then obtain *K* **where** ‹*K* ∈# *E*› **and** *K*: ‹*K* ∈ *lits-of-l ?M*› **and** *lev-K*: ‹*get-level ?M K* = *0*›
    using *unit False* **by** *auto*
  moreover {
    have ‹*get-level ?M L* > *0*›
      using *False* **by** *auto*
    then have ‹*atm-of L* ≠ *atm-of K*›
      using *lev-K* **by** *fastforce* **}**
  ultimately show ‹∃ *La*. *La* ∈# *E* ∧ (*?D* = *None* ∨ *0* < *count-decided M* ⟶
      get-level M La = *0* ∧ *La* ∈ *lits-of-l M*)›
    using *False* **by** *auto*
qed
next
  case (*backtrack-unit-clause L D K M1 M2 M D′ i N U NE UE*) **note** *decomp* = *this*(*2*) **and**
    lev-L = *this*(*3*) **and** *i* = *this*(*5*) **and** *lev-K* = *this*(*6*) **and** *D′*[*simp*] = *this*(*7*) **and** *twl* = *this*(*10*)
  let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})›
  let *?T* = ‹(*Propagated L* {#*L*#} # *M1*, *N*, *U*, *None*, *NE*, *add-mset* {#*L*#} *UE*, {#}, {#− *L*#})›
  let *?M* = ‹*Propagated L* {#*L*#} # *M1*›
  have *unit*: ‹*entailed-clss-inv ?S*›
    using *twl* **unfolding** *twl-struct-invs-def* **by** *fast*
  obtain *M3* **where** *M*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
    using *decomp* **by** *auto*
  define *M2′* **where** ‹*M2′* = (*M3* @ *M2*) @ *Decided K* # []›
  have *M2′*: ‹*M* = *M2′* @ *M1*›
    unfolding *M M2′-def* **by** *simp*
  have *count-dec-M2′*: ‹*count-decided M2′* ≠ *0*›
    unfolding *M2′-def* **by** *auto*
  have *lev-M*: ‹*count-decided M* > *0*›
    unfolding *M* **by** *auto*
  have *n-d*: ‹*no-dup M*›
    using *twl* **unfolding** *cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def twl-struct-invs-def*
      cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def **by** (*auto simp*: *trail.simps*)
  have *count-dec-M1*: ‹*count-decided M1* = *0*›
    using *no-dup-append-decided-Cons-lev*[*of* ‹*M3* @ *M2*› *K M1*]
      lev-K n-d i **unfolding** *M* **by** *simp*

  show *?case*
    unfolding *entailed-clss-inv.simps Ball-def*
  proof (*intro allI impI*)
    fix *C*
    assume *C*: ‹*C* ∈# *NE* + *add-mset* {#*L*#} *UE*›
    show ‹∃ *La*. *La* ∈# *C*∧ (*None* = *None* ∨ *0* < *count-decided ?M* ⟶ *get-level ?M La* = *0* ∧
        La ∈ *lits-of-l ?M*)›
    proof (*cases* ‹*C* ∈# *NE* + *UE*›)

case *True*
then obtain K'' **where** *C-K*: ‹$K'' \in\#$ *C*› **and** *K*: ‹$K'' \in$ *lits-of-l M*› **and**
lev-K'': ‹*get-level M K''* = *0*›
using *unit lev-M* **by** *auto*
have ‹$K'' \in$ *lits-of-l M1*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹$K'' \in$ *lits-of-l M2'*›
using *K* **unfolding** *M2'* **by** *auto*
then have *ex-L*: ‹∃ *L*∈*set* ((*M3* @ *M2*) @ [*Decided K*]). ¬ *atm-of* (*lit-of L*) ≠ *atm-of K''*›
by (*metis M2'-def image-iff lits-of-def*)
have ‹*get-level* (*M2'* @ *M1*) *K''* = *get-level M2' K''* + *count-decided M1*›
using ‹$K'' \in$ *lits-of-l M2'*› *Decided-Propagated-in-iff-in-lits-of-l get-level-skip-end*
by *blast*

with *last-in-set-dropWhile*[*OF ex-L*, *unfolded M2'-def*[*symmetric*]]
have ‹¬*get-level M K''* = *0*›
unfolding *M2'* **using** ‹$K'' \in$ *lits-of-l M2'*› **by** (*force simp*: *filter-empty-conv get-level-def*)
then show *False*
using *lev-K''* **by** *arith*
qed
then have *K*: ‹$K'' \in$ *lits-of-l ?M*›
unfolding *M* **by** *auto*
moreover {
have ‹*atm-of L* ≠ *atm-of K''*›
using *lev-L lev-K'' lev-M* **by** (*auto simp*: *atm-of-eq-atm-of*)
then have ‹*get-level ?M K''* = *0*›
using *count-dec-M1 count-decided-ge-get-level*[*of ?M K''*] **by** *auto* **}**
ultimately show *?thesis*
using *C-K* **by** *auto*
next
case *False*
then have ‹*C* = {#*L*#}›
using *C* **by** *auto*
then show *?thesis*
using *count-dec-M1* **by** *auto*
qed
qed
next
case (*backtrack-nonunit-clause L D K M1 M2 M D' i N U NE UE L'*) **note** *decomp* = *this*(*2*) **and**
lev-L-M = *this*(*3*) **and** *lev-K* = *this*(*6*) **and** *twl* = *this*(*13*)
let *?S* = ‹(*M*, *N*, *U*, *Some D*, *NE*, *UE*, {#}, {#})›
let *?T* = ‹(*Propagated L D'* # *M1*, *N*, *add-mset* (*TWL-Clause* {#*L*, *L'*#} (*D'* − {#*L*, *L'*#})) *U*,
None,
NE, *UE*, {#}, {#−*L*#})›
let *?M* = ‹*Propagated L D'* # *M1*›
have *unit*: ‹*entailed-clss-inv ?S*›
using *twl* **unfolding** *twl-struct-invs-def* **by** *fast*
obtain *M3* **where** *M*: ‹*M* = *M3* @ *M2* @ *Decided K* # *M1*›
using *decomp* **by** *auto*
define *M2'* **where** ‹*M2'* = (*M3* @ *M2*) @ *Decided K* # []›
have *M2'*: ‹*M* = *M2'* @ *M1*›
unfolding *M M2'-def* **by** *simp*
have *count-dec-M2'*: ‹*count-decided M2'* ≠ *0*›
unfolding *M2'-def* **by** *auto*
have *lev-M*: ‹*count-decided M* > *0*›

unfolding *M* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *twl* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def twl-struct-invs-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
have *count-dec-M1*: ‹*count-decided M1 = i*›
using *no-dup-append-decided-Cons-lev*[*of* ‹*M3 @ M2*› *K M1*]
lev-K n-d **unfolding** *M* **by** *simp*

show *?case*
unfolding *entailed-clss-inv.simps Ball-def*
proof (*intro allI impI*)
fix *C*
assume *C*: ‹*C* ∈# *NE* + *UE*›
then obtain *K''* **where** *C-K*: ‹*K''* ∈# *C*› **and** *K*: ‹*K''* ∈ *lits-of-l M*› **and**
lev-K'': ‹*get-level M K''* = *0*›
using *unit lev-M* **by** *auto*
have *K''-M1*: ‹*K''* ∈ *lits-of-l M1*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*K''* ∈ *lits-of-l M2'*›
using *K* **unfolding** *M2'* **by** *auto*
then have ‹∃ *L*∈*set* ((*M3 @ M2*) @ [*Decided K*]). ¬ *atm-of* (*lit-of L*) ≠ *atm-of K''*›
by (*metis M2'-def image-iff lits-of-def*)
then have *ex-L*: ‹∃ *L*∈*set* ((*M3 @ M2*) @ [*Decided K*]). ¬ *atm-of* (*lit-of L*) ≠ *atm-of K''*›
by (*metis M2'-def image-iff lits-of-def*)
have ‹*get-level* (*M2' @ M1*) *K''* = *get-level M2' K''* + *count-decided M1*›
using ‹*K''* ∈ *lits-of-l M2'*› *Decided-Propagated-in-iff-in-lits-of-l get-level-skip-end*
by *blast*

with *last-in-set-dropWhile*[*OF ex-L*, *unfolded M2'-def*[*symmetric*]] **have** ‹¬*get-level M K''* = *0*›
unfolding *M2'* **using** ‹*K''* ∈ *lits-of-l M2'*› **by** (*force simp*: *filter-empty-conv get-level-def*)
then show *False*
using *lev-K''* **by** *arith*
qed
then have *K*: ‹*K''* ∈ *lits-of-l ?M*›
unfolding *M* **by** *auto*
moreover **{**
have ‹*undefined-lit* (*M3 @ M2 @* [*Decided K*]) *K''*›
by (*rule no-dup-append-in-atm-notin*[*of -* ‹*M1*›])
(*use n-d M K''-M1* **in** *auto*)
then have ‹*get-level M1 K''* = *0*›
using *lev-K''* **unfolding** *M* **by** (*auto simp*: *image-Un*)
moreover have ‹*atm-of L* ≠ *atm-of K''*›
using *lev-K'' lev-M lev-L-M* **by** (*metis atm-of-eq-atm-of get-level-uminus not-gr-zero*)
ultimately have ‹*get-level ?M K''* = *0*›
by *auto* **}**
ultimately show ‹∃ *La*. *La* ∈# *C* ∧ (*None* = *None* ∨ *0* < *count-decided ?M* ⟶
get-level ?M La = *0* ∧ *La* ∈ *lits-of-l ?M*)›
using *C-K* **by** *auto*
qed
qed

The Strategy

lemma *no-literals-to-update-no-cp*:
assumes

WS: ‹*clauses-to-update S* = {#}› **and** *Q*: ‹*literals-to-update S* = {#}› **and**
twl: ‹*twl-struct-invs S*›
shows
‹*no-step* $cdcl_W$*-restart-mset.propagate* ($state_W$*-of S*)› **and**
‹*no-step* $cdcl_W$*-restart-mset.conflict* ($state_W$*-of S*)›
proof −
obtain *M N U NE UE D* **where**
S: ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, {#})›
using *WS Q* **by** (*cases S*) *auto*

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

have *twl-st-inv*: ‹*twl-st-inv S*› **and**
struct-inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
excep: ‹*twl-st-exception-inv S*› **and**
confl-cands: ‹*confl-cands-enqueued S*› **and**
propa-cands: ‹*propa-cands-enqueued S*› **and**
unit: ‹*entailed-clss-inv S*›
using *twl* **unfolding** *twl-struct-invs-def* **by** *fast+*
have *n-d*: ‹*no-dup M*›
using *struct-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps S*)
then have *L-uL*: ‹$L \in$ *lits-of-l M* $\Longrightarrow -L \notin$ *lits-of-l M*› **for** *L*
using *consistent-interp-def distinct-consistent-interp* **by** *blast*
have ‹$\forall$ *C* $\in$# *N* + *U*. ¬*M*⊨*as CNot* (*clause C*)›
using *confl-cands* **unfolding** *S* **by** *auto*
moreover have ‹¬*M*⊨*as CNot C*› **if** *C*: ‹*C* $\in$# *NE* + *UE*› **for** *C*
proof −
obtain *L* **where** *L*: ‹*L* $\in$# *C*› **and** ‹*L* $\in$ *lits-of-l M*›
using *unit C* **unfolding** *S* **by** *auto*
then have ‹*M* ⊨*a C*›
by (*auto simp*: *true-annot-def dest*!: *multi-member-split*)
then show *?thesis*
using *L* ‹*L* $\in$ *lits-of-l M*› **by** (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model*
dest: *L-uL multi-member-split*)
qed
ultimately have *ns-confl*: ‹*no-step* $cdcl_W$*-restart-mset.conflict* ($state_W$*-of S*)›
by (*auto elim*!: $cdcl_W$*-restart-mset.conflictE simp*: *S trail.simps clauses-def*)

have *ns-propa*: ‹*no-step* $cdcl_W$*-restart-mset.propagate* ($state_W$*-of S*)›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then obtain *C L* **where**
C: ‹*C* $\in$# *clause* ‘# (*N* + *U*) + *NE* + *UE*› **and**
L: ‹*L* $\in$# *C*› **and**
M: ‹*M* ⊨*as CNot* (*remove1-mset L C*)› **and**
undef: ‹*undefined-lit M L*›
by (*auto elim*!: $cdcl_W$*-restart-mset.propagateE simp*: *S trail.simps clauses-def*) *blast+*
show *False*
proof (*cases* ‹*C* $\in$# *clause* ‘# (*N* + *U*)›)
case *True*
then show *?thesis*
using *propa-cands L M undef* **by** (*auto simp*: *S*)

next
  case *False*
  then have ‹$C \in\#$ *NE* + *UE*›
    using *C* **by** *auto*
  then obtain L'' **where** L'': ‹$L'' \in\# C$› **and** L''*-def*: ‹$L'' \in$ *lits-of-l M*›
    using *unit* **unfolding** *S* **by** *auto*
  then show *?thesis*
    using *undef* L'' L''*-def L M L-uL*
    by (*auto simp*: *S true-annots-true-cls-def-iff-negation-in-model*
      add-mset-eq-add-mset
      Decided-Propagated-in-iff-in-lits-of-l dest!: *multi-member-split*)
qed
qed
note *ns-confl ns-propa*
}
moreover {
  assume ‹*get-conflict S* $\neq$ *None*›
  then have ‹*no-step* $cdcl_W$*-restart-mset.propagate* ($state_W$*-of S*)›
    ‹*no-step* $cdcl_W$*-restart-mset.conflict* ($state_W$*-of S*)›
    by (*auto elim*!: $cdcl_W$*-restart-mset.propagateE* $cdcl_W$*-restart-mset.conflictE*
      simp: *S conflicting.simps*)
}
ultimately show ‹*no-step* $cdcl_W$*-restart-mset.propagate* ($state_W$*-of S*)›
    ‹*no-step* $cdcl_W$*-restart-mset.conflict* ($state_W$*-of S*)›
  by *blast+*
qed

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

lemma *cdcl-twl-stgy-*$cdcl_W$*-stgy2*:
  assumes ‹*cdcl-twl-stgy S T*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* ($state_W$*-of S*) ($state_W$*-of T*) $\vee$
    ($state_W$*-of S* = $state_W$*-of T* $\wedge$ (*literals-to-update-measure T*, *literals-to-update-measure S*)
    $\in$ *lexn less-than 2*)›
  using *assms*(*1*)
proof (*induction rule*: *cdcl-twl-stgy.induct*)
  case (*cp* S')
  then show *?case*
    using *twl* **by** (*auto dest*!: *cdcl-twl-cp-*$cdcl_W$*-stgy*)
next
  case (*other'* S') **note** *o* = *this*(*1*)
  have *wq*: ‹*clauses-to-update S* = {#}› **and** *p*: ‹*literals-to-update S* = {#}›
    using *o* **by** (*cases rule*: *cdcl-twl-o.cases*; *auto*)+
  show *?case*
    apply (*rule disjI1*)
    apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.other'*)
    using *no-literals-to-update-no-cp*[*OF wq p twl*] **apply** (*simp*; *fail*)
    using *no-literals-to-update-no-cp*[*OF wq p twl*] **apply** (*simp*; *fail*)
    using *cdcl-twl-o-*$cdcl_W$*-o*[*of S* S', *OF o*] *twl* **apply** (*simp add*: *twl-struct-invs-def*; *fail*)
    done
qed

lemma *cdcl-twl-stgy-*$cdcl_W$*-stgy*:
  assumes ‹*cdcl-twl-stgy S T*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy*** ($state_W$*-of S*) ($state_W$*-of T*)›
  using *cdcl-twl-stgy-*$cdcl_W$*-stgy2*[*OF assms*] **by** *auto*

lemma *cdcl-twl-o-twl-struct-invs*:
  assumes
    cdcl: ‹*cdcl-twl-o S T*› **and**
    twl: ‹*twl-struct-invs S*›
  shows ‹*twl-struct-invs T*›
proof −
  have $cdcl_W$: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-restart* ($state_W$*-of S*) ($state_W$*-of T*)›
    using *twl* **unfolding** *twl-struct-invs-def*
    by (*meson cdcl* $cdcl_W$*-restart-mset.other cdcl-twl-o-*$cdcl_W$*-o*)

  have *wq*: ‹*clauses-to-update S* = {#}› **and** *p*: ‹*literals-to-update S* = {#}›
    using *cdcl* **by** (*cases rule*: *cdcl-twl-o.cases*; *auto*)+
  have $cdcl_W$*-stgy*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* ($state_W$*-of S*) ($state_W$*-of T*)›
    apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.other′*)
    using *no-literals-to-update-no-cp*[*OF wq p twl*] **apply** (*simp*; *fail*)
    using *no-literals-to-update-no-cp*[*OF wq p twl*] **apply** (*simp*; *fail*)
    using *cdcl-twl-o-*$cdcl_W$*-o*[*of S T*, *OF cdcl*] *twl* **apply** (*simp add*: *twl-struct-invs-def*; *fail*)
    done
  have *init*: ‹*init-clss* ($state_W$*-of T*) = *init-clss* ($state_W$*-of S*)›
    using $cdcl_W$ **by** (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-restart-init-clss*)
  show *?thesis*
    unfolding *twl-struct-invs-def*
    apply (*intro conjI*)
    subgoal by (*use cdcl cdcl-twl-o-twl-st-inv twl* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-valid* **in** ‹*blast*; *fail*›)
    subgoal by (*use* $cdcl_W$ $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-inv twl twl-struct-invs-def* **in** ‹*blast*; *fail*›)
    subgoal by (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-no-smaller-propa*[*OF* $cdcl_W$*-stgy*])
      ((*use twl* **in** ‹*simp add*: *init twl-struct-invs-def*; *fail*›)+)[*2*]
    subgoal by (*use cdcl cdcl-twl-o-twl-st-exception-inv twl* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-no-duplicate-queued* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-distinct-queued* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-confl-cands-enqueued twl twl-struct-invs-def* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-propa-cands-enqueued twl twl-struct-invs-def* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl twl cdcl-twl-o-conflict-None-queue* **in** ‹*blast*; *fail*›)
    subgoal by (*use cdcl cdcl-twl-o-entailed-clss-inv twl twl-struct-invs-def* **in** *blast*)
    subgoal by (*use cdcl twl-o-clauses-to-update twl* **in** *blast*)
    subgoal by (*use cdcl cdcl-twl-o-past-invs twl twl-struct-invs-def* **in** *blast*)
    done
qed

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

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

lemma *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*:
  assumes ‹*cdcl-twl-stgy*** *S T*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** (*state*$_W$*-of S*) (*state*$_W$*-of T*)›
  using *assms* **by** (*induction rule*: *rtranclp-induct*)
    (*auto dest*!: *cdcl-twl-stgy-cdcl*$_W$*-stgy intro*: *rtranclp-cdcl-twl-stgy-twl-struct-invs*)

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

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

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

show *False*
using *cdcl-twl-cp.update-clause*[*of C L L' M K N U* ‹*fst NU*› ‹*snd NU*› *NE UE WS' Q*]
watched uL-M L'-M K undef-K-K-M upd ns-cp **unfolding** *S WS'* **by** *simp*
qed
then have *p*: ‹*literals-to-update S* = {#}›
using *cdcl-twl-cp.pop*[*of M N U NE UE*] *S ns-cp* **by** (*cases* ‹*Q*›) *fastforce+*
show *?thesis* **using** *wq p* **by** *blast*
qed

lemma *no-step-cdcl-twl-o-no-step-cdcl*$_W$*-o*:
assumes
ns-o: ‹*no-step cdcl-twl-o S*› **and**
twl: ‹*twl-struct-invs S*› **and**
p: ‹*literals-to-update S* = {#}› **and**
w-q: ‹*clauses-to-update S* = {#}›
shows ‹*no-step cdcl*$_W$*-restart-mset.cdcl*$_W$*-o* (*state*$_W$*-of S*)›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then obtain *T* **where** *T*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-o* (*state*$_W$*-of S*) *T*›
by *blast*
obtain *M N U D NE UE* **where** *S*: ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, {#})›
using *p w-q* **by** (*cases S*) *auto*
have *unit*: ‹*entailed-clss-inv S*›
using *twl* **unfolding** *twl-struct-invs-def* **by** *fast+*
show *False*
using *T*
proof (*cases rule*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-o-induct*)
case (*decide L T*) **note** *confl* = *this*(*1*) **and** *undef* = *this*(*2*) **and** *atm* = *this*(*3*) **and** *T* = *this*(*4*)
show *?thesis*
using *cdcl-twl-o.decide*[*of M L N NE U UE*] *confl undef atm ns-o* **unfolding** *S*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state*)
next
case (*skip L C' M' E T*) **note** *M* = *this* **and** *confl* = *this*(*2*) **and** *uL-E* = *this*(*3*) **and** *E* = *this*(*4*)
and
T = *this*(*5*)
show *?thesis*
using *cdcl-twl-o.skip*[*of L E C' M' N U NE UE*] *M uL-E E ns-o* **unfolding** *S*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state*)
next
case (*resolve L E M' D T*) **note** *M* = *this*(*1*) **and** *L-E* = *this*(*2*) **and** *hd* = *this*(*3*) **and**
confl = *this*(*4*) **and** *uL-D* = *this*(*5*) **and** *max-lvl* = *this*(*6*)
show *?thesis*
using *cdcl-twl-o.resolve*[*of L D E M' N U NE UE*] *M L-E ns-o max-lvl uL-D confl* **unfolding** *S*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state*)
next
case (*backtrack L C K i M1 M2 T D'*) **note** *confl* = *this*(*1*) **and** *decomp* = *this*(*2*) **and**
lev-L-bt = *this*(*3*) **and** *lev-L* = *this*(*4*) **and** *i* = *this*(*5*) **and** *lev-K* = *this*(*6*) **and** *D'-C* = *this*(*7*)
show *?thesis*
proof (*cases* ‹*D'* = {#}›)
case *True*
show *?thesis*
using *cdcl-twl-o.backtrack-unit-clause*[*of L* ‹*add-mset L C*› *K M1 M2 M*
‹*add-mset L D'*› *i N U NE UE*]
decomp True lev-L-bt lev-L i lev-K ns-o confl backtrack **unfolding** *S*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state clauses-def inf-sup-aci*(*6*) *sup.left-commute*)
next

case *False*
then obtain L' **where**
L'-C: ‹$L' \in\# D'$› **and** *lev-*L': ‹*get-level* M $L' = i$›
using *i get-maximum-level-exists-lit-of-max-level*[*of* D' *M*] *confl S*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state S dest*: *in-diffD*)

show *?thesis*
using *cdcl-twl-o.backtrack-nonunit-clause*[*of L* ‹*add-mset L C*› *K M1 M2 M* ‹*add-mset L* D'›
i N U NE UE L']
using *decomp lev-L-bt lev-L i lev-K False* L'*-C lev-*L' *ns-o confl backtrack*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state S inf-sup-aci*(*6*) *sup.left-commute clauses-def*
dest: *in-diffD*)
qed
qed
qed

lemma *no-step-cdcl-twl-stgy-no-step-cdcl*$_W$*-stgy*:
assumes *ns*: ‹*no-step cdcl-twl-stgy S*› **and** *twl*: ‹*twl-struct-invs S*›
shows ‹*no-step cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy* (*state*$_W$*-of S*)›
proof −
have *ns-cp*: ‹*no-step cdcl-twl-cp S*› **and** *ns-o*: ‹*no-step cdcl-twl-o S*›
using *ns* **by** (*auto simp*: *cdcl-twl-stgy.simps*)
then have *w-q*: ‹*clauses-to-update S* = {#}› **and** *p*: ‹*literals-to-update S* = {#}›
using *ns-cp no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp twl* **by** *blast+*
then have
‹*no-step cdcl*$_W$*-restart-mset.propagate* (*state*$_W$*-of S*)› **and**
‹*no-step cdcl*$_W$*-restart-mset.conflict* (*state*$_W$*-of S*)›
using *no-literals-to-update-no-cp twl* **by** *blast+*
moreover have ‹*no-step cdcl*$_W$*-restart-mset.cdcl*$_W$*-o* (*state*$_W$*-of S*)›
using *w-q p ns-o no-step-cdcl-twl-o-no-step-cdcl*$_W$*-o twl* **by** *blast*
ultimately show *?thesis*
by (*auto simp*: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy.simps*)
qed

lemma *full-cdcl-twl-stgy-cdcl*$_W$*-stgy*:
assumes ‹*full cdcl-twl-stgy S T*› **and** *twl*: ‹*twl-struct-invs S*›
shows ‹*full cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy* (*state*$_W$*-of S*) (*state*$_W$*-of T*)›
by (*metis* (*no-types*, *hide-lams*) *assms*(*1*) *full-def no-step-cdcl-twl-stgy-no-step-cdcl*$_W$*-stgy*
rtranclp-cdcl-twl-stgy-cdcl$_W$*-stgy rtranclp-cdcl-twl-stgy-twl-struct-invs twl*)

definition *init-state-twl* **where**
‹*init-state-twl N* ≡ ([], *N*, {#}, *None*, {#}, {#}, {#}, {#})›
lemma
assumes
struct: ‹∀ *C* ∈# *N*. *struct-wf-twl-cls C*› **and**
tauto: ‹∀ *C* ∈# *N*. ¬*tautology* (*clause C*)›
shows
twl-stgy-invs-init-state-twl: ‹*twl-stgy-invs* (*init-state-twl N*)› **and**
twl-struct-invs-init-state-twl: ‹*twl-struct-invs* (*init-state-twl N*)›
proof −
have [*simp*]: ‹*twl-lazy-update* [] *C*› ‹*watched-literals-false-of-max-level* [] *C*›
‹*twl-exception-inv* ([], *N*, {#}, *None*, {#}, {#}, {#}, {#}) *C*› **for** *C*
by (*cases C*; *solves* ‹*auto simp*: *twl-exception-inv.simps*›)+

have *size-C*: ‹*size* (*clause C*) ≥ *2*› **if** ‹*C* ∈# *N*› **for** *C*

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

have ‹*distinct-mset* (*clause C*)› **if** ‹*C* ∈# *N*› **for** *C*
  using *struct that* **by** (*cases C*) (*auto*)
then have *dist*: ‹*distinct-mset-mset* (*clause* ‘# *N*)›
  by (*auto simp*: *distinct-mset-set-def*)
then have [*simp*]: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ([], *clause* ‘# *N*, {#}, *None*)›
  using *struct* **unfolding** *init-state.simps*[*symmetric*]
  by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*)
have [*simp*]: ‹$cdcl_W$*-restart-mset.no-smaller-propa* ([], *clause* ‘# *N*, {#}, *None*)›
  by(*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def* $cdcl_W$*-restart-mset-state*)

show *stgy-invs*: ‹*twl-stgy-invs* (*init-state-twl N*)›
  by (*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
    $cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
    $cdcl_W$*-restart-mset-state* $cdcl_W$*-restart-mset.no-smaller-confl-def init-state-twl-def*)
show ‹*twl-struct-invs* (*init-state-twl N*)›
  using *struct tauto*
  by (*auto simp*: *twl-struct-invs-def twl-st-inv.simps clauses-to-update-prop.simps*
    past-invs.simps $cdcl_W$*-restart-mset-state init-state-twl-def*
    $cdcl_W$*-restart-mset.no-strange-atm-def*)
qed

lemma *full-cdcl-twl-stgy-*$cdcl_W$*-stgy-conclusive-from-init-state*:
  fixes *N* :: ‹′*v twl-clss*›
  assumes
    full-cdcl-twl-stgy: ‹*full cdcl-twl-stgy* (*init-state-twl N*) *T*› **and**
    struct: ‹∀ *C* ∈# *N*. *struct-wf-twl-cls C*› **and**
    no-tauto: ‹∀ *C* ∈# *N*. ¬*tautology* (*clause C*)›
  shows ‹*conflicting* ($state_W$*-of T*) = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*clause* ‘# *N*)) ∨
    (*conflicting* ($state_W$*-of T*) = *None* ∧ *trail* ($state_W$*-of T*) ⊨*asm clause* ‘# *N* ∧
    satisfiable (*set-mset* (*clause* ‘# *N*))))›
proof −
  have ‹*distinct-mset* (*clause C*)› **if** ‹*C* ∈# *N*› **for** *C*
    using *struct that* **by** (*cases C*) *auto*
  then have *dist*: ‹*distinct-mset-mset* (*clause* ‘# *N*)›
    using *struct* **by** (*auto simp*: *distinct-mset-set-def*)

  have ‹*twl-struct-invs* (*init-state-twl N*)›
    using *struct no-tauto* **by** (*rule twl-struct-invs-init-state-twl*)
  with *full-cdcl-twl-stgy*
  have ‹*full* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* ($state_W$*-of* (*init-state-twl N*)) ($state_W$*-of T*)›
    by (*rule full-cdcl-twl-stgy-*$cdcl_W$*-stgy*)
  then have ‹*full* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* (*init-state* (*clause* ‘# *N*)) ($state_W$*-of T*)›
    by (*simp add*: *init-state.simps init-state-twl-def*)
  then show *?thesis*
    by (*rule* $cdcl_W$*-restart-mset.full-*$cdcl_W$*-stgy-final-state-conclusive-from-init-state*)

(*use dist* **in** *auto*)
qed

lemma *cdcl-twl-o-twl-stgy-invs*:
‹*cdcl-twl-o S T* $\Longrightarrow$ *twl-struct-invs S* $\Longrightarrow$ *twl-stgy-invs S* $\Longrightarrow$ *twl-stgy-invs T*›
using $cdcl_W$*-restart-mset.rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-stgy-invariant cdcl-twl-stgy-*$cdcl_W$*-stgy other′* $cdcl_W$*-restart-mset.*$cdcl_W$*-restart-conflict-non-zero-unless-level-0*
unfolding *twl-struct-invs-def twl-stgy-invs-def*
apply (*intro conjI*)
apply *blast*
by (*smt* $cdcl_W$*-restart-mset.*$cdcl_W$*-restart-conflict-non-zero-unless-level-0* $cdcl_W$*-restart-mset.other cdcl-twl-o-*$cdcl_W$*-o twl-struct-invs-def twl-struct-invs-no-false-clause*)

Well-foundedness **lemma** *wf-*$cdcl_W$*-stgy-state*$_W$*-of*:
‹*wf* {(*T*, *S*). $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of S*) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* (*state*$_W$*-of S*) (*state*$_W$*-of T*)}›
using *wf-if-measure-f*[*OF* $cdcl_W$*-restart-mset.wf-*$cdcl_W$*-stgy*, *of state*$_W$*-of*] **by** *simp*

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

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

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

have *twl*: ‹*twl-struct-invs S*› **and** *cdcl*: ‹*cdcl-twl-cp S T*›
using *x-TWL x* **by** *auto*
have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*state*$_W$*-of S*)›
using *twl* **by** (*auto simp*: *twl-struct-invs-def*)
moreover have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* (*state*$_W$*-of S*) (*state*$_W$*-of T*) ∨
(*state*$_W$*-of S* = *state*$_W$*-of T* ∧
(*literals-to-update-measure T*, *literals-to-update-measure S*) ∈ *lexn less-than 2*)›
using *cdcl cdcl-twl-cp-*$cdcl_W$*-stgy twl* **by** *blast*
ultimately show ‹*x* ∈ *?CDCL* ∪ *?P*›
unfolding *x* **by** *blast*
qed
ultimately show *?thesis*

using *wf-subset*[*of* ‹*?CDCL* ∪ *?P*›] **by** *blast*
qed

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

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

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

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

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

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

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

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

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

lemma *tranclp-cdcl-twl-cp-stgyD*: ‹*cdcl-twl-cp*$^{++}$ *S T* $\Longrightarrow$ *cdcl-twl-stgy*$^{++}$ *S T*›
using *tranclp-mono*[*of cdcl-twl-cp cdcl-twl-stgy*] *cdcl-twl-stgy.intros*(*1*)
by *blast*

lemma *wf-cdcl-twl-o*:
‹*wf* {(*T*, *S*::′*v twl-st*). *twl-struct-invs S* ∧ *cdcl-twl-o S T*}›
by (*rule wf-subset*[*OF wf-cdcl-twl-stgy*]) (*auto intro*: *cdcl-twl-stgy.intros*)

lemma *tranclp-wf-cdcl-twl-o*:
‹*wf* {(*T*, *S*::′*v twl-st*). *twl-struct-invs S* ∧ *cdcl-twl-o*$^{++}$ *S T*}›
by (*rule wf-subset*[*OF tranclp-wf-cdcl-twl-stgy*]) (*auto dest*: *tranclp-cdcl-twl-o-stgyD*)

lemma (**in** −)*propa-cands-enqueued-mono*:
‹*U*′ ⊆# *U* $\Longrightarrow$ *N*′ ⊆# *N* $\Longrightarrow$
propa-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) $\Longrightarrow$
propa-cands-enqueued (*M*, *N*′, *U*′, *D*, *NE*′, *UE*′, *WS*, *Q*)›
by (*cases D*) (*auto 5 5*)

lemma (**in** −)*confl-cands-enqueued-mono*:
‹*U*′ ⊆# *U* $\Longrightarrow$ *N*′ ⊆# *N* $\Longrightarrow$
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) $\Longrightarrow$
confl-cands-enqueued (*M*, *N*′, *U*′, *D*, *NE*′, *UE*′, *WS*, *Q*)›
by (*cases D*) *auto*

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

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

lemma (**in** −) *rtranclp-cdcl-twl-stgy-twl-stgy-invs*:
assumes
‹*cdcl-twl-stgy*** *S T*› **and**
‹*twl-struct-invs S*› **and**
‹*twl-stgy-invs S*›
shows ‹*twl-stgy-invs T*›
using *assms* $cdcl_W$*-restart-mset.rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-stgy-invariant*
rtranclp-cdcl-twl-stgy-$cdcl_W$*-stgy*
by (*metis* $cdcl_W$*-restart-mset.rtranclp-*$cdcl_W$*-restart-conflict-non-zero-unless-level-0*
$cdcl_W$*-restart-mset.rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$*-restart twl-stgy-invs-def*
twl-struct-invs-def twl-struct-invs-no-false-clause)

lemma *after-fast-restart-replay*:
assumes
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*′, *N*, *U*, *None*)› **and**
stgy-invs: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant* (*M*′, *N*, *U*, *None*)› **and**
smaller-propa: ‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*′, *N*, *U*, *None*)› **and**
kept: ‹∀ *L E*. *Propagated L E* ∈ *set* (*drop* (*length M*′ − *n*) *M*′) ⟶ *E* ∈# *N* + *U*′› **and**
U′*-U*: ‹*U*′ ⊆# *U*›
shows

‹$cdcl_W$-restart-mset.$cdcl_W$-stgy** ([], N, U′, None) (drop (length M′ − n) M′, N, U′, None)›
proof −
  let ?S = ‹λn. (drop (length M′ − n) M′, N, U′, None)›
  note $cdcl_W$-restart-mset-state[simp]
  have
    M-lev: ‹$cdcl_W$-restart-mset.$cdcl_W$-M-level-inv (M′, N, U, None)› **and**
    alien: ‹$cdcl_W$-restart-mset.no-strange-atm (M′, N, U, None)› **and**
    confl: ‹$cdcl_W$-restart-mset.$cdcl_W$-conflicting (M′, N, U, None)› **and**
    learned: ‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clause (M′, N, U, None)›
    using inv **unfolding** $cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv-def **by** fast+

  have smaller-confl: ‹$cdcl_W$-restart-mset.no-smaller-confl (M′, N, U, None)›
    using stgy-invs **unfolding** $cdcl_W$-restart-mset.$cdcl_W$-stgy-invariant-def **by** blast
  have n-d: ‹no-dup M′›
    using M-lev **unfolding** $cdcl_W$-restart-mset.$cdcl_W$-M-level-inv-def **by** simp
  let ?L = ‹λm. M′ ! (length M′ − Suc m)›
  have undef-nth-Suc:
     ‹undefined-lit (drop (length M′ − m) M′) (lit-of (?L m))›
     if ‹m < length M′›
     for m
  proof −
    define k **where**
      ‹k = length M′ − Suc m›
    then have Sk: ‹length M′ − m = Suc k›
      using that **by** linarith
    have k-le-M′: ‹k < length M′›
      using that **unfolding** k-def **by** linarith
    have n-d′: ‹no-dup (take k M′ @ ?L m # drop (Suc k) M′)›
      using n-d
      apply (subst (asm) append-take-drop-id[symmetric, of - ‹Suc k›])
      apply (subst (asm) take-Suc-conv-app-nth)
       apply (rule k-le-M′)
      apply (subst k-def[symmetric])
      by simp

    show ?thesis
      using n-d′
      apply (subst (asm) no-dup-append-cons)
      apply (subst (asm) k-def[symmetric])+
      apply (subst k-def[symmetric])+
      apply (subst Sk)+
      by blast
  qed

  have atm-in:
    ‹atm-of (lit-of (M′ ! m)) ∈ atms-of-mm N›
    if ‹m < length M′›
    for m
    using alien that
    by (auto simp: $cdcl_W$-restart-mset.no-strange-atm-def lits-of-def)

  show ?thesis
    using kept
  proof (induction n)
    case 0
    then show ?case **by** simp

```
next
  case (Suc m) note IH = this(1) and kept = this(2)
  consider
    (le) ‹m < length M'› |
    (ge) ‹m ≥ length M'›
    by linarith
  then show ?case
  proof (cases)
    case ge
    then show ?thesis
      using Suc by auto
  next
    case le
    define k where
      ‹k = length M' − Suc m›
    then have Sk: ‹length M' − m = Suc k›
      using le by linarith
    have k-le-M': ‹k < length M'›
      using le unfolding k-def by linarith
    have kept': ‹∀ L E. Propagated L E ∈ set (drop (length M' − m) M') ⟶ E ∈# N + U'›
      using kept k-le-M' unfolding k-def[symmetric] Sk
      by (subst (asm) Cons-nth-drop-Suc[symmetric]) auto
    have M': ‹M' = take (length M' − Suc m) M' @ ?L m # trail (?S m)›
      apply (subst append-take-drop-id[symmetric, of - ‹Suc k›])
      apply (subst take-Suc-conv-app-nth)
       apply (rule k-le-M')
      apply (subst k-def[symmetric])
      unfolding k-def[symmetric] Sk
      by auto

    have ‹cdcl_W-restart-mset.cdcl_W-stgy (?S m) (?S (Suc m))›
    proof (cases ‹?L (m)›)
      case (Decided K) note K = this
      have dec: ‹cdcl_W-restart-mset.decide (?S m) (?S (Suc m))›
        apply (rule cdcl_W-restart-mset.decide-rule[of - ‹lit-of (?L m)›])
        subgoal by simp
        subgoal using undef-nth-Suc[of m] le by simp
        subgoal using le by (auto simp: atm-in)
        subgoal using le k-le-M' K unfolding k-def[symmetric] Sk
          by (auto simp: state-eq-def state-def Cons-nth-drop-Suc[symmetric])
        done
      have Dec: ‹M' ! k = Decided K›
        using K unfolding k-def[symmetric] Sk .

      have H: ‹D + {#L#} ∈# N + U ⟶ undefined-lit (trail (?S m)) L ⟶
          ¬ (trail (?S m)) ⊨as CNot D› for D L
        using smaller-propa unfolding cdcl_W-restart-mset.no-smaller-propa-def
          trail.simps clauses-def
          cdcl_W-restart-mset-state
        apply (subst (asm) M')
        unfolding Dec Sk k-def[symmetric]
        by (auto simp: clauses-def state-eq-def)
      have ‹D ∈# N ⟶ undefined-lit (trail (?S m)) L ⟶ L ∈# D ⟶
          ¬ (trail (?S m)) ⊨as CNot (remove1-mset L D)› and
        ‹D ∈# U' ⟶ undefined-lit (trail (?S m)) L ⟶ L ∈# D ⟶
          ¬ (trail (?S m)) ⊨as CNot (remove1-mset L D)›for D L
```

using *H*[*of* ‹*remove1-mset L D*› *L*] *U'-U* **by** *auto*
then have *nss*: ‹*no-step* $cdcl_W$*-restart-mset.propagate* (*?S m*)›
by (*auto simp*: $cdcl_W$*-restart-mset.propagate.simps clauses-def*
state-eq-def k-def[*symmetric*] *Sk*)

have *H*: ‹*D* ∈# *N* + *U'* ⟶ ¬ (*trail* (*?S m*)) ⊨*as CNot D*› **for** *D*
using *smaller-confl U'-U* **unfolding** $cdcl_W$*-restart-mset.no-smaller-confl-def*
trail.simps clauses-def $cdcl_W$*-restart-mset-state*
apply (*subst* (*asm*) *M'*)
unfolding *Dec Sk k-def*[*symmetric*]
by (*auto simp*: *clauses-def state-eq-def*)
then have *nsc*: ‹*no-step* $cdcl_W$*-restart-mset.conflict* (*?S m*)›
by (*auto simp*: $cdcl_W$*-restart-mset.conflict.simps clauses-def state-eq-def*
k-def[*symmetric*] *Sk*)
show *?thesis*
apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.other'*)
apply (*rule nsc*)
apply (*rule nss*)
apply (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-o.decide*)
apply (*rule dec*)
done
next
case *K*: (*Propagated K C*)
have *Propa*: ‹*M'* ! *k* = *Propagated K C*›
using *K* **unfolding** *k-def*[*symmetric*] *Sk* **.**
have
M-C: ‹*trail* (*?S m*) ⊨*as CNot* (*remove1-mset K C*)› **and**
K-C: ‹*K* ∈# *C*›
using *confl* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def trail.simps*
by (*subst* (*asm*)(*3*) *M'*; *auto simp*: *k-def*[*symmetric*] *Sk Propa*)+
have [*simp*]: ‹*k* − *min* (*length M'*) *k* = *0*›
unfolding *k-def* **by** *auto*
have *C-N-U*: ‹*C* ∈# *N* + *U'*›
using *learned kept* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def Sk*
k-def[*symmetric*] $cdcl_W$*-restart-mset.reasons-in-clauses-def*
apply (*subst* (*asm*)(*4*)*M'*)
apply (*subst* (*asm*)(*10*)*M'*)
unfolding *K*
by (*auto simp*: *K k-def*[*symmetric*] *Sk Propa clauses-def*)
have ‹$cdcl_W$*-restart-mset.propagate* (*?S m*) (*?S* (*Suc m*))›
apply (*rule* $cdcl_W$*-restart-mset.propagate-rule*[*of - C K*])
subgoal by *simp*
subgoal using *C-N-U* **by** (*simp add*: *clauses-def*)
subgoal using *K-C* **.**
subgoal using *M-C* **.**
subgoal using *undef-nth-Suc*[*of m*] *le K* **by** (*simp add*: *k-def*[*symmetric*] *Sk*)
subgoal
using *le k-le-M' K* **unfolding** *k-def*[*symmetric*] *Sk*
by (*auto simp*: *state-eq-def*
state-def Cons-nth-drop-Suc[*symmetric*])
done
then show *?thesis*
by (*rule* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy.propagate'*)
qed
then show *?thesis*
using *IH*[*OF kept'*] **by** *simp*

    qed
  qed
qed

lemma *after-fast-restart-replay-no-stgy*:
  assumes
    inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (M', N, U, *None*)› **and**
    kept: ‹∀ L E. *Propagated* L E ∈ *set* (*drop* (*length* M' − n) M') ⟶ E ∈# N + U'› **and**
    U'-U: ‹U' ⊆# U›
  shows
    ‹$cdcl_W$*-restart-mset.*$cdcl_W$** ([], N, U', *None*) (*drop* (*length* M' − n) M', N, U', *None*)›
proof −
  let *?S* = ‹λn. (*drop* (*length* M' − n) M', N, U', *None*)›
  note $cdcl_W$*-restart-mset-state*[*simp*]
  have
    M-lev: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* (M', N, U, *None*)› **and**
    alien: ‹$cdcl_W$*-restart-mset.no-strange-atm* (M', N, U, *None*)› **and**
    confl: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting* (M', N, U, *None*)› **and**
    learned: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause* (M', N, U, *None*)›
    using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* **by** *fast+*

  have *n-d*: ‹*no-dup* M'›
    using *M-lev* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** *simp*
  let *?L* = ‹λm. M' ! (*length* M' − *Suc* m)›
  have *undef-nth-Suc*:
     ‹*undefined-lit* (*drop* (*length* M' − m) M') (*lit-of* (*?L* m))›
     if ‹m < *length* M'›
     for m
  proof −
    define k **where**
      ‹k = *length* M' − *Suc* m›
    then have *Sk*: ‹*length* M' − m = *Suc* k›
      using *that* **by** *linarith*
    have *k-le-*M': ‹k < *length* M'›
      using *that* **unfolding** *k-def* **by** *linarith*
    have *n-d*$'$: ‹*no-dup* (*take* k M' @ *?L* m # *drop* (*Suc* k) M')›
      using *n-d*
      apply (*subst* (*asm*) *append-take-drop-id*[*symmetric*, *of* - ‹*Suc* k›])
      apply (*subst* (*asm*) *take-Suc-conv-app-nth*)
       apply (*rule* *k-le-*M')
      apply (*subst* *k-def*[*symmetric*])
      by *simp*

    show *?thesis*
      using *n-d*$'$
      apply (*subst* (*asm*) *no-dup-append-cons*)
      apply (*subst* (*asm*) *k-def*[*symmetric*])+
      apply (*subst* *k-def*[*symmetric*])+
      apply (*subst* *Sk*)+
      by *blast*
  qed

  have *atm-in*:
    ‹*atm-of* (*lit-of* (M' ! m)) ∈ *atms-of-mm* N›
    if ‹m < *length* M'›
    for m

using *alien that*
by (*auto simp*: *cdcl*$_W$*-restart-mset.no-strange-atm-def lits-of-def*)

show *?thesis*
using *kept*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*) **note** *IH* = *this*(*1*) **and** *kept* = *this*(*2*)
consider
(*le*) ‹*m* < *length M'*› |
(*ge*) ‹*m* ≥ *length M'*›
by *linarith*
then show *?case*
proof *cases*
case *ge*
then show *?thesis*
using *Suc* **by** *auto*
next
case *le*
define *k* **where**
‹*k* = *length M'* − *Suc m*›
then have *Sk*: ‹*length M'* − *m* = *Suc k*›
using *le* **by** *linarith*
have *k-le-M'*: ‹*k* < *length M'*›
using *le* **unfolding** *k-def* **by** *linarith*
have *kept'*: ‹∀ *L E*. *Propagated L E* ∈ *set* (*drop* (*length M'* − *m*) *M'*) ⟶ *E* ∈# *N* + *U'*›
using *kept k-le-M'* **unfolding** *k-def*[*symmetric*] *Sk*
by (*subst* (*asm*) *Cons-nth-drop-Suc*[*symmetric*]) *auto*
have *M'*: ‹*M'* = *take* (*length M'* − *Suc m*) *M'* @ *?L m* # *trail* (*?S m*)›
apply (*subst append-take-drop-id*[*symmetric*, *of* - ‹*Suc k*›])
apply (*subst take-Suc-conv-app-nth*)
apply (*rule k-le-M'*)
apply (*subst k-def*[*symmetric*])
unfolding *k-def*[*symmetric*] *Sk*
by *auto*

have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$ (*?S m*) (*?S* (*Suc m*))›
proof (*cases* ‹*?L* (*m*)›)
case (*Decided K*) **note** *K* = *this*
have *dec*: ‹*cdcl*$_W$*-restart-mset.decide* (*?S m*) (*?S* (*Suc m*))›
apply (*rule cdcl*$_W$*-restart-mset.decide-rule*[*of* - ‹*lit-of* (*?L m*)›])
subgoal by *simp*
subgoal using *undef-nth-Suc*[*of m*] *le* **by** *simp*
subgoal using *le* **by** (*auto simp*: *atm-in*)
subgoal using *le k-le-M' K* **unfolding** *k-def*[*symmetric*] *Sk*
by (*auto simp*: *state-eq-def state-def Cons-nth-drop-Suc*[*symmetric*])
done
have *Dec*: ‹*M'* ! *k* = *Decided K*›
using *K* **unfolding** *k-def*[*symmetric*] *Sk* **.**

show *?thesis*
apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*.intros*(*3*))
apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*-o.decide*)
apply (*rule dec*)

```
        done
      next
        case K: (Propagated K C)
        have Propa: ‹M' ! k = Propagated K C›
          using K unfolding k-def[symmetric] Sk .
        have
          M-C: ‹trail (?S m) ⊨as CNot (remove1-mset K C)› and
          K-C: ‹K ∈# C›
          using confl unfolding cdcl_W-restart-mset.cdcl_W-conflicting-def trail.simps
          by (subst (asm)(3) M'; auto simp: k-def[symmetric] Sk Propa)+
        have [simp]: ‹k - min (length M') k = 0›
          unfolding k-def by auto
        have C-N-U: ‹C ∈# N + U'›
          using learned kept unfolding cdcl_W-restart-mset.cdcl_W-learned-clause-alt-def Sk
            k-def[symmetric] cdcl_W-restart-mset.reasons-in-clauses-def
          apply (subst (asm)(4)M')
          apply (subst (asm)(10)M')
          unfolding K
          by (auto simp: K k-def[symmetric] Sk Propa clauses-def)
        have ‹cdcl_W-restart-mset.propagate (?S m) (?S (Suc m))›
          apply (rule cdcl_W-restart-mset.propagate-rule[of - C K])
          subgoal by simp
          subgoal using C-N-U by (simp add: clauses-def)
          subgoal using K-C .
          subgoal using M-C .
          subgoal using undef-nth-Suc[of m] le K by (simp add: k-def[symmetric] Sk)
          subgoal
            using le k-le-M' K unfolding k-def[symmetric] Sk
            by (auto simp: state-eq-def
                state-def Cons-nth-drop-Suc[symmetric])
          done
        then show ?thesis
          by (rule cdcl_W-restart-mset.cdcl_W.intros)
      qed
      then show ?thesis
        using IH[OF kept'] by simp
    qed
  qed
qed

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

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

lemma cdcl-twl-o-all-learned-diff-learned:
  assumes ‹cdcl-twl-o S T›
  shows
    ‹clause `# get-learned-clss S ⊆# clause `# get-learned-clss T ∧
```

get-init-learned-clss S ⊆# *get-init-learned-clss T*∧
get-all-init-clss S = *get-all-init-clss T*⟩
by (*use assms* **in** ⟨*induction rule*: *cdcl-twl-o.induct*⟩)
(*auto simp*: *update-clauses.simps size-Suc-Diff1*)

lemma *cdcl-twl-cp-all-learned-diff-learned*:
assumes ⟨*cdcl-twl-cp S T*⟩
shows
⟨*clause* ‘# *get-learned-clss S* = *clause* ‘# *get-learned-clss T* ∧
get-init-learned-clss S = *get-init-learned-clss T* ∧
get-all-init-clss S = *get-all-init-clss T*⟩
apply (*use assms* **in** ⟨*induction rule*: *cdcl-twl-cp.induct*⟩)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *D*
by (*cases D*)
(*auto simp*: *update-clauses.simps size-Suc-Diff1 dest*!: *multi-member-split*)
done

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

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

lemma *rtranclp-cdcl-twl-stgy-all-learned-diff-learned-size*:
assumes ⟨*cdcl-twl-stgy*** *S T*⟩
shows
⟨*size* (*get-all-learned-clss T*) − *size* (*get-all-learned-clss S*) ≥
size (*get-learned-clss T*) − *size* (*get-learned-clss S*)⟩
using *rtranclp-cdcl-twl-stgy-all-learned-diff-learned*[*OF assms*]
apply (*cases S*, *cases T*)
using *size-mset-mono* **by** *force*+

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

lemma *tranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*:
assumes *ST*: ‹*cdcl-twl-stgy*$^{++}$ *S T*› **and**
twl: ‹*twl-struct-invs S*› **and**
‹*clauses-to-update S* = {#}› **and**
‹*literals-to-update S* = {#}›
shows ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*$^{++}$ (*state*$_W$*-of S*) (*state*$_W$*-of T*)›
proof −
obtain *S'* **where**
SS': ‹*cdcl-twl-stgy S S'*› **and**
S'T: ‹*cdcl-twl-stgy*** *S' T*›
using *ST* **unfolding** *tranclp-unfold-begin* **by** *blast*

have *1*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy* (*state*$_W$*-of S*) (*state*$_W$*-of S'*)›
using *cdcl-twl-stgy-cdcl*$_W$*-stgy3*[*OF SS' assms*(*2*−*4*)]
by *blast*
have *struct-S'*: ‹*twl-struct-invs S'*›
using *twl SS'* **by** (*blast intro*: *cdcl-twl-stgy-twl-struct-invs*)
have *2*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** (*state*$_W$*-of S'*) (*state*$_W$*-of T*)›
apply (*rule rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*)
apply (*rule S'T*)
by (*rule struct-S'*)
show *?thesis*
using *1 2* **by** *auto*
qed

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

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

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

S clauses-def cdcl$_W$-restart-mset.cdcl$_W$-conflicting-def cdcl$_W$-restart-mset-state
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init-def
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def cdcl$_W$-restart-mset.cdcl$_W$-learned-clause-alt-def)
have ‹*set-mset N* ∪ *set-mset U* ⊨*ps CNot D*›
by (*rule true-clss-clss-true-clss-cls-true-clss-clss*[*OF N-U-M M-D*])
then have ‹*set-mset N* ⊨*ps CNot D*› ‹*set-mset N* ⊨*p D*›
using *N-U N-U-D true-clss-clss-left-right* **by** *blast+*
then have ‹*unsatisfiable* (*set-mset N*)›
by (*rule true-clss-clss-CNot-true-clss-cls-unsatisfiable*)

then show *?thesis*
by (*auto simp*: *S clauses-def cdcl$_W$-restart-mset-state dest*: *satisfiable-decreasing*)
qed

lemma *conflict-of-level-unsatisfiable2*:
assumes
struct: ‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv S*› **and**
dec: ‹*count-decided* (*trail S*) = *0*› **and**
confl: ‹*conflicting S* ≠ *None*›
shows ‹*unsatisfiable* (*set-mset* (*init-clss S* + *learned-clss S*))›
proof −
obtain *M N U D* **where** *S*: ‹*S* = (*M*, *N*, *U*, *Some D*)›
by (*cases S*) (*use confl* **in** ‹*auto simp*: *cdcl$_W$-restart-mset-state*›)
have [*simp*]: ‹*get-all-ann-decomposition M* = [([], *M*)]›
by (*rule no-decision-get-all-ann-decomposition*)
(*use dec* **in** ‹*auto simp*: *count-decided-def filter-empty-conv S cdcl$_W$-restart-mset-state*›)
have
M-D: ‹*M* ⊨*as CNot D*› **and**
N-U-M: ‹*set-mset N* ∪ *set-mset U* ⊨*ps unmark-l M*› **and**
n-d: ‹*no-dup M*› **and**
N-U-D: ‹*set-mset N* ∪ *set-mset U* ⊨*p D*›
using *assms*
by (*auto simp*: *cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def all-decomposition-implies-def*
S clauses-def cdcl$_W$-restart-mset.cdcl$_W$-conflicting-def cdcl$_W$-restart-mset-state
cdcl$_W$-restart-mset.cdcl$_W$-learned-clauses-entailed-by-init-def
cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv-def cdcl$_W$-restart-mset.cdcl$_W$-learned-clause-alt-def)
have ‹*set-mset N* ∪ *set-mset U* ⊨*ps CNot D*›
by (*rule true-clss-clss-true-clss-cls-true-clss-clss*[*OF N-U-M M-D*])
then have ‹*set-mset N* ∪ *set-mset U* ⊨*ps CNot D*› ‹*set-mset N* ∪ *set-mset U* ⊨*p D*›
using *N-U-D true-clss-clss-left-right* **by** *blast+*
then have ‹*unsatisfiable* (*set-mset N* ∪ *set-mset U*)›
by (*rule true-clss-clss-CNot-true-clss-cls-unsatisfiable*)

then show *?thesis*
by (*auto simp*: *S clauses-def cdcl$_W$-restart-mset-state dest*: *satisfiable-decreasing*)
qed

end
theory *Watched-Literals-Algorithm*
imports
WB-More-Refinement
Watched-Literals-Transition-System
begin

1.2 First Refinement: Deterministic Rule Application

1.2.1 Unit Propagation Loops

definition *set-conflicting* :: ‹′v twl-cls ⇒ ′v twl-st ⇒ ′v twl-st› **where**
‹set-conflicting = (λC (M, N, U, D, NE, UE, WS, Q). (M, N, U, Some (clause C), NE, UE, {#}, {#}))›

definition *propagate-lit* :: ‹′v literal ⇒ ′v twl-cls ⇒ ′v twl-st ⇒ ′v twl-st› **where**
‹propagate-lit = (λL′ C (M, N, U, D, NE, UE, WS, Q).
    (Propagated L′ (clause C) # M, N, U, D, NE, UE, WS, add-mset (−L′) Q))›

definition *update-clauseS* :: ‹′v literal ⇒ ′v twl-cls ⇒ ′v twl-st ⇒ ′v twl-st nres› **where**
‹update-clauseS = (λL C (M, N, U, D, NE, UE, WS, Q). do {
      K ← SPEC (λL. L ∈# unwatched C ∧ −L ∉ lits-of-l M);
      if K ∈ lits-of-l M
      then RETURN (M, N, U, D, NE, UE, WS, Q)
      else do {
        (N′, U′) ← SPEC (λ(N′, U′). update-clauses (N, U) C L K (N′, U′));
        RETURN (M, N′, U′, D, NE, UE, WS, Q)
      }
  })›

definition *unit-propagation-inner-loop-body* :: ‹′v literal ⇒ ′v twl-cls ⇒
  ′v twl-st ⇒ ′v twl-st nres› **where**
  ‹unit-propagation-inner-loop-body = (λL C S. do {
    do {
      bL′ ← SPEC (λK. K ∈# clause C);
      if bL′ ∈ lits-of-l (get-trail S)
      then RETURN S
      else do {
        L′ ← SPEC (λK. K ∈# watched C − {#L#});
        ASSERT (watched C = {#L, L′#});
        if L′ ∈ lits-of-l (get-trail S)
        then RETURN S
        else
          if ∀L ∈# unwatched C. −L ∈ lits-of-l (get-trail S)
          then
            if −L′ ∈ lits-of-l (get-trail S)
            then do {RETURN (set-conflicting C S)}
            else do {RETURN (propagate-lit L′ C S)}
          else do {
            update-clauseS L C S
          }
      }
    }
  })
›

definition *unit-propagation-inner-loop* :: ‹′v twl-st ⇒ ′v twl-st nres› **where**
  ‹unit-propagation-inner-loop S_0 = do {
    n ← SPEC(λ-::nat. True);
   (S, -) ← $WHILE_T$$^{λ(S, n).\ twl\text{-}struct\text{-}invs\ S\ ∧\ twl\text{-}stgy\text{-}invs\ S\ ∧\ cdcl\text{-}twl\text{-}cp^{**}\ S_0\ S\ ∧}$ (clauses-to-update S ≠ {#} ∨ n
      (λ(S, n). clauses-to-update S ≠ {#} ∨ n > 0)
      (λ(S, n). do {
        b ← SPEC(λb. (b ⟶ n > 0) ∧ (¬b ⟶ clauses-to-update S ≠ {#}));

if $\neg b$ *then do* {
  ASSERT(*clauses-to-update* $S \neq \{\#\}$);
  $(L,\ C) \leftarrow$ *SPEC* ($\lambda C.\ C \in\#$ *clauses-to-update* S);
  let S' = *set-clauses-to-update* (*clauses-to-update* $S - \{\#(L,\ C)\#\}$) S;
  $T \leftarrow$ *unit-propagation-inner-loop-body* L C S';
  RETURN (T, *if get-conflict* T = *None then* n *else* 0)
} *else do* { ~~This branch allows us to do skip some clauses.~~
  RETURN (S, $n - 1$)
}
})
(S_0, n);
RETURN S
}
⟩

lemma *unit-propagation-inner-loop-body*:
fixes S :: ⟨′*v twl-st*⟩
assumes
⟨*clauses-to-update* $S \neq \{\#\}$⟩ **and**
x-WS: ⟨$(L,\ C) \in\#$ *clauses-to-update* S⟩ **and**
inv: ⟨*twl-struct-invs* S⟩ **and**
inv-s: ⟨*twl-stgy-invs* S⟩ **and**
confl: ⟨*get-conflict* S = *None*⟩
shows
⟨*unit-propagation-inner-loop-body* L C
(*set-clauses-to-update* (*remove1-mset* $(L,\ C)$ (*clauses-to-update* S)) S)
$\leq$ (*SPEC* ($\lambda T'$. *twl-struct-invs* $T' \wedge$ *twl-stgy-invs* $T' \wedge$ *cdcl-twl-cp*** S $T' \wedge$
$(T',\ S) \in$ *measure* (*size* $\circ$ *clauses-to-update*)))⟩ (**is** *?spec*) **and**
⟨*nofail* (*unit-propagation-inner-loop-body* L C
(*set-clauses-to-update* (*remove1-mset* $(L,\ C)$ (*clauses-to-update* S)) S))⟩ (**is** *?fail*)
proof −
obtain *M N U D NE UE WS Q* **where**
S: ⟨$S = (M,\ N,\ U,\ D,\ NE,\ UE,\ WS,\ Q)$⟩
by (*cases* S) *auto*

have ⟨$C \in\# N + U$⟩ **and** *struct*: ⟨*struct-wf-twl-cls* C⟩ **and** *L-C*: ⟨$L \in\#$ *watched* C⟩
using *inv multi-member-split*[*OF x-WS*]
unfolding *twl-struct-invs-def twl-st-inv.simps S*
by *force+*
show *?fail*
unfolding *unit-propagation-inner-loop-body-def Let-def S*
by (*cases* C) (*use struct L-C* **in** ⟨*auto simp*: *refine-pw-simps S size-2-iff update-clauseS-def*⟩)
note [[*goals-limit=15*]]
show *?spec*
using *assms* **unfolding** *unit-propagation-inner-loop-body-def update-clause.simps*
proof (*refine-vcg*; (*unfold prod.inject clauses-to-update.simps set-clauses-to-update.simps*
ball-simps)?; *clarify?*; (*unfold triv-forall-equality*)?)
fix L' :: ⟨′*v literal*⟩
assume
⟨*clauses-to-update* $S \neq \{\#\}$⟩ **and**
WS: ⟨$(L,\ C) \in\#$ *clauses-to-update* S⟩ **and**
twl-inv: ⟨*twl-struct-invs* S⟩
have ⟨$C \in\# N + U$⟩ **and** *struct*: ⟨*struct-wf-twl-cls* C⟩ **and** *L-C*: ⟨$L \in\#$ *watched* C⟩
using *twl-inv WS* **unfolding** *twl-struct-invs-def twl-st-inv.simps S* **by** (*auto*; *fail*)+

define WS' **where** ⟨$WS' = WS - \{\#(L,\ C)\#\}$⟩

have *WS-WS'*: ‹*WS* = *add-mset* (*L*, *C*) *WS'*›
  using *WS* **unfolding** *WS'-def S* **by** *auto*

have *D*: ‹*D* = *None*›
  using *confl S* **by** *auto*

let *?S'* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *C*) *WS'*, *Q*)›
let *?T* = ‹(*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)›
let *?T'* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *WS'*, *Q*)›

{ — blocking literal
  fix *K'*
  assume
      K': ‹*K'* ∈# *clause C*› **and**
      L': ‹*K'* ∈ *lits-of-l* (*get-trail ?T*)›

  have ‹*cdcl-twl-cp ?S' ?T'*›
    by (*rule cdcl-twl-cp.delete-from-working*) (*use L' K' S* **in** *simp-all*)

  then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
    using *L' D* **by** (*simp add*: *S WS-WS'*)
  show ‹*twl-struct-invs ?T*›
    using *cdcl inv D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-struct-invs*)

  show ‹*twl-stgy-invs ?T*›
    using *cdcl inv-s inv D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)

  show ‹*cdcl-twl-cp*** *S ?T*›
    using *D WS-WS' cdcl* **by** *auto*

  show ‹(*?T*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*)›
    by (*simp add*: *WS'-def*[*symmetric*] *WS-WS' S*)

}

assume *L'*: ‹*L'* ∈# *remove1-mset L* (*watched C*)›
show *watched*: ‹*watched C* = {#*L*, *L'*#}›
  by (*cases C*) (*use struct L-C L'* **in** ‹*auto simp*: *size-2-iff*›)
then have *L-C'*: ‹*L* ∈# *clause C*› **and** *L'-C'*: ‹*L'* ∈# *clause C*›
  by (*cases C*; *auto*; *fail*)+

{ — if *L'* ∈ *lits-of-l M*, then:
  assume *L'*: ‹*L'* ∈ *lits-of-l* (*get-trail ?T*)›

  have ‹*cdcl-twl-cp ?S' ?T'*›
    by (*rule cdcl-twl-cp.delete-from-working*) (*use L' L'-C' watched S* **in** *simp-all*)

  then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
    using *L' watched D* **by** (*simp add*: *S WS-WS'*)
  show ‹*twl-struct-invs ?T*›
    using *cdcl inv D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-struct-invs*)

  show ‹*twl-stgy-invs ?T*›
    using *cdcl inv-s inv D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)

  show ‹*cdcl-twl-cp*** *S ?T*›

using *D WS-WS′ cdcl* **by** *auto*

show ‹(*?T*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*)›
by (*simp add*: *WS′-def*[*symmetric*] *WS-WS′ S*)

}
— if L' ∈ *lits-of-l M*, else:
let *?M* = ‹*get-trail ?T*›
assume L': ‹L' ∉ *lits-of-l ?M*›
{
{ — if ∀ *La*∈#*unwatched C*. − *La* ∈ *lits-of-l* (*get-trail* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)), then
assume *unwatched*: ‹∀ *L*∈#*unwatched C*. − *L* ∈ *lits-of-l ?M*›

{ — if − L' ∈ *lits-of-l* (*get-trail* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)) then
let *?T′* = ‹(*M*, *N*, *U*, *Some* (*clause C*), *NE*, *UE*, {#}, {#})›
let *?T* = ‹*set-conflicting C* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)›
assume *uL′*: ‹−L' ∈ *lits-of-l ?M*›
have *cdcl*: ‹*cdcl-twl-cp ?S′ ?T′*›
by (*rule cdcl-twl-cp.conflict*) (*use uL′ L′ watched unwatched S* **in** *simp-all*)
then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
using *uL′ L′ watched unwatched* **by** (*simp add*: *set-conflicting-def WS-WS′ S D*)

show ‹*twl-struct-invs ?T*›
using *cdcl inv D* **unfolding** *WS-WS′*
by (*force intro*: *cdcl-twl-cp-twl-struct-invs*)
show ‹*twl-stgy-invs ?T*›
using *cdcl inv inv-s D* **unfolding** *WS-WS′*
by (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)
show ‹$cdcl\text{-}twl\text{-}cp^{**}$ *S ?T*›
using *D WS-WS′ cdcl S* **by** *auto*
show ‹(*?T*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*)›
by (*simp add*: *S WS′-def*[*symmetric*] *WS-WS′ set-conflicting-def*)
}

{ — if − L' ∈ *lits-of-l M* else
let *?S* = ‹(*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)›
let *?T′* = ‹(*Propagated L′* (*clause C*) # *M*, *N*, *U*, *None*, *NE*, *UE*, *WS′*, *add-mset* (− L') *Q*)›
let *?S′* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *C*) *WS′*, *Q*)›
let *?T* = ‹*propagate-lit L′ C* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)›
assume *uL′*: ‹− L' ∉ *lits-of-l ?M*›

have *undef*: ‹*undefined-lit M L′*›
using *uL′ L′* **by** (*auto simp*: *S defined-lit-map lits-of-def atm-of-eq-atm-of*)

have *cdcl*: ‹*cdcl-twl-cp ?S′ ?T′*›
by (*rule cdcl-twl-cp.propagate*) (*use uL′ L′ undef watched unwatched D S* **in** *simp-all*)
then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
using *uL′ L′ undef watched unwatched D S WS-WS′* **by** (*simp add*: *propagate-lit-def*)

show ‹*twl-struct-invs ?T*›
using *cdcl inv D* **unfolding** *S WS-WS′* **by** (*force intro*: *cdcl-twl-cp-twl-struct-invs*)

show ‹*cdcl-twl-cp*** *S ?T*›
  using *cdcl D WS-WS'* **by** *force*
show ‹*twl-stgy-invs ?T*›
  using *cdcl inv inv-s D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)
show ‹(*?T*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*)›
  by (*simp add*: *WS'-def*[*symmetric*] *WS-WS' S propagate-lit-def*)
}
}

fix *La*
— if ∀ *L*∈#*unwatched C*. − *L* ∈ *lits-of-l M*, else
{
let *?S* = ‹(*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)›
let *?S'* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *C*) *WS'*, *Q*)›
let *?T* = ‹*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*›
fix *K M' N' U' D' WS'' NE' UE' Q' N'' U''*
have ‹*update-clauseS L C* (*set-clauses-to-update* (*remove1-mset* (*L*, *C*) (*clauses-to-update S*)) *S*)
  ≤ *SPEC* (λ*S'*. *twl-struct-invs S'* ∧ *twl-stgy-invs S'* ∧ *cdcl-twl-cp*** *S S'* ∧
  (*S'*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*))› (**is** *?upd*)
  apply (*rewrite at* ‹*set-clauses-to-update* - ⌑› *S*)
  apply (*rewrite at* ‹*clauses-to-update* ⌑› *S*)
  unfolding *update-clauseS-def clauses-to-update.simps set-clauses-to-update.simps*
  apply *clarify*
proof *refine-vcg*
  fix *x xa a b*
  assume *K*: ‹*x* ∈# *unwatched C* ∧ − *x* ∉ *lits-of-l M*›
  have *uL*: ‹− *L* ∈ *lits-of-l M*›
    using *inv* **unfolding** *twl-struct-invs-def S WS-WS'* **by** *auto*
  { — BLIT
    let *?T* = ‹(*M*, *N*, *U*, *D*, *NE*, *UE*, *remove1-mset* (*L*, *C*) *WS*, *Q*)›
    let *?T'* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, *WS'*, *Q*)›

    assume ‹*x* ∈ *lits-of-l M*›
    have *uL*: ‹− *L* ∈ *lits-of-l M*›
      using *inv* **unfolding** *twl-struct-invs-def S WS-WS'* **by** *auto*
    have ‹*L* ∈# *clause C*› ‹*x* ∈# *clause C*›
      using *watched K* **by** (*cases C*; *simp*; *fail*)+
    have ‹*cdcl-twl-cp ?S' ?T'*›
      by (*rule cdcl-twl-cp.delete-from-working*[*OF* ‹*x* ∈# *clause C*› ‹*x* ∈ *lits-of-l M*›])
    then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
      by (*auto simp*: *S D WS-WS'*)

    show ‹*twl-struct-invs ?T*›
      using *cdcl inv D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-struct-invs*)

    have *uL*: ‹− *L* ∈ *lits-of-l M*›
      using *inv* **unfolding** *twl-struct-invs-def S WS-WS'* **by** *auto*

    show ‹*twl-stgy-invs ?T*›
      using *cdcl inv inv-s D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)
    show ‹*cdcl-twl-cp*** *S ?T*›
      using *D WS-WS' cdcl* **by** *auto*
    show ‹(*?T*, *S*) ∈ *measure* (*size* ∘ *clauses-to-update*)›
      by (*simp add*: *WS'-def*[*symmetric*] *WS-WS' S*)
  }

assume
update: ‹*case xa of* $(N', U') \Rightarrow$ *update-clauses* (N, U) C L x (N', U')› **and**
[*simp*]: ‹$xa = (a, b)$›
let $?T' =$ ‹$(M, a, b, None, NE, UE, WS', Q)$›
let $?T =$ ‹$(M, a, b, D, NE, UE,$ *remove1-mset* (L, C) $WS, Q)$›
have ‹*cdcl-twl-cp* $?S'$ $?T'$›
by (*rule cdcl-twl-cp.update-clause*)
(*use uL* L' *K update watched S* **in** ‹*simp-all add*: *true-annot-iff-decided-or-true-lit*›)
then have *cdcl*: ‹*cdcl-twl-cp S ?T*›
by (*auto simp*: *S D WS-WS'*)

show ‹*twl-struct-invs ?T*›
using *cdcl inv D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-struct-invs*)

have *uL*: ‹$-$ $L \in$ *lits-of-l M*›
using *inv* **unfolding** *twl-struct-invs-def S WS-WS'* **by** *auto*

show ‹*twl-stgy-invs ?T*›
using *cdcl inv inv-s D* **unfolding** *S WS-WS'* **by** (*force intro*: *cdcl-twl-cp-twl-stgy-invs*)
show ‹*cdcl-twl-cp*** *S ?T*›
using *D WS-WS' cdcl* **by** *auto*
show ‹$(?T, S) \in$ *measure* (*size* $\circ$ *clauses-to-update*)›
by (*simp add*: *WS'-def*[*symmetric*] *WS-WS' S*)

qed
moreover assume ‹$\neg$*?upd*›
ultimately show ‹$-$ $La \in$
lits-of-l (*get-trail* (*set-clauses-to-update* (*remove1-mset* (L, C) (*clauses-to-update S*)) *S*))›
by *fast*
}
}
qed
qed

declare *unit-propagation-inner-loop-body*(*1*)[*THEN order-trans*, *refine-vcg*]

lemma *unit-propagation-inner-loop*:
assumes ‹*twl-struct-invs S*› **and** *inv*: ‹*twl-stgy-invs S*› **and** ‹*get-conflict* $S =$ *None*›
shows ‹*unit-propagation-inner-loop* $S \leq$ *SPEC* ($\lambda S'$. *twl-struct-invs* $S' \wedge$ *twl-stgy-invs* $S' \wedge$
cdcl-twl-cp** S $S' \wedge$ *clauses-to-update* $S' = \{\#\}$)›
unfolding *unit-propagation-inner-loop-def*
apply (*refine-vcg WHILEIT-rule*[**where** $R =$ ‹*measure* ($\lambda(S, n)$. (*size o clauses-to-update*) $S + n$)›])
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp add*: *twl-struct-invs-def*)

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

declare *unit-propagation-inner-loop*[*THEN order-trans*, *refine-vcg*]

definition *unit-propagation-outer-loop* :: ‹′*v twl-st* ⇒ ′*v twl-st nres*› **where**
‹*unit-propagation-outer-loop* S_0 =
$WHILE_T{}^{\lambda S.\ twl\text{-}struct\text{-}invs\ S\ \wedge\ twl\text{-}stgy\text{-}invs\ S\ \wedge\ cdcl\text{-}twl\text{-}cp^{**}\ S_0\ S\ \wedge\ clauses\text{-}to\text{-}update\ S\ =\ \{\#\}}$
(λ*S*. *literals-to-update S* ≠ {#})
(λ*S*. *do* {
L ← *SPEC* (λ*L*. *L* ∈# *literals-to-update S*);
let S′ = *set-clauses-to-update* {#(*L*, *C*)|*C* ∈# *get-clauses S*. *L* ∈# *watched C*#}
(*set-literals-to-update* (*literals-to-update S* − {#*L*#}) *S*);
ASSERT(*cdcl-twl-cp S S*′);
unit-propagation-inner-loop S′
})
S_0
›

abbreviation *unit-propagation-outer-loop-spec* **where**
‹*unit-propagation-outer-loop-spec S S*′ ≡ *twl-struct-invs S*′ ∧ *cdcl-twl-cp*** *S S*′ ∧
literals-to-update S′ = {#} ∧ (∀ *S*′*a*. ¬ *cdcl-twl-cp S*′ *S*′*a*) ∧ *twl-stgy-invs S*′›

lemma *unit-propagation-outer-loop*:
assumes ‹*twl-struct-invs S*› **and** ‹*clauses-to-update S* = {#}› **and** *confl*: ‹*get-conflict S* = *None*› **and**
‹*twl-stgy-invs S*›
shows ‹*unit-propagation-outer-loop S* ≤ *SPEC* (λ*S*′. *twl-struct-invs S*′ ∧ *cdcl-twl-cp*** *S S*′ ∧
literals-to-update S′ = {#} ∧ *no-step cdcl-twl-cp S*′ ∧ *twl-stgy-invs S*′)›
proof −

have *assert-twl-cp*: ‹*cdcl-twl-cp T*
(*set-clauses-to-update* (*Pair L* ‘# {#*Ca* ∈# *get-clauses T*. *L* ∈# *watched Ca*#})
(*set-literals-to-update* (*remove1-mset L* (*literals-to-update T*)) *T*))› (**is** *?twl*) **and**
assert-twl-struct-invs:
‹*twl-struct-invs* (*set-clauses-to-update* (*Pair L* ‘# {#*Ca* ∈# *get-clauses T*. *L* ∈# *watched Ca*#})
(*set-literals-to-update* (*remove1-mset L* (*literals-to-update T*)) *T*))›
(**is** ‹*twl-struct-invs ?T*′›) **and**
assert-stgy-invs:
‹*twl-stgy-invs* (*set-clauses-to-update* (*Pair L* ‘# {#*Ca* ∈# *get-clauses T*. *L* ∈# *watched Ca*#})
(*set-literals-to-update* (*remove1-mset L* (*literals-to-update T*)) *T*))› (**is** *?stgy*)
if
p: ‹*literals-to-update T* ≠ {#}› **and**
L-T: ‹*L* ∈# *literals-to-update T*› **and**
invs: ‹*twl-struct-invs T* ∧ *twl-stgy-invs T* ∧*cdcl-twl-cp*** *S T* ∧ *clauses-to-update T* = {#}›
for *L T*
proof −

from *that* **have**
p: ‹*literals-to-update* $T \neq \{\#\}$› **and**
L-T: ‹$L \in\#$ *literals-to-update* T› **and**
struct-invs: ‹*twl-struct-invs* T› **and**
‹*cdcl-twl-cp*** S T› **and**
w-q: ‹*clauses-to-update* $T = \{\#\}$›
by *fast+*
have ‹*get-conflict* T = *None*›
using *w-q p invs* **unfolding** *twl-struct-invs-def* **by** *auto*
then obtain *M N U NE UE Q* **where**
T: ‹T = (M, N, U, *None*, *NE*, *UE*, $\{\#\}$, Q)›
using *w-q p* **by** (*cases* T) *auto*
define Q' **where** ‹Q' = *remove1-mset* L Q›
have *Q*: ‹Q = *add-mset* L Q'›
using *L-T* **unfolding** Q'*-def* T **by** *auto*

— Show assertion that one step has been done
show *twl*: *?twl*
unfolding *T set-clauses-to-update.simps set-literals-to-update.simps literals-to-update.simps* Q'*-def*[*symmetric*]
unfolding *Q get-clauses.simps*
by (*rule cdcl-twl-cp.pop*)
then show ‹*twl-struct-invs ?T'*›
using *cdcl-twl-cp-twl-struct-invs struct-invs* **by** *blast*

then show *?stgy*
using *twl cdcl-twl-cp-twl-stgy-invs*[*OF twl*] *invs* **by** *blast*
qed

show *?thesis*
unfolding *unit-propagation-outer-loop-def*
apply (*refine-vcg WHILEIT-rule*[**where** R = ‹$\{(T, S).$ *twl-struct-invs* $S \wedge$ *cdcl-twl-cp*$^{++}$ S $T\}$›])
apply ((*simp-all add*: *assms tranclp-wf-cdcl-twl-cp*; *fail*)+)[*6*]
subgoal by (*rule assert-twl-cp*) — Assertion
subgoal by (*rule assert-twl-struct-invs*) — WHILE-loop invariants
subgoal by (*rule assert-stgy-invs*)
subgoal for *S L*
by (*cases S*)
(*auto simp*: *twl-st twl-struct-invs-def*)
subgoal by (*simp*; *fail*)
subgoal by *auto*
subgoal by *auto*
subgoal by *simp*
subgoal by *auto* — Termination
subgoal — Final invariants
by *simp*
subgoal by *simp*
subgoal by *auto*
subgoal by (*auto simp*: *cdcl-twl-cp.simps*)
subgoal by *simp*
done
qed
declare *unit-propagation-outer-loop*[*THEN order-trans*, *refine-vcg*]

1.2.2 Other Rules

Decide

definition *find-unassigned-lit* :: ‹'v twl-st ⇒ 'v literal option nres› **where**
‹find-unassigned-lit = (λS.
SPEC (λL.
(L ≠ None ⟶ undefined-lit (get-trail S) (the L) ∧
atm-of (the L) ∈ atms-of-mm (get-all-init-clss S)) ∧
(L = None ⟶ (∄L. undefined-lit (get-trail S) L ∧
atm-of L ∈ atms-of-mm (get-all-init-clss S)))))›

definition *propagate-dec* **where**
‹propagate-dec = (λL (M, N, U, D, NE, UE, WS, Q). (Decided L # M, N, U, D, NE, UE, WS, {#−L#}))›

definition *decide-or-skip* :: ‹'v twl-st ⇒ (bool × 'v twl-st) nres› **where**
‹decide-or-skip S = do {
L ← find-unassigned-lit S;
case L of
None ⇒ RETURN (True, S)
| Some L ⇒ RETURN (False, propagate-dec L S)
}
›

lemma *decide-or-skip-spec*:
assumes ‹clauses-to-update S = {#}› **and** ‹literals-to-update S = {#}› **and** ‹get-conflict S = None›
and
twl: ‹twl-struct-invs S› **and** *twl-s*: ‹twl-stgy-invs S›
shows ‹decide-or-skip S ≤ SPEC(λ(brk, T). cdcl-twl-o** S T ∧
get-conflict T = None ∧
no-step cdcl-twl-o T ∧ (brk ⟶ no-step cdcl-twl-stgy T) ∧ twl-struct-invs T ∧
twl-stgy-invs T ∧ clauses-to-update T = {#} ∧
(¬brk ⟶ literals-to-update T ≠ {#}) ∧
(¬no-step cdcl-twl-o S ⟶ cdcl-twl-o++ S T))›
proof −
obtain *M N U NE UE* **where** *S*: ‹S = (M, N, U, None, NE, UE, {#}, {#})›
using *assms* **by** (*cases S*) *auto*
have *atm-N-U*:
‹atm-of L ∈ atms-of-mm (clauses N + NE)›
if *U*: ‹atm-of L ∈ atms-of-ms (clause ‘ set-mset U)› **and**
undef: ‹undefined-lit M L›
for *L*
proof −
have ‹$cdcl_W$-restart-mset.no-strange-atm ($state_W$-of S)› **and** *unit*: ‹entailed-clss-inv S›
using *twl* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *fast+*
then show *?thesis*
using *that*
by (*auto simp*: $cdcl_W$*-restart-mset.no-strange-atm-def S* $cdcl_W$*-restart-mset-state image-Un*)
qed
{
fix *L*
assume *undef*: ‹undefined-lit M L› **and** *L*: ‹atm-of L ∈ atms-of-mm (clauses N + NE)›
let *?T* = ‹(Decided L # M, N, U, None, NE, UE, {#}, {#− L#})›
have *o*: ‹cdcl-twl-o (M, N, U, None, NE, UE, {#}, {#}) ?T›

by (*rule cdcl-twl-o.decide*) (*use undef L* **in** *auto*)
have *twl'*: ‹*twl-struct-invs ?T*›
using *S cdcl-twl-o-twl-struct-invs o twl* **by** *blast*
have *twl-s'*: ‹*twl-stgy-invs ?T*›
using *S cdcl-twl-o-twl-stgy-invs o twl twl-s* **by** *blast*
note *o twl' twl-s'*
} **note** $H = this$
show *?thesis*
using *assms* **unfolding** *S find-unassigned-lit-def propagate-dec-def decide-or-skip-def*
apply (*refine-vcg*)
subgoal by *fast*
subgoal by *blast*
subgoal by (*force simp*: *H elim*!: *cdcl-twl-oE cdcl-twl-stgyE cdcl-twl-cpE dest*!: *atm-N-U*)
subgoal by (*force elim*!: *cdcl-twl-oE cdcl-twl-stgyE cdcl-twl-cpE*)
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by (*auto elim*!: *cdcl-twl-oE*)
subgoal using *atm-N-U* **by** (*auto simp*: *cdcl-twl-o.simps decide*)
subgoal by *auto*
subgoal by (*auto elim*!: *cdcl-twl-oE*)
subgoal by *auto*
subgoal using *atm-N-U H* **by** *auto*
subgoal using *H atm-N-U* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *H atm-N-U* **by** *auto*
done
qed

declare *decide-or-skip-spec*[*THEN order-trans*, *refine-vcg*]

Skip and Resolve Loop

definition *skip-and-resolve-loop-inv* **where**
‹*skip-and-resolve-loop-inv* S_0 =
(λ(*brk*, *S*). *cdcl-twl-o*** S_0 *S* ∧ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧
clauses-to-update S = {#} ∧ *literals-to-update S* = {#} ∧
get-conflict S ≠ *None* ∧
count-decided (*get-trail S*) ≠ *0* ∧
get-trail S ≠ [] ∧
get-conflict S ≠ *Some* {#} ∧
(*brk* ⟶ *no-step cdcl*$_W$*-restart-mset.skip* (*state*$_W$*-of S*) ∧
no-step cdcl$_W$*-restart-mset.resolve* (*state*$_W$*-of S*)))›

definition *tl-state* :: ‹*'v twl-st* ⇒ *'v twl-st*› **where**
‹*tl-state* = (λ(*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*). (*tl M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*))›

definition *update-confl-tl* :: ‹*'v clause option* ⇒ *'v twl-st* ⇒ *'v twl-st*› **where**
‹*update-confl-tl* = (λ*D* (*M*, *N*, *U*, -, *NE*, *UE*, *WS*, *Q*). (*tl M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*))›

definition *skip-and-resolve-loop* :: ‹*'v twl-st* ⇒ *'v twl-st nres*› **where**
‹*skip-and-resolve-loop* S_0 =
do {
(-, *S*) ←

$WHILE_T{}^{skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\ S_0}$
(λ(*uip*, *S*). ¬*uip* ∧ ¬*is-decided* (*hd* (*get-trail S*)))
(λ(-, *S*).
do {
ASSERT(*get-trail S* ≠ []);
let D′ = *the* (*get-conflict S*);
(*L*, *C*) ← *SPEC*(λ(*L*, *C*). *Propagated L C* = *hd* (*get-trail S*));
if −*L* ∉# *D′ then*
do {*RETURN* (*False*, *tl-state S*)}
else
if get-maximum-level (*get-trail S*) (*remove1-mset* (−*L*) *D′*) = *count-decided* (*get-trail S*)
then
do {*RETURN* (*False*, *update-confl-tl* (*Some* ($cdcl_W$*-restart-mset.resolve-cls L D′ C*)) *S*)}
else
do {*RETURN* (*True*, *S*)}
}
)
(*False*, S_0);
RETURN S
}
›

lemma *skip-and-resolve-loop-spec*:
assumes *struct-S*: ‹*twl-struct-invs S*› **and** *stgy-S*: ‹*twl-stgy-invs S*› **and**
‹*clauses-to-update S* = {#}› **and** ‹*literals-to-update S* = {#}› **and**
‹*get-conflict S* ≠ *None*› **and** *count-dec*: ‹*count-decided* (*get-trail S*) > *0*›
shows ‹*skip-and-resolve-loop S* ≤ *SPEC*(λ*T*. *cdcl-twl-o*** *S T* ∧ *twl-struct-invs T* ∧ *twl-stgy-invs T*
∧
no-step $cdcl_W$*-restart-mset.skip* ($state_W$*-of T*) ∧
no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of T*) ∧
get-conflict T ≠ *None* ∧ *clauses-to-update T* = {#} ∧ *literals-to-update T* = {#})›
unfolding *skip-and-resolve-loop-def*
proof (*refine-vcg WHILEIT-rule*[**where** *R* = ‹*measure* (λ(*brk*, *S*). *Suc* (*length* (*get-trail S*) − *If brk 1 0*))›];
remove-dummy-vars)
show ‹*wf* (*measure* (λ(*brk*, *S*). *Suc* (*length* (*get-trail S*) − (*if brk then 1 else 0*))))›
by *auto*

have ‹*get-trail S* ⊨*as CNot* (*the* (*get-conflict S*))› **if** ‹*get-conflict S* ≠ *None*›
using *assms that* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def* **by** (*cases S*, *auto simp add*: $cdcl_W$*-restart-mset-state*)
then have ‹*get-trail S* ≠ []› **if** ‹*get-conflict S* ≠ *Some* {#}›
using *that assms* **by** *auto*
then show ‹*skip-and-resolve-loop-inv S* (*False*, *S*)›
using *assms* **by** (*cases S*) (*auto simp*: *skip-and-resolve-loop-inv-def* $cdcl_W$*-restart-mset.skip.simps*
$cdcl_W$*-restart-mset.resolve.simps* $cdcl_W$*-restart-mset-state*
twl-stgy-invs-def $cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)

fix *brk* :: *bool* **and** *T* :: ‹*′a twl-st*›
assume
inv: ‹*skip-and-resolve-loop-inv S* (*brk*, *T*)› **and**
brk: ‹*case* (*brk*, *T*) *of* (*brk*, *S*) ⇒ ¬ *brk* ∧ ¬ *is-decided* (*hd* (*get-trail S*))›
have [*simp*]: ‹*brk* = *False*›
using *brk* **by** *auto*
show *M-not-empty*: ‹*get-trail T* ≠ []›
using *brk inv* **unfolding** *skip-and-resolve-loop-inv-def* **by** *auto*

fix L :: ‹*'a literal*› **and** C
assume
LC: ‹*case* (*L*, *C*) *of* (*L*, *C*) ⇒ *Propagated L C* = *hd* (*get-trail T*)›

obtain *M N U D NE UE WS Q* **where**
T: ‹*T* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)›
by (*cases T*)

obtain *M'* :: ‹(*'a*, *'a clause*) *ann-lits*› **and** *D'* **where**
M: ‹*get-trail T* = *Propagated L C* # *M'*› **and** *WS*: ‹*WS* = {#}› **and** *Q*: ‹*Q* = {#}› **and** *D*: ‹*D* = *Some D'*› **and**
st: ‹*cdcl-twl-o*** *S T*› **and** *twl*: ‹*twl-struct-invs T*› **and** *D'*: ‹*D'* ≠ {#}› **and**
twl-stgy-S: ‹*twl-stgy-invs T*› **and**
[*simp*]: ‹*count-decided* (*tl M*) > *0*› ‹*count-decided* (*tl M*) ≠ *0*›
using *brk inv LC* **unfolding** *skip-and-resolve-loop-inv-def*
by (*cases* ‹*get-trail T*›; *cases* ‹*hd* (*get-trail T*)›) (*auto simp*: *T*)

{ — skip
assume *LD*: ‹− *L* ∉# *the* (*get-conflict T*)›
let *?T* = ‹*tl-state T*›
have *o-S-T*: ‹*cdcl-twl-o T ?T*›
using *cdcl-twl-o.skip*[*of L* ‹*the D*› *C M' N U NE UE*]
using *LD D inv M* **unfolding** *skip-and-resolve-loop-inv-def T WS Q D* **by** (*auto simp*: *tl-state-def*)
have *st-T*: ‹*cdcl-twl-o*** *S ?T*›
using *st o-S-T* **by** *auto*
moreover have *twl-T*: ‹*twl-struct-invs ?T*›
using *struct-S twl o-S-T cdcl-twl-o-twl-struct-invs* **by** *blast*
moreover have *twl-stgy-T*: ‹*twl-stgy-invs ?T*›
using *twl o-S-T stgy-S twl-stgy-S cdcl-twl-o-twl-stgy-invs* **by** *blast*
moreover have ‹*tl M* ≠ []›
using *twl-T D D'* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
by (*auto simp*: $cdcl_W$*-restart-mset-state T tl-state-def*)
ultimately show ‹*skip-and-resolve-loop-inv S* (*False*, *tl-state T*)›
using *WS Q D D'* **unfolding** *skip-and-resolve-loop-inv-def tl-state-def T*
by *simp*

show ‹((*False*, *?T*), (*brk*, *T*))
∈ *measure* (λ(*brk*, *S*). *Suc* (*length* (*get-trail S*) − (*if brk then 1 else 0*)))›
using *M-not-empty* **by** (*simp add*: *tl-state-def T M*)

}
{ — resolve
assume
LD: ‹¬− *L* ∉# *the* (*get-conflict T*)› **and**
max: ‹*get-maximum-level* (*get-trail T*) (*remove1-mset* (− *L*) (*the* (*get-conflict T*)))
= *count-decided* (*get-trail T*)›
let *?D* = ‹*remove1-mset* (− *L*) (*the* (*get-conflict T*)) ∪# *remove1-mset L C*›
let *?T* = ‹*update-confl-tl* (*Some ?D*) *T*›
have *count-dec*: ‹*count-decided M'* = *count-decided M*›
using *M* **unfolding** *T* **by** *auto*
then have *o-S-T*: ‹*cdcl-twl-o T ?T*›
using *cdcl-twl-o.resolve*[*of L* ‹*the D*› *C M' N U NE UE*] *LD D max M WS Q D*
by (*auto simp*: *T D update-confl-tl-def*)
then have *st-T*: ‹*cdcl-twl-o*** *S ?T*›

using *st* **by** *auto*
moreover have *twl-T*: ⟨*twl-struct-invs ?T*⟩
using *st-T twl o-S-T cdcl-twl-o-twl-struct-invs* **by** *blast*
moreover have *twl-stgy-T*: ⟨*twl-stgy-invs ?T*⟩
using *twl o-S-T twl-stgy-S cdcl-twl-o-twl-stgy-invs* **by** *blast*
moreover {
have ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting* ($state_W$*-of ?T*)⟩
using *twl-T D D′ M* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *fast*
then have ⟨*tl M* ⊨*as CNot ?D*⟩
using *M* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
by (*auto simp add*: $cdcl_W$*-restart-mset-state T update-confl-tl-def*)
}
moreover have ⟨*get-conflict ?T* ≠ *Some* {#}⟩
using *twl-stgy-T count-dec* **unfolding** *twl-stgy-invs-def update-confl-tl-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def T*
by (*auto simp*: *trail.simps conflicting.simps*)
ultimately show ⟨*skip-and-resolve-loop-inv S* (*False*, *?T*)⟩
using *WS Q D D′* **unfolding** *skip-and-resolve-loop-inv-def*
by (*auto simp add*: $cdcl_W$*-restart-mset.skip.simps* $cdcl_W$*-restart-mset.resolve.simps*
$cdcl_W$*-restart-mset-state update-confl-tl-def T*)

show ⟨((*False*, *?T*), (*brk*, *T*)) ∈ *measure* (λ(*brk*, *S*). *Suc* (*length* (*get-trail S*)
− (*if brk then 1 else 0*)))⟩
using *M-not-empty* **by** (*simp add*: *T update-confl-tl-def*)
}
{ — No step
assume
LD: ⟨¬− *L* ∉# *the* (*get-conflict T*)⟩ **and**
max: ⟨*get-maximum-level* (*get-trail T*) (*remove1-mset* (− *L*) (*the* (*get-conflict T*)))
≠ *count-decided* (*get-trail T*)⟩

show ⟨*skip-and-resolve-loop-inv S* (*True*, *T*)⟩
using *inv max LD D M* **unfolding** *skip-and-resolve-loop-inv-def*
by (*auto simp add*: $cdcl_W$*-restart-mset.skip.simps* $cdcl_W$*-restart-mset.resolve.simps*
$cdcl_W$*-restart-mset-state T*)
show ⟨((*True*, *T*), (*brk*, *T*)) ∈ *measure* (λ(*brk*, *S*). *Suc* (*length* (*get-trail S*) − (*if brk then 1 else 0*)))⟩
using *M-not-empty* **by** *simp*
}
next — Final properties
fix *brk T U*
assume
inv: ⟨*skip-and-resolve-loop-inv S* (*brk*, *T*)⟩ **and**
brk: ⟨¬(*case* (*brk*, *T*) *of* (*brk*, *S*) ⇒ ¬ *brk* ∧ ¬ *is-decided* (*hd* (*get-trail S*)))⟩
show ⟨$cdcl\text{-}twl\text{-}o^{**}$ *S T*⟩
using *inv* **by** (*auto simp add*: *skip-and-resolve-loop-inv-def*)

{ assume ⟨*is-decided* (*hd* (*get-trail T*))⟩
then have ⟨*no-step* $cdcl_W$*-restart-mset.skip* ($state_W$*-of T*)⟩ **and**
⟨*no-step* $cdcl_W$*-restart-mset.resolve* ($state_W$*-of T*)⟩
by (*cases T*; *auto simp add*: $cdcl_W$*-restart-mset.skip.simps*
$cdcl_W$*-restart-mset.resolve.simps* $cdcl_W$*-restart-mset-state*)+
}
moreover
{ assume ⟨*brk*⟩

then have ⟨*no-step* $cdcl_W$*-restart-mset.skip* (*state*$_W$*-of* T)⟩ **and**
⟨*no-step* $cdcl_W$*-restart-mset.resolve* (*state*$_W$*-of* T)⟩
using *inv* **by** (*auto simp*: *skip-and-resolve-loop-inv-def*)
}
ultimately show ⟨$\neg$ $cdcl_W$*-restart-mset.skip* (*state*$_W$*-of* T) U⟩ **and**
⟨$\neg$ $cdcl_W$*-restart-mset.resolve* (*state*$_W$*-of* T) U⟩
using *brk* **unfolding** *prod.case* **by** *blast+*

show ⟨*twl-struct-invs* T⟩
using *inv* **unfolding** *skip-and-resolve-loop-inv-def* **by** *auto*
show ⟨*twl-stgy-invs* T⟩
using *inv* **unfolding** *skip-and-resolve-loop-inv-def* **by** *auto*

show ⟨*get-conflict* $T \neq$ *None*⟩
using *inv* **by** (*auto simp*: *skip-and-resolve-loop-inv-def*)

show ⟨*clauses-to-update* $T = \{\#\}$⟩
using *inv* **by** (*auto simp*: *skip-and-resolve-loop-inv-def*)

show ⟨*literals-to-update* $T = \{\#\}$⟩
using *inv* **by** (*auto simp*: *skip-and-resolve-loop-inv-def*)
qed

declare *skip-and-resolve-loop-spec*[*THEN order-trans*, *refine-vcg*]

Backtrack

definition *extract-shorter-conflict* :: ⟨$'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st nres*⟩ **where**
⟨*extract-shorter-conflict* = (λ(M, N, U, D, NE, UE, WS, Q).
$SPEC(\lambda S'. \exists D'. S' = (M, N, U, Some\ D', NE, UE, WS, Q) \wedge$
$D' \subseteq\#\ the\ D \wedge clause\ `\#\ (N + U) + NE + UE \models pm\ D' \wedge -lit\text{-}of\ (hd\ M) \in\#\ D'))$⟩

fun *equality-except-conflict* :: ⟨$'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st* $\Rightarrow$ *bool*⟩ **where**
⟨*equality-except-conflict* (M, N, U, D, NE, UE, WS, Q) (M', N', U', D', NE', UE', WS', Q') $\longleftrightarrow$
$M = M' \wedge N = N' \wedge U = U' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$⟩

lemma *extract-shorter-conflict-alt-def*:
⟨*extract-shorter-conflict* S =
$SPEC(\lambda S'. \exists D'.$ *equality-except-conflict* $S\ S' \wedge Some\ D' =$ *get-conflict* $S' \wedge$
$D' \subseteq\#\ the$ (*get-conflict* S) $\wedge$ *clause* `# (*get-clauses* S) + *unit-clss* $S \models pm\ D' \wedge$
$-lit\text{-}of$ (*hd* (*get-trail* S)) $\in\#\ D'$)⟩
unfolding *extract-shorter-conflict-def*
by (*cases* S) (*auto simp*: *ac-simps*)

definition *reduce-trail-bt* :: ⟨$'v$ *literal* $\Rightarrow$ $'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st nres*⟩ **where**
⟨*reduce-trail-bt* = (λL (M, N, U, D', NE, UE, WS, Q). *do* {
$M1 \leftarrow SPEC(\lambda M1. \exists K\ M2.$ (*Decided* K # $M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* M) $\wedge$
get-level M K = *get-maximum-level* M (*the* $D' - \{\#-L\#\}$) + *1*);
RETURN ($M1$, N, U, D', NE, UE, WS, Q)
})⟩

definition *propagate-bt* :: ⟨$'v$ *literal* $\Rightarrow$ $'v$ *literal* $\Rightarrow$ $'v$ *twl-st* $\Rightarrow$ $'v$ *twl-st*⟩ **where**
⟨*propagate-bt* = ($\lambda L\ L'$ (M, N, U, D, NE, UE, WS, Q).
(*Propagated* ($-L$) (*the* D) # M, N, *add-mset* (*TWL-Clause* $\{\#-L, L'\#\}$ (*the* $D - \{\#-L, L'\#\}$))
U, *None*,
NE, UE, WS, $\{\#L\#\}$))⟩

definition *propagate-unit-bt* :: ⟨$'v$ *literal* ⇒ $'v$ *twl-st* ⇒ $'v$ *twl-st*⟩ **where**
⟨*propagate-unit-bt* = (λL (M, N, U, D, NE, UE, WS, Q).
($Propagated$ ($-L$) (the D) # M, N, U, $None$, NE, $add\text{-}mset$ (the D) UE, WS, {#L#}))⟩

definition *backtrack-inv* **where**
⟨*backtrack-inv* $S \longleftrightarrow$ *get-trail* $S \neq []$ $\wedge$ *get-conflict* $S \neq$ *Some* {#}⟩

definition *backtrack* :: ⟨$'v$ *twl-st* ⇒ $'v$ *twl-st nres*⟩ **where**
⟨*backtrack* S =
do {
ASSERT(*backtrack-inv* S);
let L = *lit-of* (*hd* (*get-trail* S));
$S \leftarrow$ *extract-shorter-conflict* S;
$S \leftarrow$ *reduce-trail-bt* L S;

if size (*the* (*get-conflict* S)) > 1
then do {
$L' \leftarrow$ *SPEC*($\lambda L'$. $L' \in\#$ *the* (*get-conflict* S) $-$ {#$-L$#} $\wedge$ $L \neq -L'$ $\wedge$
get-level (*get-trail* S) L' = *get-maximum-level* (*get-trail* S) (*the* (*get-conflict* S) $-$ {#$-L$#}));
RETURN (*propagate-bt* L L' S)
}
else do {
RETURN (*propagate-unit-bt* L S)
}
}
⟩

lemma
assumes *confl*: ⟨*get-conflict* $S \neq$ *None*⟩ ⟨*get-conflict* $S \neq$ *Some* {#}⟩ **and**
w-q: ⟨*clauses-to-update* S = {#}⟩ **and** *p*: ⟨*literals-to-update* S = {#}⟩ **and**
ns-s: ⟨*no-step* $cdcl_W$*-restart-mset.skip* ($state_W$*-of* S)⟩ **and**
ns-r: ⟨*no-step* $cdcl_W$*-restart-mset.resolve* ($state_W$*-of* S)⟩ **and**
twl-struct: ⟨*twl-struct-invs* S⟩ **and** *twl-stgy*: ⟨*twl-stgy-invs* S⟩
shows
backtrack-spec:
⟨*backtrack* $S \leq$ *SPEC* (λT. *cdcl-twl-o* S T $\wedge$ *get-conflict* T = *None* $\wedge$ *no-step cdcl-twl-o* T $\wedge$
twl-struct-invs T $\wedge$ *twl-stgy-invs* T $\wedge$ *clauses-to-update* T = {#} $\wedge$
literals-to-update $T \neq$ {#})⟩ (**is** *?spec*) **and**
backtrack-nofail:
⟨*nofail* (*backtrack* S)⟩ (**is** *?fail*)
proof −
let *?S* = ⟨$state_W$*-of* S⟩
have *inv-s*: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant ?S*⟩ **and**
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv ?S*⟩
using *twl-struct twl-stgy* **unfolding** *twl-struct-invs-def twl-stgy-invs-def* **by** *fast+*
let *?D'* = ⟨*the* (*conflicting ?S*)⟩
have *M-CNot-D'*: ⟨*trail ?S* $\models$*as CNot ?D'*⟩
using *inv confl* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
by (*cases* ⟨*conflicting ?S*⟩; *cases* S) (*auto simp*: $cdcl_W$*-restart-mset-state*)
then have *trail*: ⟨*get-trail* $S \neq []$⟩
using *confl* **unfolding** *true-annots-true-cls-def-iff-negation-in-model*
by (*cases* S) (*auto simp*: $cdcl_W$*-restart-mset-state*)
show *?spec*

unfolding *backtrack-def extract-shorter-conflict-def reduce-trail-bt-def*
proof (*refine-vcg*; *remove-dummy-vars*; *clarify?*)
show ‹*backtrack-inv S*›
using *trail confl* **unfolding** *backtrack-inv-def* **by** *fast*

fix *M M1 M2* :: ‹(*'a*, *'a clause*) *ann-lits*› **and**
N U :: ‹*'a twl-clss*› **and**
D :: ‹*'a clause option*› **and** *D'* :: ‹*'a clause*› **and** *NE UE* :: ‹*'a clauses*› **and**
WS :: ‹*'a clauses-to-update*› **and** *Q* :: ‹*'a lit-queue*› **and** *K K'* :: ‹*'a literal*›
let *?S* = ‹(*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)›
let *?T* = ‹(*M*, *N*, *U*, *Some D'*, *NE*, *UE*, *WS*, *Q*)›
let *?U* = ‹(*M1*, *N*, *U*, *Some D'*, *NE*, *UE*, *WS*, *Q*)›
let *?MS* = ‹*get-trail ?S*›
let *?MT* = ‹*get-trail ?T*›
assume
S: ‹*S* = (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)› **and**
D'-D: ‹*D'* ⊆# *the D*› **and**
L-D': ‹−*lit-of* (*hd M*) ∈# *D'*› **and**
N-U-NE-UE-D': ‹*clause* '# (*N* + *U*) + *NE* + *UE* ⊨*pm D'*› **and**
decomp: ‹(*Decided K'* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
lev-K': ‹*get-level M K'* = *get-maximum-level M* (*remove1-mset* (− *lit-of* (*hd ?MS*))
(*the* (*Some D'*))) + *1*›
have *WS*: ‹*WS* = {#}› **and** *Q*: ‹*Q* = {#}›
using *w-q p* **unfolding** *S* **by** *auto*

have *uL-D*: ‹− *lit-of* (*hd M*) ∈# *the D*›
using *decomp N-U-NE-UE-D' D'-D L-D' lev-K'*
unfolding *WS Q*
by *auto*

have *D-Some-the*: ‹*D* = *Some* (*the D*)›
using *confl S* **by** *auto*
let *?S'* = ‹$state_W$*-of S*›
have *inv-s*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant ?S'*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv ?S'*›
using *twl-struct twl-stgy* **unfolding** *twl-struct-invs-def twl-stgy-invs-def* **by** *fast+*
have *Q*: ‹*Q* = {#}› **and** *WS*: ‹*WS* = {#}›
using *w-q p* **unfolding** *S* **by** *auto*
have *M-CNot-D'*: ‹*M* ⊨*as CNot D'*›
using *M-CNot-D' S D'-D*
by (*auto simp*: $cdcl_W$*-restart-mset-state true-annots-true-cls-def-iff-negation-in-model*)
obtain *L'' M'* **where** *M*: ‹*M* = *L''* # *M'*›
using *trail S* **by** (*cases M*) *auto*
have *D'-empty*: ‹*D'* ≠ {#}›
using *L-D'* **by** *auto*
have *L'-D*: ‹−*lit-of L''* ∈# *D'*›
using *L-D'* **by** (*auto simp*: $cdcl_W$*-restart-mset-state M*)
have *lev-inv*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv ?S'*›
using *inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* **by** *fast*
then have *n-d*: ‹*no-dup M*› **and** *dec*: ‹*backtrack-lvl ?S'* = *count-decided M*›
using *S* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: $cdcl_W$*-restart-mset-state*)
then have *uL''-M*: ‹−*lit-of L''* ∉ *lits-of-l M*›
by (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l M*)
have ‹*get-maximum-level M* (*remove1-mset* (−*lit-of* (*hd M*)) *D'*) < *count-decided M*›
proof (*cases L''*)

```
    case (Decided x1) note L'' = this(1)
    have ‹distinct-mset (the D)›
      using inv S confl unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
        cdcl_W-restart-mset.distinct-cdcl_W-state-def
      by (auto simp: cdcl_W-restart-mset-state)
    then have ‹distinct-mset D'›
      using D'-D by (blast intro: distinct-mset-mono)
    then have ‹− x1 ∉# remove1-mset (− x1) D'›
      using L'-D L'' D'-D by (auto dest: distinct-mem-diff-mset)
    then have H: ‹∀x∈#remove1-mset (− lit-of (hd M)) D'. undefined-lit [L''] x›
      using L'' M-CNot-D' uL''-M
      by (fastforce simp: atms-of-def atm-of-eq-atm-of M true-annots-true-cls-def-iff-negation-in-model
          dest: in-diffD)
    have ‹get-maximum-level M (remove1-mset (− lit-of (hd M)) D') =
      get-maximum-level M' (remove1-mset (− lit-of (hd M)) D')›
      using get-maximum-level-skip-beginning[OF H, of M'] M
      by auto
    then show ?thesis
      using count-decided-ge-get-maximum-level[of M' ‹remove1-mset (−lit-of (hd M)) D'›] M L''
      by simp
  next
    case (Propagated L C) note L'' = this(1)
    moreover {
      have ‹∀L mark a b. a @ Propagated L mark # b = trail (state_W-of S) ⟶
        b ⊨as CNot (remove1-mset L mark) ∧ L ∈# mark›
        using inv unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
          cdcl_W-restart-mset.cdcl_W-conflicting-def
        by blast
      then have ‹L ∈# C›
        by (force simp: S M cdcl_W-restart-mset-state L'') }
    moreover have D-empty: ‹the D ≠ {#}›
      using D'-D D'-empty by auto
    moreover have ‹−L ∈# the D›
      using ns-s L'' confl D-empty
      by (force simp: cdcl_W-restart-mset.skip.simps S M cdcl_W-restart-mset-state)
    ultimately have ‹get-maximum-level M (remove1-mset (− lit-of (hd M)) (the D))
        < count-decided M›
      using ns-r confl count-decided-ge-get-maximum-level[of M
‹remove1-mset (−lit-of (hd M)) (the D)›]
      by (fastforce simp add: cdcl_W-restart-mset.resolve.simps S M
          cdcl_W-restart-mset-state)

    moreover have ‹get-maximum-level M (remove1-mset (− lit-of (hd M)) D') ≤
          get-maximum-level M (remove1-mset (− lit-of (hd M)) (the D))›
      by (rule get-maximum-level-mono) (use D'-D in ‹auto intro: mset-le-subtract›)
    ultimately show ?thesis
      by simp
  qed

  then have ‹∃K M1 M2. (Decided K # M1, M2) ∈ set (get-all-ann-decomposition M) ∧
    get-level M K = get-maximum-level M (remove1-mset (−lit-of (hd M)) D') + 1›
    using cdcl_W-restart-mset.backtrack-ex-decomp n-d
    by (auto simp: cdcl_W-restart-mset-state S)

  define i where ‹i = get-maximum-level M (remove1-mset (− lit-of (hd M)) D')›
```

let *?T* = ‹(*Propagated* (−*lit-of* (*hd M*)) *D'* # *M1*, *N*,
add-mset (*TWL-Clause* {#−*lit-of* (*hd M*), *K*#} (*D'* − {#−*lit-of* (*hd M*), *K*#})) *U*,
None, *NE*, *UE*, *WS*, {#*lit-of* (*hd M*)#})›
let *?T'* = ‹(*Propagated* (−*lit-of* (*hd M*)) *D'* # *M1*, *N*,
add-mset (*TWL-Clause* {#−*lit-of* (*hd M*), *K*#} (*D'* − {#−*lit-of* (*hd M*), *K*#})) *U*,
None, *NE*, *UE*, *WS*, {#− (−*lit-of* (*hd M*))#})›

have *lev-D'*: ‹*count-decided M* = *get-maximum-level* (*L''* # *M'*) *D'*›
using *count-decided-ge-get-maximum-level*[*of M D'*] *L'-D*
get-maximum-level-ge-get-level[*of* ‹−*lit-of L''*› *D' M*] **unfolding** *M*
by (*auto split*: *if-splits*)

{ — conflict clause > 1 literal
assume *size-D*: ‹*1* < *size* (*the* (*get-conflict ?U*))› **and**
K-D: ‹*K* ∈# *remove1-mset* (− *lit-of* (*hd ?MS*)) (*the* (*get-conflict ?U*))› **and**
lev-K: ‹*get-level* (*get-trail ?U*) *K* = *get-maximum-level* (*get-trail ?U*)
(*remove1-mset* (− *lit-of* (*hd* (*get-trail ?S*))) (*the* (*get-conflict ?U*)))›

have ‹∀ *L'* ∈# *D'*. −*L'* ∈ *lits-of-l M*›
using *M-CNot-D' uL''-M*
by (*fastforce simp*: *atms-of-def atm-of-eq-atm-of M true-annots-true-cls-def-iff-negation-in-model*
dest: *in-diffD*)
obtain *c* **where** *c*: ‹*M* = *c* @ *M2* @ *Decided K'* # *M1*›
using *get-all-ann-decomposition-exists-prepend*[*OF decomp*] **by** *blast*
have ‹*get-level M K'* = *Suc* (*count-decided M1*)›
using *n-d* **unfolding** *c* **by** *auto*
then have *i*: ‹*i* = *count-decided M1*›
using *lev-K'* **unfolding** *i-def* **by** *auto*
have *lev-M-M1*: ‹∀ *L'* ∈# *D'* − {#−*lit-of* (*hd M*)#}. *get-level M L'* = *get-level M1 L'*›
proof
fix *L'*
assume *L'*: ‹*L'* ∈# *D'* − {#−*lit-of* (*hd M*)#}›
have ‹*get-level M L'* > *count-decided M1*› **if** ‹*defined-lit* (*c* @ *M2* @ *Decided K'* # []) *L'*›
using *get-level-skip-end*[*OF that*, *of M1*] *n-d that get-level-last-decided-ge*[*of* ‹*c* @ *M2*›]
by (*auto simp*: *c*)
moreover have ‹*get-level M L'* ≤ *i*›
using *get-maximum-level-ge-get-level*[*OF L'*, *of M*] **unfolding** *i-def* **by** *auto*
ultimately show ‹*get-level M L'* = *get-level M1 L'*›
using *n-d c L' i* **by** (*cases* ‹*defined-lit* (*c* @ *M2* @ *Decided K'* # []) *L'*›) *auto*
qed
have ‹*get-level M1* '# *remove1-mset* (− *lit-of* (*hd M*)) *D'* = *get-level M* '# *remove1-mset* (− *lit-of* (*hd M*)) *D'*›
by (*rule image-mset-cong*) (*use lev-M-M1* **in** *auto*)
then have *max-M1-M1-D*: ‹*get-maximum-level M1* (*remove1-mset* (− *lit-of* (*hd M*)) *D'*) =
get-maximum-level M (*remove1-mset* (− *lit-of* (*hd M*)) *D'*)›
unfolding *get-maximum-level-def* **by** *argo*

have ‹∃ *L'* ∈# *remove1-mset* (−*lit-of* (*hd M*)) *D'*.
get-level M L' = *get-maximum-level M* (*remove1-mset* (− *lit-of* (*hd M*)) *D'*)›
by (*rule get-maximum-level-exists-lit-of-max-level*)
(*use size-D* **in** ‹*auto simp*: *remove1-mset-empty-iff*›)
have *D'-ne-single*: ‹*D'* ≠ {#− *lit-of* (*hd M*)#}›
using *size-D* **apply** (*cases D'*, *simp*)
apply (*rename-tac L D''*)
apply (*case-tac D''*)

by *simp-all*
have ‹*cdcl-twl-o* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) *?T'*›
unfolding *Q WS option.sel list.sel*
apply (*subst D-Some-the*)
apply (*rule cdcl-twl-o.backtrack-nonunit-clause*[*of* ‹−*lit-of* (*hd M*)› - *K' M1 M2* - - *i*])
subgoal using *D'-D L-D'* **by** *blast*
subgoal using *L'-D decomp M* **by** *auto*
subgoal using *L'-D decomp M* **by** *auto*
subgoal using *L'-D M lev-D'* **by** *auto*
subgoal using *i lev-D' i-def* **by** *auto*
subgoal using *lev-K' i-def* **by** *auto*
subgoal using *D'-ne-single* **.**
subgoal using *D'-D* **.**
subgoal using *N-U-NE-UE-D'* **.**
subgoal using *L-D'* **.**
subgoal using *K-D* **by** (*auto dest*: *in-diffD*)
subgoal using *lev-K lev-M-M1 K-D* **by** (*simp add*: *i-def max-M1-M1-D*)
done
then show *cdcl*: ‹*cdcl-twl-o ?S* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*)›
unfolding *WS Q* **by** (*auto simp*: *propagate-bt-def*)

show ‹*get-conflict* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) = *None*›
by (*auto simp*: *propagate-bt-def*)

show ‹*twl-struct-invs* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*)›
using *S cdcl cdcl-twl-o-twl-struct-invs twl-struct* **by** (*auto simp*: *propagate-bt-def*)
show ‹*twl-stgy-invs* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*)›
using *S cdcl cdcl-twl-o-twl-stgy-invs twl-struct twl-stgy* **by** *blast*
show ‹*clauses-to-update* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) = {#}›
using *WS* **by** (*auto simp*: *propagate-bt-def*)

show *False* **if** ‹*cdcl-twl-o* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) (*an*, *ao*, *ap*, *aq*, *ar*, *as*, *at*, *b*)›
for *an ao ap aq ar as at b*
using *that* **by** (*auto simp*: *cdcl-twl-o.simps propagate-bt-def*)

show *False* **if** ‹*literals-to-update* (*propagate-bt* (*lit-of* (*hd* (*get-trail ?S*))) *K ?U*) = {#}›
using *that* **by** (*auto simp*: *propagate-bt-def*)

}

{ — conflict clause has 1 literal
assume ‹¬ *1* < *size* (*the* (*get-conflict ?U*))›
then have *D'*: ‹*D'* = {#−*lit-of* (*hd M*)#}›
using *L'-D* **by** (*cases D'*) (*auto simp*: *M*)
let *?T* = ‹(*Propagated* (− *lit-of* (*hd M*)) *D'* # *M1*, *N*, *U*, *None*, *NE*, *add-mset D' UE*, *WS*, *unmark* (*hd M*))›
let *?T'* = ‹(*Propagated* (− *lit-of* (*hd M*)) *D'* # *M1*, *N*, *U*, *None*, *NE*, *add-mset D' UE*, *WS*, {#− (−*lit-of* (*hd M*))#})›

have *i-0*: ‹*i* = *0*›
using *i-def* **by** (*auto simp*: *D'*)

have ‹*cdcl-twl-o* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) *?T'*›
unfolding *D' option.sel WS Q* **apply** (*subst D-Some-the*)
apply (*rule cdcl-twl-o.backtrack-unit-clause*[*of* - ‹*the D*› *K' M1 M2* - *D' i*])

subgoal using *D′-D D′* **by** *auto*
subgoal using *decomp* **by** *simp*
subgoal by (*simp add*: *M*)
subgoal using *D′* **by** (*auto simp*: *get-maximum-level-add-mset*)
subgoal using *i-def* **by** *simp*
subgoal using *lev-K′ i-def*[*symmetric*] **by** *auto*
subgoal using *D′* **.**
subgoal using *D′-D* **.**
subgoal using *N-U-NE-UE-D′* **.**
done
then show *cdcl*: ‹*cdcl-twl-o* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)
(*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*)›
by (*auto simp add*: *propagate-unit-bt-def*)
show ‹*get-conflict* (*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*) = *None*›
by (*auto simp add*: *propagate-unit-bt-def*)

show ‹*twl-struct-invs* (*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*)›
using *S cdcl cdcl-twl-o-twl-struct-invs twl-struct* **by** *blast*

show ‹*twl-stgy-invs* (*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*)›
using *S cdcl cdcl-twl-o-twl-stgy-invs twl-struct twl-stgy* **by** *blast*
show ‹*clauses-to-update* (*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*) = {#}›
using *WS* **by** (*auto simp add*: *propagate-unit-bt-def*)
show *False* **if** ‹*literals-to-update* (*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*) = {#}›
using *that* **by** (*auto simp add*: *propagate-unit-bt-def*)
fix *an ao ap aq ar as at b*
show *False* **if** ‹*cdcl-twl-o* (*propagate-unit-bt* (*lit-of* (*hd* (*get-trail ?S*))) *?U*) (*an*, *ao*, *ap*, *aq*, *ar*, *as*, *at*, *b*) ›
using *that* **by** (*auto simp*: *cdcl-twl-o.simps propagate-unit-bt-def*)
}
qed
then show *?fail*
using *nofail-simps*(*2*) *pwD1* **by** *blast*
qed

declare *backtrack-spec*[*THEN order-trans*, *refine-vcg*]

Full loop

definition *cdcl-twl-o-prog* :: ‹*′v twl-st* ⇒ (*bool* × *′v twl-st*) *nres*› **where**
‹*cdcl-twl-o-prog S* =
do {
if get-conflict S = *None*
then decide-or-skip S
else do {
if count-decided (*get-trail S*) > *0*
then do {
T ← *skip-and-resolve-loop S*;
ASSERT(*get-conflict T* ≠ *None* ∧ *get-conflict T* ≠ *Some* {#});
U ← *backtrack T*;
RETURN (*False*, *U*)
}
else
RETURN (*True*, *S*)
}
}

⟩

setup ⟨*map-theory-claset* (*fn ctxt* => *ctxt delSWrapper* (*split-all-tac*))⟩
declare *split-paired-All*[*simp del*]

lemma *skip-and-resolve-same-decision-level*:
assumes ⟨*cdcl-twl-o S T*⟩ ⟨*get-conflict T* ≠ *None*⟩
shows ⟨*count-decided* (*get-trail T*) = *count-decided* (*get-trail S*)⟩
using *assms* **by** (*induction rule*: *cdcl-twl-o.induct*) *auto*

lemma *skip-and-resolve-conflict-before*:
assumes ⟨*cdcl-twl-o S T*⟩ ⟨*get-conflict T* ≠ *None*⟩
shows ⟨*get-conflict S* ≠ *None*⟩
using *assms* **by** (*induction rule*: *cdcl-twl-o.induct*) *auto*

lemma *rtranclp-skip-and-resolve-same-decision-level*:
⟨$cdcl\text{-}twl\text{-}o^{**}$ *S T* ⟹ *get-conflict S* ≠ *None* ⟹ *get-conflict T* ≠ *None* ⟹
count-decided (*get-trail T*) = *count-decided* (*get-trail S*)⟩
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *skip-and-resolve-conflict-before*[*of T U*]
by (*auto simp*: *skip-and-resolve-same-decision-level*)
done

lemma *empty-conflict-lvl0*:
⟨*twl-stgy-invs T* ⟹ *get-conflict T* = *Some* {#} ⟹ *count-decided* (*get-trail T*) = *0*⟩
by (*cases T*) (*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def* *trail.simps conflicting.simps*)

abbreviation *cdcl-twl-o-prog-spec* **where**
⟨*cdcl-twl-o-prog-spec S* ≡ λ(*brk*, *T*).
$cdcl\text{-}twl\text{-}o^{**}$ *S T* ∧
(*get-conflict T* ≠ *None* ⟶ *count-decided* (*get-trail T*) = *0*) ∧
(¬ *brk* ⟶ *get-conflict T* = *None* ∧ (∀ *S'*. ¬ *cdcl-twl-o T S'*)) ∧
(*brk* ⟶ *get-conflict T* ≠ *None* ∨ (∀ *S'*. ¬ *cdcl-twl-stgy T S'*)) ∧
twl-struct-invs T ∧ *twl-stgy-invs T* ∧ *clauses-to-update T* = {#} ∧
(¬ *brk* ⟶ *literals-to-update T* ≠ {#}) ∧
(¬*brk* ⟶ ¬ (∀ *S'*. ¬ *cdcl-twl-o S S'*) ⟶ $cdcl\text{-}twl\text{-}o^{++}$ *S T*)⟩

lemma *cdcl-twl-o-prog-spec*:
assumes ⟨*twl-struct-invs S*⟩ **and** ⟨*twl-stgy-invs S*⟩ **and** ⟨*clauses-to-update S* = {#}⟩ **and**
⟨*literals-to-update S* = {#}⟩ **and**
ns-cp: ⟨*no-step cdcl-twl-cp S*⟩
shows
⟨*cdcl-twl-o-prog S* ≤ *SPEC*(*cdcl-twl-o-prog-spec S*)⟩
(**is** ⟨- ≤ *?S*⟩)
proof −
have [*iff*]: ⟨¬ *cdcl-twl-cp S T*⟩ **for** *T*
using *ns-cp* **by** *fast*

show *?thesis*
unfolding *cdcl-twl-o-prog-def*
apply (*refine-vcg decide-or-skip-spec*[*THEN order-trans*]; *remove-dummy-vars*)
— initial invariants

subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal by *simp*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal for *T* **using** *assms empty-conflict-lvl0*[*of T*]
rtranclp-skip-and-resolve-same-decision-level[*of S T*] **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** (*auto elim*!: *cdcl-twl-oE simp*: *image-Un*)
subgoal by (*auto elim*!: *cdcl-twl-stgyE cdcl-twl-oE cdcl-twl-cpE*)
subgoal by (*auto simp*: *rtranclp-unfold elim*!: *cdcl-twl-oE*)
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal for *uip* **by** *auto*
done
qed

declare *cdcl-twl-o-prog-spec*[*THEN order-trans*, *refine-vcg*]

1.2.3 Full Strategy

abbreviation *cdcl-twl-stgy-prog-inv* **where**
‹*cdcl-twl-stgy-prog-inv* S_0 ≡ λ(*brk*, *T*). *twl-struct-invs T* ∧ *twl-stgy-invs T* ∧
(*brk* ⟶ *final-twl-state T*) ∧ *cdcl-twl-stgy*** S_0 *T* ∧ *clauses-to-update T* = {#} ∧
(¬*brk* ⟶ *get-conflict T* = *None*)›

definition *cdcl-twl-stgy-prog* :: ‹'*v twl-st* ⇒ '*v twl-st nres*› **where**
‹*cdcl-twl-stgy-prog* S_0 =
do {
do {
(*brk*, *T*) ← $WHILE_T$$^{cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}inv\ S_0}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*).
do {
T ← *unit-propagation-outer-loop S*;
cdcl-twl-o-prog T
})
(*False*, S_0);
RETURN T
}

}
›

lemma *wf-cdcl-twl-stgy-measure*:
‹*wf* ({((*brkT*, *T*), (*brkS*, *S*)). *twl-struct-invs* $S \wedge$ *cdcl-twl-stgy*$^{++}$ S T}
∪ {((*brkT*, *T*), (*brkS*, *S*)). $S = T \wedge brkT \wedge \neg brkS$})›
(**is** ‹*wf* (*?TWL* ∪ *?BOOL*)›)
proof (*rule wf-union-compatible*)
show ‹*wf ?TWL*›
using *tranclp-wf-cdcl-twl-stgy wf-snd-wf-pair* **by** *blast*
show ‹*?TWL O ?BOOL* ⊆ *?TWL*›
by *auto*

show ‹*wf ?BOOL*›
unfolding *wf-iff-no-infinite-down-chain*
proof *clarify*
fix *f* :: ‹*nat* ⇒ *bool* × *'b*›
assume *H*: ‹∀ *i*. (*f* (*Suc i*), *f i*) ∈ {((*brkT*, *T*), *brkS*, *S*). $S = T \wedge brkT \wedge \neg\ brkS$}›
then have ‹(*f* (*Suc 0*), *f 0*) ∈ {((*brkT*, *T*), *brkS*, *S*). $S = T \wedge brkT \wedge \neg\ brkS$}› **and**
‹(*f* (*Suc 1*), *f 1*) ∈ {((*brkT*, *T*), *brkS*, *S*). $S = T \wedge brkT \wedge \neg\ brkS$}›
by *presburger+*
then show *False*
by *auto*
qed
qed

lemma *cdcl-twl-o-final-twl-state*:
assumes
‹*cdcl-twl-stgy-prog-inv S* (*brk*, *T*)› **and**
‹*case* (*brk*, *T*) *of* (*brk*, -) ⇒ ¬ *brk*› **and**
twl-o: ‹*cdcl-twl-o-prog-spec U* (*True*, *V*)›
shows ‹*final-twl-state V*›
proof –
have ‹*cdcl-twl-o*** *U V*› **and**
confl-lev: ‹*get-conflict V* ≠ *None* ⟶ *count-decided* (*get-trail V*) = *0*› **and**
final: ‹*get-conflict V* ≠ *None* ∨ (∀ *S'*. ¬ *cdcl-twl-stgy V S'*)›
‹*twl-struct-invs V*›
‹*twl-stgy-invs V*›
‹*clauses-to-update V* = {#}›
using *twl-o*
by *force+*

show *?thesis*
unfolding *final-twl-state-def*
using *confl-lev final*
by *auto*
qed

lemma *cdcl-twl-stgy-in-measure*:
assumes
twl-stgy: ‹*cdcl-twl-stgy-prog-inv S* (*brk0*, *T*)› **and**
brk0: ‹*case* (*brk0*, *T*) *of* (*brk*, *uu-*) ⇒ ¬ *brk*› **and**
twl-o: ‹*cdcl-twl-o-prog-spec U V*› **and**
[*simp*]: ‹*twl-struct-invs U*› **and**
TU: ‹*cdcl-twl-cp*** *T U*› **and**
‹*literals-to-update U* = {#}›

shows ‹(V, *brk0*, T)
$\in \{((brkT,\ T),\ brkS,\ S).\ twl\text{-}struct\text{-}invs\ S \land cdcl\text{-}twl\text{-}stgy^{++}\ S\ T\} \cup$
$\{((brkT,\ T),\ brkS,\ S).\ S = T \land brkT \land \neg\ brkS\}$›

proof –
have [*simp*]: ‹*twl-struct-invs* T›
using *twl-stgy* **by** *fast+*
obtain brk' V' **where**
V: ‹$V = (brk',\ V')$›
by (*cases* V)
have
UV: ‹$cdcl\text{-}twl\text{-}o^{**}\ U\ V'$› **and**
‹($get\text{-}conflict\ V' \neq None \longrightarrow count\text{-}decided\ (get\text{-}trail\ V') = 0$)› **and**
not-brk': ‹($\neg\ brk' \longrightarrow get\text{-}conflict\ V' = None \land (\forall S'.\ \neg\ cdcl\text{-}twl\text{-}o\ V'\ S')$)› **and**
brk': ‹($brk' \longrightarrow get\text{-}conflict\ V' \neq None \lor (\forall S'.\ \neg\ cdcl\text{-}twl\text{-}stgy\ V'\ S')$)› **and**
[*simp*]: ‹*twl-struct-invs* V'›
‹*twl-stgy-invs* V'›
‹$clauses\text{-}to\text{-}update\ V' = \{\#\}$› **and**
no-lits-to-upd: ‹($0 < count\text{-}decided\ (get\text{-}trail\ V') \longrightarrow \neg\ brk' \longrightarrow literals\text{-}to\text{-}update\ V' \neq \{\#\}$)›
‹($\neg brk' \longrightarrow \neg\ (\forall S'.\ \neg\ cdcl\text{-}twl\text{-}o\ U\ S') \longrightarrow cdcl\text{-}twl\text{-}o^{++}\ U\ V'$)›
using *twl-o* **unfolding** V
by *fast+*
have ‹$cdcl\text{-}twl\text{-}stgy^{**}\ T\ V'$›
using *TU UV* **by** (*auto dest*!: *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD*)
then have *TV-or-tranclp-TV*: ‹$T = V' \lor cdcl\text{-}twl\text{-}stgy^{++}\ T\ V'$›
unfolding *rtranclp-unfold* **by** *auto*
have [*simp*]: ‹$\neg\ cdcl\text{-}twl\text{-}stgy^{++}\ V'\ V'$›
using *wf-not-refl*[*OF tranclp-wf-cdcl-twl-stgy*, *of* V'] **by** *auto*
have [*simp*]: ‹$brk0 = False$›
using *brk0* **by** *auto*

have ‹brk'› **if** ‹$T = V'$›
proof –
have *ns-TV*: ‹$\neg cdcl\text{-}twl\text{-}stgy^{++}\ T\ V'$›
using *that*[*symmetric*] *wf-not-refl*[*OF tranclp-wf-cdcl-twl-stgy*, *of* T] **by** *auto*

have *ns-T-T*: ‹$\neg cdcl\text{-}twl\text{-}o^{++}\ T\ T$›
using *wf-not-refl*[*OF tranclp-wf-cdcl-twl-o*, *of* T] **by** *auto*
have ‹$T = U$›
by (*metis* (*no-types*, *hide-lams*) *TU UV ns-TV rtranclp-cdcl-twl-cp-stgyD*
rtranclp-cdcl-twl-o-stgyD rtranclp-tranclp-tranclp rtranclp-unfold)
show *?thesis*
using *assms* ‹$literals\text{-}to\text{-}update\ U = \{\#\}$› **unfolding** V *that*[*symmetric*] ‹$T = U$›[*symmetric*]
by (*auto simp*: *ns-T-T*)
qed

then show *?thesis*
using *TV-or-tranclp-TV*
unfolding V
by *auto*
qed

lemma *cdcl-twl-o-prog-cdcl-twl-stgy*:
assumes
twl-stgy: ‹*cdcl-twl-stgy-prog-inv* S (brk, S')› **and**
‹*case* (brk, S') *of* (brk, *uu-*) $\Rightarrow \neg$ *brk*› **and**
twl-o: ‹*cdcl-twl-o-prog-spec* T (brk', U)› **and**

⟨*twl-struct-invs T*⟩ **and**
cp: ⟨*cdcl-twl-cp*** *S′ T*⟩ **and**
⟨*literals-to-update T* = {#}⟩ **and**
⟨∀ *S′*. ¬ *cdcl-twl-cp T S′*⟩ **and**
⟨*twl-stgy-invs T*⟩
shows ⟨*cdcl-twl-stgy*** *S U*⟩
proof −
have ⟨*cdcl-twl-stgy*** *S S′*⟩
using *twl-stgy* **by** *fast*
moreover {
have ⟨*cdcl-twl-o*** *T U*⟩
using *twl-o* **by** *fast*
then have ⟨*cdcl-twl-stgy*** *S′ U*⟩
using *cp* **by** (*auto dest*!: *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD*)
}
ultimately show *?thesis* **by** *auto*
qed

lemma *cdcl-twl-stgy-prog-spec*:
assumes ⟨*twl-struct-invs S*⟩ **and** ⟨*twl-stgy-invs S*⟩ **and** ⟨*clauses-to-update S* = {#}⟩ **and**
⟨*get-conflict S* = *None*⟩
shows
⟨*cdcl-twl-stgy-prog S* ≤ *conclusive-TWL-run S*⟩
unfolding *cdcl-twl-stgy-prog-def full-def conclusive-TWL-run-def*
apply (*refine-vcg WHILEIT-rule*[**where**
R = ⟨{((*brkT*, *T*), (*brkS*, *S*)). *twl-struct-invs S* ∧ *cdcl-twl-stgy*⁺⁺ *S T*} ∪
{((*brkT*, *T*), (*brkS*, *S*)). *S* = *T* ∧ *brkT* ∧ ¬*brkS*}⟩];
remove-dummy-vars)
— Well foundedness of the relation
subgoal using *wf-cdcl-twl-stgy-measure* **.**

— initial invariants:
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*

— loop invariants:
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp*)
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by (*rule cdcl-twl-o-final-twl-state*)
subgoal by (*rule cdcl-twl-o-prog-cdcl-twl-stgy*)
subgoal by *simp*
subgoal for *brk0 T U brl V*
by *clarsimp*

— Final properties
subgoal for *brk0 T U V* — termination
by (*rule cdcl-twl-stgy-in-measure*)

subgoal by *simp*
subgoal by *fast*
done

definition *cdcl-twl-stgy-prog-break* :: ⟨*'v twl-st* ⇒ *'v twl-st nres*⟩ **where**
⟨*cdcl-twl-stgy-prog-break* S_0 =
do {
  b ← *SPEC*(λ-. *True*);
  (b, *brk*, T) ← $WHILE_T{}^{\lambda(b,\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}inv\ S_0\ S}$
    (λ(b, *brk*, -). b ∧ ¬*brk*)
    (λ(-, *brk*, S). *do* {
      T ← *unit-propagation-outer-loop* S;
      T ← *cdcl-twl-o-prog* T;
      b ← *SPEC*(λ-. *True*);
      RETURN (b, T)
    })
    (b, *False*, S_0);
  if brk then RETURN T
  else — finish iteration is required only
    cdcl-twl-stgy-prog T
}
⟩

lemma *wf-cdcl-twl-stgy-measure-break*:
⟨*wf* ({((*bT*, *brkT*, T), (*bS*, *brkS*, S)). *twl-struct-invs* S ∧ *cdcl-twl-stgy*$^{++}$ S T} ∪
    {((*bT*, *brkT*, T), (*bS*, *brkS*, S)). S = T ∧ *brkT* ∧ ¬*brkS*}
)⟩
  (**is** ⟨*?wf ?R*⟩)
proof −
  have *1*: ⟨*wf* ({((*brkT*, T), *brkS*, S). *twl-struct-invs* S ∧ *cdcl-twl-stgy*$^{++}$ S T} ∪
  {((*brkT*, T), *brkS*, S). S = T ∧ *brkT* ∧ ¬ *brkS*})⟩
  (**is** ⟨*wf ?S*⟩)
  by (*rule wf-cdcl-twl-stgy-measure*)
  have ⟨*wf* {((*bT*, T), (*bS*, S)). (T, S) ∈ *?S*}⟩
    apply (*rule wf-snd-wf-pair*)
    apply (*rule wf-subset*)
    apply (*rule 1*)
    apply *auto*
    done
  then show *?thesis*
    apply (*rule wf-subset*)
    apply *auto*
    done
qed

lemma *cdcl-twl-stgy-prog-break-spec*:
  assumes ⟨*twl-struct-invs S*⟩ **and** ⟨*twl-stgy-invs S*⟩ **and** ⟨*clauses-to-update* S = {#}⟩ **and**
  ⟨*get-conflict* S = *None*⟩
  shows
  ⟨*cdcl-twl-stgy-prog-break* S ≤ *conclusive-TWL-run* S⟩
  unfolding *cdcl-twl-stgy-prog-break-def full-def conclusive-TWL-run-def*
  apply (*refine-vcg cdcl-twl-stgy-prog-spec*[*unfolded conclusive-TWL-run-def*]
      WHILEIT-rule[**where**
    R = ⟨{((*bT*, *brkT*, T), (*bS*, *brkS*, S)). *twl-struct-invs* S ∧ *cdcl-twl-stgy*$^{++}$ S T} ∪
      {((*bT*, *brkT*, T), (*bS*, *brkS*, S)). S = T ∧ *brkT* ∧ ¬*brkS*}⟩];

remove-dummy-vars)
— Well foundedness of the relation
subgoal using *wf-cdcl-twl-stgy-measure-break* **.**

— initial invariants:
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*
subgoal using *assms* **by** *simp*

— loop invariants:
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp*)
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal for *x a aa ba xa x1a*
by (*rule cdcl-twl-o-final-twl-state*[*of S a aa ba*]) *simp-all*
subgoal for *x a aa ba xa x1a*
by (*rule cdcl-twl-o-prog-cdcl-twl-stgy*[*of S a aa ba xa x1a*]) *fast+*
subgoal by *simp*
subgoal for *brk0 T U brl V*
by *clarsimp*

— Final properties
subgoal for *x a aa ba xa xb* — termination
using *cdcl-twl-stgy-in-measure*[*of S a aa ba xa*] **by** *fast*
subgoal by *simp*
subgoal by *fast*

— second loop
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal using *assms* **by** *auto*
done

end
theory *Watched-Literals-Transition-System-Restart*
imports *Watched-Literals-Transition-System*
begin

Unlike the basic CDCL, it does not make any sense to fully restart the trail: the part propagated at level 0 (only the part due to unit clauses) has to be kept. Therefore, we allow fast restarts (i.e. a restart where part of the trail is reused).

There are two cases:

- either the trail is strictly decreasing;
- or it is kept and the number of clauses is strictly decreasing.

This ensures that *something* changes to prove termination.

In practice, there are two types of restarts that are done:

- First, a restart can be done to enforce that the SAT solver goes more into the direction expected by the decision heuristics.
- Second, a full restart can be done to simplify inprocessing and garbage collection of the memory: instead of properly updating the trail, we restart the search. This is not necessary (i.e., glucose and minisat do not do it), but it simplifies the proofs by allowing to move clauses without taking care of updating references in the trail. Moreover, as this happens "rarely" (around once every few thousand conflicts), it should not matter too much.

Restarts are the "local search" part of all modern SAT solvers.

inductive *cdcl-twl-restart* :: ‹'v twl-st ⇒ 'v twl-st ⇒ bool› **where**
restart-trail:
  ‹*cdcl-twl-restart* (M, N, U, *None*, NE, UE, {#}, Q)
      (M', N', U', *None*, NE + *clauses* NE', UE + *clauses* UE', {#}, {#})›
 if
  ‹(*Decided* K # M', $M2$) ∈ *set* (*get-all-ann-decomposition* M)› **and**
  ‹U' + UE' ⊆# U› **and**
  ‹N = N' + NE'› **and**
  ‹∀ E∈#NE'+UE'. ∃ L∈#*clause* E. L ∈ *lits-of-l* M' ∧ *get-level* M' L = *0*›
  ‹∀ L E. *Propagated* L E ∈ *set* M' ⟶ E ∈# *clause* '# (N + U') + NE + UE + *clauses* UE'› |
restart-clauses:
  ‹*cdcl-twl-restart* (M, N, U, *None*, NE, UE, {#}, Q)
    (M, N', U', *None*, NE + *clauses* NE', UE + *clauses* UE', {#}, Q)›
 if
  ‹U' + UE' ⊆# U› **and**
  ‹N = N' + NE'› **and**
  ‹∀ E∈#NE'+UE'. ∃ L∈#*clause* E. L ∈ *lits-of-l* M ∧ *get-level* M L = *0*›
  ‹∀ L E. *Propagated* L E ∈ *set* M ⟶ E ∈# *clause* '# (N + U') + NE + UE + *clauses* UE'›

inductive-cases *cdcl-twl-restartE*: ‹*cdcl-twl-restart* S T›

lemma *cdcl-twl-restart-cdcl*$_W$*-stgy*:
 assumes
  ‹*cdcl-twl-restart* S V› **and**
  ‹*twl-struct-invs* S› **and**
  ‹*twl-stgy-invs* S›
 shows
  ‹∃ T. *cdcl*$_W$*-restart-mset.restart* (*state*$_W$*-of* S) T ∧ *cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** T (*state*$_W$*-of* V) ∧
    cdcl$_W$*-restart-mset.cdcl*$_W$*-restart*** (*state*$_W$*-of* S) (*state*$_W$*-of* V)›
 using *assms*
proof (*induction rule*: *cdcl-twl-restart.induct*)
 case (*restart-trail* K M' $M2$ M U' UE' U N N' NE' NE UE Q)
 note *decomp* = *this*(*1*) **and** *learned* = *this*(*2*) **and** N = *this*(*3*) **and**
  has-true = *this*(*4*) **and** *kept* = *this*(*5*) **and** *inv* = *this*(*6*) **and** *stgy-invs* = *this*(*7*)
 let *?S* = ‹(M, N, U, *None*, NE, UE, {#}, Q)›
 let *?T* = ‹([], *clause* '# N + NE, *clause* '# U' + UE + *clauses* UE', *None*)›
 let *?V* = ‹(M', N, U', *None*, NE, UE + *clauses* UE', {#}, {#})›
 have *restart*: ‹*cdcl*$_W$*-restart-mset.restart* (*state*$_W$*-of ?S*) *?T*›
  using *learned*

by (*auto simp*: *cdcl*$_W$*-restart-mset.restart.simps state-def clauses-def cdcl*$_W$*-restart-mset-state*
intro: *image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
have *struct-invs*:
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of ?S*)› **and**
smaller-propa:
‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of ?S*)›
using *inv* **unfolding** *twl-struct-invs-def* **by** *fast+*
have *drop-M-M′*: ‹*drop* (*length M* − *length M′*) *M* = *M′*›
using *decomp* **by** (*auto*)
have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** *?T*
(*drop* (*length M* − *length M′*) *M*, *clause* ‘# *N* + *NE*, *clause* ‘# *U′* + *UE* + *clauses UE′*, *None*)›
for *n*
apply (*rule after-fast-restart-replay*[*of M* ‹*clause* ‘# *N* + *NE*› ‹*clause* ‘# *U*+*UE*› -
‹*clause* ‘# *U′* + *UE* +*clauses UE′*›])
subgoal using *struct-invs* **by** *simp*
subgoal using *stgy-invs* **unfolding** *twl-stgy-invs-def* **by** *simp*
subgoal using *smaller-propa* **by** *simp*
subgoal using *kept* **unfolding** *drop-M-M′* **by** (*auto simp add*: *ac-simps*)
subgoal using *learned*
by (*auto simp*: *image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
done
then have *st*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** *?T* (*state*$_W$*-of ?V*)›
unfolding *drop-M-M′* **by** (*simp add*: *ac-simps*)
moreover have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-restart*** (*state*$_W$*-of ?S*) (*state*$_W$*-of ?V*)›
using *restart st*
by (*auto dest*!: *cdcl*$_W$*-restart-mset.cdcl*$_W$*-rf.intros cdcl*$_W$*-restart-mset.cdcl*$_W$*-restart.intros*
cdcl$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-rtranclp-cdcl*$_W$*-restart*)
ultimately show *?case*
using *restart* **unfolding** *N state*$_W$*-of.simps image-mset-union add.assoc*
add.commute[*of* ‹*clauses NE′*›]
by *fast*
next
case (*restart-clauses U′ UE′ U N N′ NE′ M NE UE Q*)
note *learned* = *this*(*1*) **and** *N* = *this*(*2*) **and** *has-true* = *this*(*3*) **and** *kept* = *this*(*4*) **and**
inv = *this*(*5*) **and** *stgy-invs* = *this*(*6*)
let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*)›
let *?T* = ‹([], *clause* ‘# *N* + *NE*, *clause* ‘# *U′* + *UE* + *clauses UE′*, *None*)›
let *?V* = ‹(*M*, *N*, *U′*, *None*, *NE*, *UE* + *clauses UE′*, {#}, {#})›
have *restart*: ‹*cdcl*$_W$*-restart-mset.restart* (*state*$_W$*-of ?S*) *?T*›
using *learned*
by (*auto simp*: *cdcl*$_W$*-restart-mset.restart.simps state-def clauses-def cdcl*$_W$*-restart-mset-state*
intro!: *image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
have *struct-invs*:
‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*))› **and**
smaller-propa:
‹*cdcl*$_W$*-restart-mset.no-smaller-propa* (*state*$_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*))›
using *inv* **unfolding** *twl-struct-invs-def* **by** *fast+*

have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** *?T*
(*drop* (*length M* − *length M*) *M*, *clause* ‘# *N* + *NE*, *clause* ‘# *U′* + *UE*+ *clauses UE′*, *None*)›
for *n*
apply (*rule after-fast-restart-replay*[*of M* ‹*clause* ‘# *N* + *NE*› ‹*clause* ‘# *U*+*UE*› -
‹*clause* ‘# *U′* + *UE* + *clauses UE′*›])
subgoal using *struct-invs* **by** *simp*
subgoal using *stgy-invs* **unfolding** *twl-stgy-invs-def* **by** *simp*
subgoal using *smaller-propa* **by** *simp*

subgoal using *kept* **by** (*auto simp add*: *ac-simps*)
subgoal using *learned*
by (*auto simp*: *image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
done
then have *st*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy*** *?T* ($state_W$*-of ?V*)›
by (*simp add*: *ac-simps*)
moreover have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-restart*** ($state_W$*-of ?S*) ($state_W$*-of ?V*)›
using *restart st*
by (*auto dest!*: $cdcl_W$*-restart-mset.*$cdcl_W$*-rf.intros* $cdcl_W$*-restart-mset.*$cdcl_W$*-restart.intros*
$cdcl_W$*-restart-mset.rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$*-restart*)
ultimately show *?case*
using *restart* **unfolding** *N* $state_W$*-of.simps image-mset-union add.assoc*
add.commute[*of* ‹*clauses NE'*›]
by *fast*
qed

lemma *cdcl-twl-restart-*$cdcl_W$:
assumes
‹*cdcl-twl-restart S V*› **and**
‹*twl-struct-invs S*›
shows
‹∃ *T*. $cdcl_W$*-restart-mset.restart* ($state_W$*-of S*) *T* ∧ $cdcl_W$*-restart-mset.*$cdcl_W$** *T* ($state_W$*-of V*)›
using *assms*
proof (*induction rule*: *cdcl-twl-restart.induct*)
case (*restart-trail K M' M2 M U' UE' U N N' NE' NE UE Q*)
note *decomp* = *this*(*1*) **and** *learned* = *this*(*2*) **and** *N* = *this*(*3*) **and**
has-true = *this*(*4*) **and** *kept* = *this*(*5*) **and** *inv* = *this*(*6*)
let *?S* = ‹(*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*)›
let *?T* = ‹([], *clause* ‘# *N* + *NE*, *clause* ‘# *U'* + *UE* + *clauses UE'*, *None*)›
let *?V* = ‹(*M'*, *N*, *U'*, *None*, *NE*, *UE* + *clauses UE'*, {#}, {#})›
have *restart*: ‹$cdcl_W$*-restart-mset.restart* ($state_W$*-of ?S*) *?T*›
using *learned*
by (*auto simp*: $cdcl_W$*-restart-mset.restart.simps state-def clauses-def* $cdcl_W$*-restart-mset-state*
image-mset-subseteq-mono[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
have *struct-invs*:
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*))› **and**
smaller-propa:
‹$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*))›
using *inv* **unfolding** *twl-struct-invs-def* **by** *fast+*
have *drop-M-M'*: ‹*drop* (*length M* − *length M'*) *M* = *M'*›
using *decomp* **by** (*auto*)
have ‹$cdcl_W$*-restart-mset.*$cdcl_W$** *?T*
(*drop* (*length M* − *length M'*) *M*, *clause* ‘# *N* + *NE*, *clause* ‘# *U'* + *UE*+ *clauses UE'*, *None*)›
for *n*
apply (*rule after-fast-restart-replay-no-stgy*[*of M* ‹*clause* ‘# *N* + *NE*› ‹*clause* ‘# *U*+*UE*› -
‹*clause* ‘# *U'* + *UE* + *clauses UE'*›])
subgoal using *struct-invs* **by** *simp*
subgoal using *kept* **unfolding** *drop-M-M'* **by** (*auto simp add*: *ac-simps*)
subgoal using *learned*
by (*auto simp*: *image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
done
then have *st*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$** *?T* ($state_W$*-of ?V*)›
unfolding *drop-M-M'* **by** (*simp add*: *ac-simps*)
then show *?case*
using *restart* **by** (*auto simp*: *ac-simps N*)
next

case (*restart-clauses* U' UE' U N N' NE' M NE UE Q)
note *learned* = *this*(*1*) **and** N = *this*(*2*) **and** *has-true* = *this*(*3*) **and** *kept* = *this*(*4*) **and** *inv* = *this*(*5*)
let *?S* = ‹(M, N, U, *None*, NE, UE, {#}, Q)›
let *?T* = ‹([], *clause* ‘# N + NE, *clause* ‘# U' + UE + *clauses* UE', *None*)›
let *?V* = ‹(M, N, U', *None*, NE, UE + *clauses* UE', {#}, {#})›
have *restart*: ‹$cdcl_W$*-restart-mset.restart* ($state_W$*-of ?S*) *?T*›
using *learned*
by (*auto simp*: $cdcl_W$*-restart-mset.restart.simps state-def clauses-def* $cdcl_W$*-restart-mset-state image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
have *struct-invs*:
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of ?S*)› **and**
smaller-propa:
‹$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of ?S*)›
using *inv* **unfolding** *twl-struct-invs-def* **by** *fast+*
have ‹$cdcl_W$*-restart-mset.*$cdcl_W{}^{**}$ *?T*
(*drop* (*length* M − *length* M) M, *clause* ‘# N + NE, *clause* ‘# U' + UE+ *clauses* UE', *None*)›
for n
apply (*rule after-fast-restart-replay-no-stgy*[*of* M ‹*clause* ‘# N + NE› ‹*clause* ‘# U+UE› -
‹*clause* ‘# U' + UE+ *clauses* UE'›])
subgoal using *struct-invs* **by** *simp*
subgoal using *kept* **by** (*auto simp add*: *ac-simps*)
subgoal
using *learned* **by** (*auto simp*: *image-mset-subseteq-mono*[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
done
then have *st*: ‹$cdcl_W$*-restart-mset.*$cdcl_W{}^{**}$ *?T* ($state_W$*-of ?V*)›
by (*simp add*: *ac-simps*)
then show *?case*
using *restart* **by** (*auto simp*: *ac-simps* N)
qed

lemma *cdcl-twl-restart-twl-struct-invs*:
assumes
‹*cdcl-twl-restart* S T› **and**
‹*twl-struct-invs* S›
shows ‹*twl-struct-invs* T›
using *assms*
proof (*induction rule*: *cdcl-twl-restart.induct*)
case (*restart-trail* K M' *M2* M U' UE' U N N' NE' NE UE Q)
note *decomp* = *this*(*1*) **and** *learned'* = *this*(*2*) **and** N = *this*(*3*) **and**
has-true = *this*(*4*) **and** *kept* = *this*(*5*) **and** *inv* = *this*(*6*)
let *?S* = ‹(M, N, U, *None*, NE, UE, {#}, Q)›
let *?S'* = ‹(M', N', U', *None*, NE+ *clauses* NE', UE + *clauses* UE', {#}, {#})›
have *learned*: ‹$U' \subseteq\# U$›
using *learned'* **by** (*rule mset-le-decr-left1*)
have
twl-st-inv: ‹*twl-st-inv ?S*› **and**
‹*valid-enqueued ?S*› **and**
struct-inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv*
($state_W$*-of ?S*)› **and**
smaller: ‹$cdcl_W$*-restart-mset.no-smaller-propa*
($state_W$*-of ?S*)› **and**
‹*twl-st-exception-inv ?S*› **and**
no-dup-q: ‹*no-duplicate-queued ?S*› **and**
dist: ‹*distinct-queued ?S*› **and**
‹*confl-cands-enqueued ?S*› **and**

⟨*propa-cands-enqueued ?S*⟩ **and**
⟨*get-conflict ?S* ≠ *None* ⟶
clauses-to-update ?S = {#} ∧
literals-to-update ?S = {#}⟩ **and**
unit: ⟨*entailed-clss-inv ?S*⟩ **and**
to-upd: ⟨*clauses-to-update-inv ?S*⟩ **and**
past: ⟨*past-invs ?S*⟩
using *inv* **unfolding** *twl-struct-invs-def* **by** *clarify+*
have
ex: ⟨(∀ *C*∈#*N* + *U*. *twl-lazy-update M′ C* ∧
watched-literals-false-of-max-level M′ C ∧
twl-exception-inv (*M′*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) *C*)⟩ **and**
conf-cands: ⟨*confl-cands-enqueued* (*M′*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})⟩ **and**
propa-cands: ⟨*propa-cands-enqueued* (*M′*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})⟩ **and**
clss-to-upd: ⟨*clauses-to-update-inv* (*M′*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})⟩
using *past get-all-ann-decomposition-exists-prepend*[*OF decomp*] **unfolding** *past-invs.simps*
by *force+*

have *excp-inv*: ⟨*twl-st-exception-inv* (*M′*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})⟩
using *ex* **unfolding** *twl-st-exception-inv.simps* **by** *blast+*
have *twl-st-inv′*: ⟨*twl-st-inv* (*M′*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})⟩
using *ex learned twl-st-inv*
unfolding *twl-st-exception-inv.simps twl-st-inv.simps*
by *auto*
have *n-d*: ⟨*no-dup M*⟩
using *struct-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
obtain *M3* **where**
M: ⟨*M* = *M3* @ *M2* @ *Decided K* # *M′*⟩
using *decomp* **by** *blast*
define *M3′* **where** ⟨*M3′* = *M3* @ *M2*⟩
then have *M3′*: ⟨*M* = *M3′* @ *Decided K* # *M′*⟩
unfolding *M* **by** *auto*
have *entailed-clss-inv*: ⟨*entailed-clss-inv ?S′*⟩
unfolding *entailed-clss-inv.simps*
proof
fix *C*
assume ⟨*C* ∈# *NE* + *clauses NE′* + (*UE* + *clauses UE′*)⟩
moreover have ⟨*L* ∈ *lits-of-l M* ∧ *get-level M L* = *0* ⟹*L* ∈ *lits-of-l M′* ∧ *get-level M′ L* = *0*⟩
for *L*
using *n-d*
by (*cases* ⟨*undefined-lit M3′ L*⟩)
(*auto simp*: *M3′ atm-of-eq-atm-of get-level-cons-if*
dest: *in-lits-of-l-defined-litD split*: *if-splits*)
ultimately obtain *L* **where**
lev-L: ⟨*get-level M′ L* = *0*⟩
⟨*L* ∈ *lits-of-l M′*⟩ **and**
C: ⟨*L* ∈# *C*⟩
using *unit has-true* **by** *auto blast+*
then have ⟨*L* ∈ *lits-of-l M′*⟩
apply (*cases* ⟨*defined-lit M3′ L*⟩)
using *n-d* **unfolding** *M3′* **by** (*auto simp*: *get-level-cons-if split*: *if-splits*
dest: *in-lits-of-l-defined-litD*)
moreover have ⟨*get-level M′ L* = *0*⟩
apply (*cases* ⟨*defined-lit M3′ L*⟩)
using *n-d lev-L* **unfolding** *M3′* **by** (*auto simp*: *get-level-cons-if split*: *if-splits*

dest: *in-lits-of-l-defined-litD*)
ultimately show ‹∃ *L*. *L* ∈# *C* ∧
(*None* = *None* ∨ *0* < *count-decided M'* ⟶
get-level M' L = *0* ∧ *L* ∈ *lits-of-l M'*)›
using *C* **by** *blast*
qed
have *a*: ‹*N* ⊆# *N*› **and** *NN'*: ‹*N'* ⊆# *N*› **using** *N* **by** *auto*
have *past-invs*: ‹*past-invs ?S'*›
unfolding *past-invs.simps*
proof (*intro conjI impI allI*)
fix *M1 M2 K'*
assume *H*:‹*M'* = *M2* @ *Decided K'* # *M1*›
let *?U* = ‹(*M1*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#})›
let *?U'* = ‹(*M1*, *N'*, *U'*, *None*, *NE*+*clauses NE'*, *UE*+*clauses UE'*, {#}, {#})›
have ‹*M* = (*M3'* @ *Decided K* # *M2*) @ *Decided K'* # *M1*›
using *H M3'* **by** *simp*
then have
1: ‹∀ *C*∈#*N* + *U*.
twl-lazy-update M1 C ∧
watched-literals-false-of-max-level M1 C ∧
twl-exception-inv ?U C› **and**
2: ‹*confl-cands-enqueued ?U*› **and**
3: ‹*propa-cands-enqueued ?U*› **and**
4: ‹*clauses-to-update-inv ?U*›
using *past* **unfolding** *past-invs.simps* **by** *blast*+
show ‹∀ *C*∈#*N'* + *U'*.
twl-lazy-update M1 C ∧
watched-literals-false-of-max-level M1 C ∧
twl-exception-inv ?U' C›
using *1 learned twl-st-exception-inv-mono*[*OF learned NN'*, *of M1 None NE* ‹*UE*›
‹{#}› ‹{#}› ‹*NE*+*clauses NE'*› ‹*UE*+*clauses UE'*›] *N* **by** *auto*
show ‹*confl-cands-enqueued ?U'*›
using *confl-cands-enqueued-mono*[*OF learned NN' 2*] **.**
show ‹*propa-cands-enqueued ?U'*›
using *propa-cands-enqueued-mono*[*OF learned NN' 3*] **.**
show ‹*clauses-to-update-inv ?U'*›
using *4 learned* **by** (*auto simp add*: *filter-mset-empty-conv N*)
qed
have *clss-to-upd*: ‹*clauses-to-update-inv ?S'*›
using *clss-to-upd learned* **by** (*auto simp add*: *filter-mset-empty-conv N*)

have [*simp*]: ‹$cdcl_W$*-restart-mset.*$cdcl_W$ ≤ $cdcl_W$*-restart-mset.*$cdcl_W$*-restart*›
using $cdcl_W$*-restart-mset.*$cdcl_W$*-*$cdcl_W$*-restart* **by** *blast*

obtain *T'* **where**
res: ‹$cdcl_W$*-restart-mset.restart* ($state_W$*-of ?S*) *T'*› **and**
res': ‹$cdcl_W$*-restart-mset.*$cdcl_W^{**}$ *T'* ($state_W$*-of ?S'*)›
using *cdcl-twl-restart-*$cdcl_W$[*OF cdcl-twl-restart.restart-trail*[*OF restart-trail*(*1*−*5*)] *inv*]
by (*auto simp*: *ac-simps N*)
then have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-restart*** ($state_W$*-of ?S*)
($state_W$*-of ?S'*)›
using *rtranclp-mono*[*of* $cdcl_W$*-restart-mset.*$cdcl_W$ $cdcl_W$*-restart-mset.*$cdcl_W$*-restart*]
$cdcl_W$*-restart-mset.*$cdcl_W$*-*$cdcl_W$*-restart*
by (*auto dest*!: $cdcl_W$*-restart-mset.*$cdcl_W$*-restart.intros*
$cdcl_W$*-restart-mset.*$cdcl_W$*-rf.intros*)
from $cdcl_W$*-restart-mset.rtranclp-*$cdcl_W$*-all-struct-inv-inv*[*OF this struct-inv*]

have *struct-inv′*:
⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of* (*M′*, *N′*, *U′*, *None*, *NE*+ *clauses NE′*, *UE*+ *clauses UE′*, {#}, {#}))⟩
by (*auto simp*: *ac-simps N*)
have *smaller′*:
⟨$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of* (*M′*, *N′*, *U′*, *None*, *NE*+ *clauses NE′*, *UE*+ *clauses UE′*, {#}, {#}))⟩
using *smaller mset-subset-eqD*[*OF learned′*]
apply (*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def M3′* $cdcl_W$*-restart-mset-state clauses-def ac-simps N*)
apply (*metis Cons-eq-appendI append-assoc image-eqI*)
apply (*metis Cons-eq-appendI append-assoc image-eqI*)
done

show *?case*
unfolding *twl-struct-invs-def*
apply (*intro conjI*)
subgoal using *twl-st-inv-mono*[*OF learned NN′ twl-st-inv′*] **by** (*auto simp*: *ac-simps N*)
subgoal by *simp*
subgoal by (*rule struct-inv′*)
subgoal by (*rule smaller′*)
subgoal using *twl-st-exception-inv-mono*[*OF learned NN′ excp-inv*] **.**
subgoal using *no-dup-q* **by** *auto*
subgoal using *dist* **by** *auto*
subgoal using *confl-cands-enqueued-mono*[*OF learned NN′ conf-cands*] **.**
subgoal using *propa-cands-enqueued-mono*[*OF learned NN′ propa-cands*] **.**
subgoal by *simp*
subgoal by (*rule entailed-clss-inv*)
subgoal by (*rule clss-to-upd*)
subgoal by (*rule past-invs*)
done
next
case (*restart-clauses U′ UE′ U N N′ NE′ M NE UE Q*)
note *learned′* = *this*(*1*) **and** *N* = *this*(*2*) **and** *has-true* = *this*(*3*) **and** *kept* = *this*(*4*) **and** *invs* = *this*(*5*)
let *?S* = ⟨(*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, *Q*)⟩
let *?T* = ⟨(*M*, *N′*, *U′*, *None*, *NE*+*clauses NE′*, *UE* + *clauses UE′*, {#}, *Q*)⟩
have *learned*: ⟨*U′* ⊆# *U*⟩
using *learned′* **by** (*rule mset-le-decr-left1*)
have
twl-st-inv: ⟨*twl-st-inv ?S*⟩ **and**
valid: ⟨*valid-enqueued ?S*⟩ **and**
struct-inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv*
($state_W$*-of ?S*)⟩ **and**
smaller: ⟨$cdcl_W$*-restart-mset.no-smaller-propa*
($state_W$*-of ?S*)⟩ **and**
excp-inv: ⟨*twl-st-exception-inv ?S*⟩ **and**
no-dup-q: ⟨*no-duplicate-queued ?S*⟩ **and**
dist: ⟨*distinct-queued ?S*⟩ **and**
confl-cands: ⟨*confl-cands-enqueued ?S*⟩ **and**
propa-cands: ⟨*propa-cands-enqueued ?S*⟩ **and**
⟨*get-conflict ?S* ≠ *None* ⟶
clauses-to-update ?S = {#} ∧
literals-to-update ?S = {#}⟩ **and**
unit: ⟨*entailed-clss-inv ?S*⟩ **and**
to-upd: ⟨*clauses-to-update-inv ?S*⟩ **and**

past: ⟨*past-invs ?S*⟩
using *invs* **unfolding** *twl-struct-invs-def* **by** *clarify+*
have *learned*: ⟨$U' \subseteq\# \ U$⟩
using *learned* **by** *auto*
have *n-d*: ⟨*no-dup M*⟩
using *struct-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *trail.simps*)
have *valid'*: ⟨*valid-enqueued ?T*⟩
using *valid* **by** *auto*
have *entailed-clss-inv*: ⟨*entailed-clss-inv ?T*⟩
unfolding *entailed-clss-inv.simps*
proof
fix *C*
assume ⟨$C \in\# \ NE + clauses \ NE' + (UE + clauses \ UE')$⟩
then obtain *L* **where**
lev-L: ⟨$get\text{-}level \ M \ L = 0$⟩
⟨$L \in lits\text{-}of\text{-}l \ M$⟩ **and**
C: ⟨$L \in\# \ C$⟩
using *unit has-true* **by** *auto*
then show ⟨$\exists L. \ L \in\# \ C \wedge$
$(None = None \vee 0 < count\text{-}decided \ M \longrightarrow$
$get\text{-}level \ M \ L = 0 \wedge L \in lits\text{-}of\text{-}l \ M)$⟩
using *C* **by** *blast*
qed
have *NN'*: ⟨$N' \subseteq\# \ N$⟩ **unfolding** *N* **by** *auto*
have *past-invs*: ⟨*past-invs* (M, N', U', *None*, $NE+clauses \ NE'$, $UE + clauses \ UE'$, {#}, Q)⟩
using *past* **unfolding** *past-invs.simps*
proof (*intro conjI impI allI*)
fix *M1 M2 K'*
assume *H*:⟨$M = M2 \ @ \ Decided \ K' \ \# \ M1$⟩
let *?U* = ⟨($M1$, N, U, *None*, NE, UE, {#}, {#})⟩
let *?U'* = ⟨($M1$, N', U', *None*, $NE+clauses \ NE'$, $UE + clauses \ UE'$, {#}, {#})⟩
have
1: ⟨$\forall \ C \in\# N + U.$
twl-lazy-update M1 C ∧
watched-literals-false-of-max-level M1 C ∧
twl-exception-inv ?U C⟩ **and**
2: ⟨*confl-cands-enqueued ?U*⟩ **and**
3: ⟨*propa-cands-enqueued ?U*⟩ **and**
4: ⟨*clauses-to-update-inv ?U*⟩
using *H past* **unfolding** *past-invs.simps* **by** *blast+*
show ⟨$\forall \ C \in\# N' + U'.$
twl-lazy-update M1 C ∧
watched-literals-false-of-max-level M1 C ∧
twl-exception-inv ?U' C⟩
using *1 learned twl-st-exception-inv-mono*[*OF learned NN'*, *of M1 None NE UE* ⟨{#}⟩ ⟨{#}⟩] *N*
by *auto*
show ⟨*confl-cands-enqueued ?U'*⟩
using *confl-cands-enqueued-mono*[*OF learned NN' 2*] **.**
show ⟨*propa-cands-enqueued ?U'*⟩
using *propa-cands-enqueued-mono*[*OF learned NN' 3*] **.**
show ⟨*clauses-to-update-inv ?U'*⟩
using *4 learned* **by** (*auto simp add*: *filter-mset-empty-conv N*)
qed

have [*simp*]: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$ ≤ $cdcl_W$*-restart-mset.*$cdcl_W$*-restart*⟩

using $cdcl_W$-*restart-mset*.$cdcl_W$-$cdcl_W$-*restart* **by** *blast*

have *clss-to-upd*: ⟨*clauses-to-update-inv ?T*⟩
 using *to-upd learned* **by** (*auto simp add*: *filter-mset-empty-conv N ac-simps*)
 obtain T' **where**
 res: ⟨$cdcl_W$-*restart-mset*.*restart* ($state_W$-*of ?S*) T'⟩ **and**
 res': ⟨$cdcl_W$-*restart-mset*.$cdcl_W^{**}$ T' ($state_W$-*of ?T*)⟩
 using *cdcl-twl-restart*-$cdcl_W$[*OF cdcl-twl-restart*.*restart-clauses*[*OF restart-clauses*(*1*−*4*)] *invs*]
 by (*auto simp*: *ac-simps N*)
then have ⟨$cdcl_W$-*restart-mset*.$cdcl_W$-*restart*** ($state_W$-*of ?S*)
 ($state_W$-*of ?T*)⟩
 using *rtranclp-mono*[*of* $cdcl_W$-*restart-mset*.$cdcl_W$ $cdcl_W$-*restart-mset*.$cdcl_W$-*restart*]
 $cdcl_W$-*restart-mset*.$cdcl_W$-$cdcl_W$-*restart*
 by (*auto dest*!: $cdcl_W$-*restart-mset*.$cdcl_W$-*restart*.*intros*
 $cdcl_W$-*restart-mset*.$cdcl_W$-*rf*.*intros*)
from $cdcl_W$-*restart-mset*.*rtranclp*-$cdcl_W$-*all-struct-inv-inv*[*OF this struct-inv*]
have *struct-inv'*:
 ⟨$cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv* ($state_W$-*of ?T*)⟩
 .

have *smaller'*:
 ⟨$cdcl_W$-*restart-mset*.*no-smaller-propa* ($state_W$-*of ?T*)⟩
 using *smaller mset-subset-eqD*[*OF learned'*]
 by (*auto 5 5 simp*: $cdcl_W$-*restart-mset*.*no-smaller-propa-def* $cdcl_W$-*restart-mset-state*
 clauses-def N ac-simps)

show *?case*
 unfolding *twl-struct-invs-def*
 apply (*intro conjI*)
 subgoal using *twl-st-inv-mono*[*OF learned NN' twl-st-inv*] **.**
 subgoal by (*rule valid'*)
 subgoal by (*rule struct-inv'*)
 subgoal by (*rule smaller'*)
 subgoal using *twl-st-exception-inv-mono*[*OF learned NN' excp-inv*] **.**
 subgoal using *no-dup-q* **by** *auto*
 subgoal using *dist* **by** *auto*
 subgoal using *confl-cands-enqueued-mono*[*OF learned NN' confl-cands*] **.**
 subgoal using *propa-cands-enqueued-mono*[*OF learned NN' propa-cands*] **.**
 subgoal by *simp*
 subgoal by (*rule entailed-clss-inv*)
 subgoal by (*rule clss-to-upd*)
 subgoal by (*rule past-invs*)
 done
qed

lemma *rtranclp-cdcl-twl-restart-twl-struct-invs*:
 assumes
 ⟨*cdcl-twl-restart*** *S T*⟩ **and**
 ⟨*twl-struct-invs S*⟩
 shows ⟨*twl-struct-invs T*⟩
 using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto simp*: *cdcl-twl-restart-twl-struct-invs*)

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

using *assms*
by (*induction rule*: *cdcl-twl-restart.induct*)
(*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*
conflicting.simps $cdcl_W$*-restart-mset.no-smaller-confl-def clauses-def trail.simps*
dest!: *get-all-ann-decomposition-exists-prepend*)

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

context *twl-restart-ops*
begin

inductive *cdcl-twl-stgy-restart* :: ‹$'v$ *twl-st* × *nat* ⇒ $'v$ *twl-st* × *nat* ⇒ *bool*› **where**
restart-step:
‹*cdcl-twl-stgy-restart* (*S*, *n*) (*U*, *Suc n*)›
if
‹*cdcl-twl-stgy*$^{++}$ *S T*› **and**
‹*size* (*get-learned-clss T*) > *f n*› **and**
‹*cdcl-twl-restart T U*› |
restart-full:
‹*cdcl-twl-stgy-restart* (*S*, *n*) (*T*, *n*)›
if
‹*full1 cdcl-twl-stgy S T*›

lemma *cdcl-twl-stgy-restart-init-clss*:
assumes ‹*cdcl-twl-stgy-restart S T*›
shows
‹*get-all-init-clss* (*fst S*) = *get-all-init-clss* (*fst T*)›
by (*use assms* **in** ‹*induction rule*: *cdcl-twl-stgy-restart.induct*›)
(*auto simp*: *full1-def cdcl-twl-restart.simps*
dest: *rtranclp-cdcl-twl-stgy-all-learned-diff-learned dest*!: *tranclp-into-rtranclp*)

lemma *rtranclp-cdcl-twl-stgy-restart-init-clss*:
assumes ‹*cdcl-twl-stgy-restart*** *S T*›
shows
‹*get-all-init-clss* (*fst S*) = *get-all-init-clss* (*fst T*)›
by (*use assms* **in** ‹*induction rule*: *rtranclp-induct*›)
(*auto simp*: *full1-def dest*: *cdcl-twl-stgy-restart-init-clss*)

lemma *cdcl-twl-stgy-restart-twl-struct-invs*:
assumes
‹*cdcl-twl-stgy-restart S T*› **and**
‹*twl-struct-invs* (*fst S*)›
shows ‹*twl-struct-invs* (*fst T*)›
using *assms*
by (*induction rule*: *cdcl-twl-stgy-restart.induct*)
(*auto simp add*: *full1-def intro*: *rtranclp-cdcl-twl-stgy-twl-struct-invs tranclp-into-rtranclp*
cdcl-twl-restart-twl-struct-invs rtranclp-cdcl-twl-stgy-twl-stgy-invs)

lemma *rtranclp-cdcl-twl-stgy-restart-twl-struct-invs*:

assumes
⟨*cdcl-twl-stgy-restart*** *S T*⟩ **and**
⟨*twl-struct-invs* (*fst S*)⟩
shows ⟨*twl-struct-invs* (*fst T*)⟩
using *assms*
by (*induction*)
(*auto intro*: *cdcl-twl-stgy-restart-twl-struct-invs*)

lemma *cdcl-twl-stgy-restart-twl-stgy-invs*:
assumes
⟨*cdcl-twl-stgy-restart S T*⟩ **and**
⟨*twl-struct-invs* (*fst S*)⟩ **and**
⟨*twl-stgy-invs* (*fst S*)⟩
shows ⟨*twl-stgy-invs* (*fst T*)⟩
using *assms*
by (*induction rule*: *cdcl-twl-stgy-restart.induct*)
(*auto simp add*: *full1-def dest*!: *tranclp-into-rtranclp*
intro: *cdcl-twl-restart-twl-stgy-invs rtranclp-cdcl-twl-stgy-twl-stgy-invs*)

lemma *no-step-cdcl-twl-stgy-restart-cdcl-twl-stgy*:
assumes
ns: ⟨*no-step cdcl-twl-stgy-restart S*⟩ **and**
⟨*twl-struct-invs* (*fst S*)⟩
shows
⟨*no-step cdcl-twl-stgy* (*fst S*)⟩
proof (*rule ccontr*)
assume ⟨¬ *?thesis*⟩
then obtain *T* **where** *T*: ⟨*cdcl-twl-stgy* (*fst S*) *T*⟩ **by** *blast*
then have ⟨*twl-struct-invs T*⟩
using *assms*(*2*) *cdcl-twl-stgy-twl-struct-invs* **by** *blast*
obtain *U* **where** *U*: ⟨*full* (λ*S T*. *twl-struct-invs S* ∧ *cdcl-twl-stgy S T*) *T U*⟩
using *wf-exists-normal-form-full*[*OF wf-cdcl-twl-stgy*] **by** *blast*
have ⟨*full cdcl-twl-stgy T U*⟩
proof −
have
st: ⟨(λ*S T*. *twl-struct-invs S* ∧ *cdcl-twl-stgy S T*)** *T U*⟩ **and**
ns: ⟨*no-step* (λ*U V*. *twl-struct-invs U* ∧ *cdcl-twl-stgy U V*) *U*⟩
using *U* **unfolding** *full-def* **by** *blast*+
have ⟨*cdcl-twl-stgy*** *T U*⟩
using *st* **by** (*induction rule*: *rtranclp-induct*) *auto*
moreover have ⟨*no-step cdcl-twl-stgy U*⟩
using *ns* ⟨*twl-struct-invs T*⟩ *calculation rtranclp-cdcl-twl-stgy-twl-struct-invs* **by** *blast*
ultimately show *?thesis*
unfolding *full-def* **by** *blast*
qed
then have ⟨*full1 cdcl-twl-stgy* (*fst S*) *U*⟩
using *T* **by** (*auto intro*: *full-fullI*)
then show *False*
using *ns cdcl-twl-stgy-restart.intros*(*2*)[*of* ⟨*fst S*⟩ *U* ⟨*snd S*⟩]
by *fastforce*
qed

lemma (**in** −) *substract-left-le*: ⟨(*a* :: *nat*) + *b* < *c* ==> *a* <= *c* − *b*⟩
by *auto*

lemma (**in** *conflict-driven-clause-learning*$_W$) *cdcl*$_W$*-stgy-new-learned-in-all-simple-clss*:

assumes
  st: ⟨$cdcl_W$*-stgy*** *R S*⟩ **and**
  invR: ⟨$cdcl_W$*-all-struct-inv R*⟩
shows ⟨*set-mset* (*learned-clss S*) ⊆ *simple-clss* (*atms-of-mm* (*init-clss S*))⟩
proof
  fix *C*
  assume *C*: ⟨*C* ∈# *learned-clss S*⟩
  have *invS*: ⟨$cdcl_W$*-all-struct-inv S*⟩
    using *rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*[*OF st invR*] **.**
  then have *dist*: ⟨*distinct-*$cdcl_W$*-state S*⟩ **and** *alien*: ⟨*no-strange-atm S*⟩
    unfolding $cdcl_W$*-all-struct-inv-def* **by** *fast+*
  have ⟨*atms-of C* ⊆ *atms-of-mm* (*init-clss S*)⟩
    using *alien C* **unfolding** *no-strange-atm-def*
    by (*auto dest*!: *multi-member-split*)
  moreover have ⟨*distinct-mset C*⟩
    using *dist C* **unfolding** *distinct-*$cdcl_W$*-state-def distinct-mset-set-def*
    by (*auto dest*: *in-diffD*)
  moreover have ⟨¬ *tautology C*⟩
    using *invS C* **unfolding** $cdcl_W$*-all-struct-inv-def*
    by (*auto dest*: *in-diffD*)
  ultimately show ⟨*C* ∈ *simple-clss* (*atms-of-mm* (*init-clss S*))⟩
    unfolding *simple-clss-def*
    by *clarify*
qed

lemma (**in** −) *learned-clss-get-all-learned-clss*[*simp*]:
  ⟨*learned-clss* ($state_W$*-of S*) = *get-all-learned-clss S*⟩
  by (*cases S*) (*auto simp*: *learned-clss.simps*)

lemma *cdcl-twl-stgy-restart-new-learned-in-all-simple-clss*:
  assumes
    st: ⟨*cdcl-twl-stgy-restart*** *R S*⟩ **and**
    invR: ⟨*twl-struct-invs* (*fst R*)⟩
  shows ⟨*set-mset* (*clauses* (*get-learned-clss* (*fst S*))) ⊆
     simple-clss (*atms-of-mm* (*get-all-init-clss* (*fst S*)))⟩
proof
  fix *C*
  assume *C*: ⟨*C* ∈# *clauses* (*get-learned-clss* (*fst S*))⟩
  have *invS*: ⟨*twl-struct-invs* (*fst S*)⟩
    using *invR rtranclp-cdcl-twl-stgy-restart-twl-struct-invs st* **by** *blast*
  then have *dist*: ⟨$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state* ($state_W$*-of* (*fst S*))⟩ **and**
      alien: ⟨$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of* (*fst S*))⟩
    unfolding *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* **by** *fast+*
  have ⟨*atms-of C* ⊆ *atms-of-mm* (*get-all-init-clss* (*fst S*))⟩
    using *alien C* **unfolding** $cdcl_W$*-restart-mset.no-strange-atm-def*
    by (*cases S*) (*auto dest*!: *multi-member-split simp*: $cdcl_W$*-restart-mset-state*)
  moreover have ⟨*distinct-mset C*⟩
    using *dist C* **unfolding** $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def distinct-mset-set-def*
    by (*cases S*) (*auto dest*: *in-diffD simp*: $cdcl_W$*-restart-mset-state*)
  moreover have ⟨¬ *tautology C*⟩
    using *invS C* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def twl-struct-invs-def*
    by (*cases S*) (*auto dest*: *in-diffD*)
  ultimately show ⟨*C* ∈ *simple-clss* (*atms-of-mm* (*get-all-init-clss* (*fst S*)))⟩
    unfolding *simple-clss-def*
    by *clarify*
qed

lemma *cdcl-twl-stgy-restart-new*:
  assumes
  ‹*cdcl-twl-stgy-restart S T*› **and**
  ‹*twl-struct-invs* (*fst S*)› **and**
  ‹*distinct-mset* (*get-all-learned-clss* (*fst S*) − *A*)›
 shows ‹*distinct-mset* (*get-all-learned-clss* (*fst T*) − *A*)›
  using *assms*
proof *induction*
  case (*restart-step S T n U*) **note** *st* = *this*(*1*) **and** *res* = *this*(*3*) **and** *invs* = *this*(*4*) **and**
    dist = *this*(*5*)
  have *st*: ‹ *cdcl-twl-stgy*** *S T*›
    using *st* **by** *auto*
  have ‹*get-all-learned-clss U* ⊆# *get-all-learned-clss T*›
    using *res* **by** (*auto simp*: *cdcl-twl-restart.simps*
     image-mset-subseteq-mono[*of* ‹- + -› - *clause*, *unfolded image-mset-union*])
  then have ‹*get-all-learned-clss U* − *A* ⊆#
        learned-clss (*state*$_W$*-of T*) − *A*›
    using *mset-le-subtract* **by** (*cases S*; *cases T*; *cases U*)
      (*auto simp*: *learned-clss.simps ac-simps*
       intro!: *distinct-mset-mono*[*of* ‹*get-all-learned-clss U* − *get-all-learned-clss S*›
        ‹*learned-clss* (*state*$_W$*-of T*) − *learned-clss* (*state*$_W$*-of S*)›])
  moreover {
    have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** (*state*$_W$*-of S*) (*state*$_W$*-of T*)›
      by (*rule rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*[*OF st*]) (*use invs* **in** *simp*)
    then have ‹*distinct-mset* (*learned-clss* (*state*$_W$*-of T*) − *A*)›
      apply (*rule cdcl*$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-distinct-mset-clauses-new-abs*)
      subgoal using *invs* **unfolding** *twl-struct-invs-def fst-conv* **by** *fast*
      subgoal using *invs* **unfolding** *twl-struct-invs-def fst-conv* **by** *fast*
      subgoal using *dist* **by** *simp*
      done
  }
  ultimately show *?case*
    unfolding *fst-conv*
    by (*rule distinct-mset-mono*)
next
  case (*restart-full S T n*) **note** *st* = *this*(*1*) **and** *invs* = *this*(*2*) **and** *dist* = *this*(*3*)
  have *st*: ‹*cdcl-twl-stgy*** *S T*›
    using *st* **unfolding** *full1-def* **by** *fastforce*
  have ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** (*state*$_W$*-of S*) (*state*$_W$*-of T*)›
    by (*rule rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*[*OF st*]) (*use invs* **in** *simp*)
  then have ‹*distinct-mset* (*learned-clss* (*state*$_W$*-of T*) − *A*)›
    apply (*rule cdcl*$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-distinct-mset-clauses-new-abs*)
    subgoal using *invs* **unfolding** *twl-struct-invs-def fst-conv* **by** *fast*
    subgoal using *invs* **unfolding** *twl-struct-invs-def fst-conv* **by** *fast*
    subgoal using *dist* **by** *simp*
    done
  then show *?case*
    by (*cases S*; *cases T*) (*auto simp*: *learned-clss.simps*)
qed

lemma *rtranclp-cdcl-twl-stgy-restart-new-abs*:
  assumes
    ‹*cdcl-twl-stgy-restart*** *S T*› **and**
    ‹*twl-struct-invs* (*fst S*)› **and**
    ‹*distinct-mset* (*get-all-learned-clss* (*fst S*) − *A*)›

shows ‹*distinct-mset* (*get-all-learned-clss* (*fst T*) − *A*)›
using *assms* **apply** (*induction*)
subgoal by *auto*
subgoal by (*auto intro*: *cdcl-twl-stgy-restart-new rtranclp-cdcl-twl-stgy-restart-twl-struct-invs*)
done

end

context *twl-restart*
begin

theorem *wf-cdcl-twl-stgy-restart*:
‹*wf* {(*T*, *S* :: ′*v twl-st* × *nat*). *twl-struct-invs* (*fst S*) ∧ *cdcl-twl-stgy-restart S T*}›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then obtain *g* :: ‹*nat* ⇒ ′*v twl-st* × *nat*› **where**
g: ‹⋀*i*. *cdcl-twl-stgy-restart* (*g i*) (*g* (*Suc i*))› **and**
inv: ‹⋀*i*. *twl-struct-invs* (*fst* (*g i*))›
unfolding *wf-iff-no-infinite-down-chain* **by** *fast*

have *H*: *False* **if** ‹*no-step cdcl-twl-stgy* (*fst* (*g i*))› **for** *i*
using *g*[*of i*] *that*
unfolding *cdcl-twl-stgy-restart.simps*
by (*auto simp*: *full1-def tranclp-unfold-begin*)

have *snd-g*: ‹*snd* (*g i*) = *i* + *snd* (*g 0*)› **for** *i*
apply (*induction i*)
subgoal by *auto*
subgoal for *i*
using *g*[*of i*] *H*[*of* ‹*Suc i*›] **by** (*auto simp*: *cdcl-twl-stgy-restart.simps full1-def*)
done
then have *snd-g-0*: ‹⋀*i*. *i* > *0* ⟹ *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*)

have ‹∃ *h*. *cdcl-twl-stgy*$^{++}$ (*fst* (*g i*)) (*h*) ∧
size (*get-all-learned-clss* (*h*)) > *f* (*snd* (*g i*)) ∧
cdcl-twl-restart (*h*) (*fst* (*g* (*i*+*1*)))›
for *i*
using *g*[*of i*] *H*[*of* ‹*Suc i*›]
unfolding *cdcl-twl-stgy-restart.simps full1-def Suc-eq-plus1*[*symmetric*]
by *force*
then obtain *h* :: ‹*nat* ⇒ ′*v twl-st*› **where**
cdcl-twl: ‹*cdcl-twl-stgy*$^{++}$ (*fst* (*g i*)) (*h i*)› **and**
size-h-g: ‹*size* (*get-all-learned-clss* (*h i*)) > *f* (*snd* (*g i*))› **and**
res: ‹*cdcl-twl-restart* (*h i*) (*fst* (*g* (*i*+*1*)))› **for** *i*
by *metis*

obtain *k* **where**
f-g-k: ‹*f* (*snd* (*g k*)) >
card (*simple-clss* (*atms-of-mm* (*init-clss* ($state_W$-*of* (*h 0*))))) +
size (*get-all-learned-clss* (*fst* (*g 0*)))›
using *not-bounded-nat-exists-larger*[*OF unbounded-f-g*] **by** *blast*

have *cdcl-twl*: ‹*cdcl-twl-stgy*** (*fst* (*g i*)) (*h i*)› **for** *i*
  using *cdcl-twl*[*of i*] **by** *auto*
have *W-g-h*: ‹*cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy*** (*state*$_W$*-of* (*fst* (*g i*))) (*state*$_W$*-of* (*h i*))› **for** *i*
  by (*rule rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*[*OF cdcl-twl*]) (*rule inv*)
have *tranclp-g*: ‹*cdcl-twl-stgy-restart*** (*g 0*) (*g i*)› **for** *i*
  apply (*induction i*)
  subgoal by *auto*
  subgoal for *i* **using** *g*[*of i*] **by** *auto*
  done

have *dist-all-g*:
  ‹*distinct-mset* (*get-all-learned-clss* (*fst* (*g i*)) − *get-all-learned-clss* (*fst* (*g 0*)))›
  for *i*
  apply (*rule rtranclp-cdcl-twl-stgy-restart-new-abs*[*OF tranclp-g*])
  subgoal using *inv* **.**
  subgoal by *simp*
  done

have *dist-h*: ‹*distinct-mset* (*get-all-learned-clss* (*h i*) − *get-all-learned-clss* (*fst* (*g 0*)))›
  (**is** ‹*distinct-mset* (*?U i*)›)
  for *i*
  unfolding *learned-clss-get-all-learned-clss*[*symmetric*]
  apply (*rule cdcl*$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-distinct-mset-clauses-new-abs*[*OF W-g-h*])
  subgoal using *inv*[*of i*] **unfolding** *twl-struct-invs-def* **by** *fast*
  subgoal using *inv*[*of i*] **unfolding** *twl-struct-invs-def* **by** *fast*
  subgoal using *dist-all-g*[*of i*] *distinct-mset-minus*
    unfolding *learned-clss-get-all-learned-clss* **by** *auto*
  done
have *dist-diff*: ‹*distinct-mset* (*c* + (*Ca* + *C*) − *ai*) ⟹
    distinct-mset (*c* − *ai*)› **for** *c Ca C ai*
  by (*metis add-diff-cancel-right′ cancel-ab-semigroup-add-class.diff-right-commute*
    distinct-mset-minus)

have ‹*get-all-learned-clss* (*fst* (*g* (*Suc i*))) ⊆# *get-all-learned-clss* (*h i*)› **for** *i*
  using *res*[*of i*] **by** (*auto simp*: *cdcl-twl-restart.simps*
    image-mset-subseteq-mono[*of* ‹- + -› - *clause*, *unfolded image-mset-union*]
    intro: *mset-le-decr-left1*)

have *h-g*: ‹*init-clss* (*state*$_W$*-of* (*h i*)) = *init-clss* (*state*$_W$*-of* (*fst* (*g i*)))› **for** *i*
  using *cdcl*$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-no-more-init-clss*[*OF W-g-h*[*of i*]] **..**

have *h-g-Suc*: ‹*init-clss* (*state*$_W$*-of* (*h i*)) = *init-clss* (*state*$_W$*-of* (*fst* (*g* (*Suc i*))))› **for** *i*
  using *res*[*of i*] **by** (*auto simp*: *cdcl-twl-restart.simps init-clss.simps*)
have *init-g-0*: ‹*init-clss* (*state*$_W$*-of* (*fst* (*g i*))) = *init-clss* (*state*$_W$*-of* (*fst* (*g 0*)))› **for** *i*
  apply (*induction i*)
  subgoal ..
  subgoal for *j*
    using *h-g*[*of j*] *h-g-Suc*[*of j*] **by** *simp*
  done
then have *K*: ‹*init-clss* (*state*$_W$*-of* (*h i*)) = *init-clss* (*state*$_W$*-of* (*fst* (*g 0*)))› **for** *i*
  using *h-g*[*of i*] **by** *simp*

have *incl*: ‹*set-mset* (*get-all-learned-clss* (*h i*)) ⊆
      simple-clss (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h i*))))› **for** *i*
  unfolding *learned-clss-get-all-learned-clss*[*symmetric*]
  supply [[*unify-trace-failure*]]

apply (*rule cdcl*$_W$*-restart-mset.cdcl*$_W$*-stgy-new-learned-in-all-simple-clss*[*of* ‹*state*$_W$*-of* (*fst* (*g i*))›])
subgoal by (*rule rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*[*OF cdcl-twl*]) (*rule inv*)
subgoal using *inv*[*of i*] **unfolding** *twl-struct-invs-def* **by** *fast*
done
have *incl*: ‹*set-mset* (*get-all-learned-clss* (*h i*)) ⊆
simple-clss (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h i*))))› (**is** ‹*set-mset* (*?V i*) ⊆ -›) **for** *i*
using *incl*[*of i*] **by** (*cases* ‹*h i*›) (*auto dest*: *in-diffD*)

have *incl-init*: ‹*set-mset* (*?U i*) ⊆ *simple-clss* (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h i*))))› **for** *i*
using *incl*[*of i*] **by** (*auto dest*: *in-diffD*)
have *size-U-atms*: ‹*size* (*?U i*) ≤ *card* (*simple-clss* (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h i*)))))› **for** *i*
apply (*subst distinct-mset-size-eq-card*[*OF dist-h*])
apply (*rule card-mono*[*OF - incl-init*])
by (*auto simp*: *simple-clss-finite*)
have *S*:
‹*size* (*?V i*) − *size* (*get-all-learned-clss* (*fst* (*g 0*))) ≤
card (*simple-clss* (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h i*)))))› **for** *i*
apply (*rule order.trans*)
apply (*rule diff-size-le-size-Diff*)
apply (*rule size-U-atms*)
done
have *S*:
‹*size* (*?V i*) ≤
card (*simple-clss* (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h i*))))) +
size (*get-all-learned-clss* (*fst* (*g 0*)))› **for** *i*
using *S*[*of i*] **by** *auto*

have *H*: ‹*card* (*simple-clss* (*atms-of-mm* (*init-clss* (*state*$_W$*-of* (*h k*))))) +
size (*get-all-learned-clss* (*fst* (*g 0*))) > *f* (*k* + *snd* (*g 0*))› **for** *k*
using *S*[*of k*] *size-h-g*[*of k*] **unfolding** *snd-g*[*symmetric*]
by *force*

show *False*
using *H*[*of k*] *f-g-k* **unfolding** *snd-g*[*symmetric*]
unfolding *K*
by *linarith*
qed

end

abbreviation *state*$_W$*-of-restart* **where**
‹*state*$_W$*-of-restart* ≡ (λ(*S*, *n*). (*state*$_W$*-of S*, *n*))›

context *twl-restart-ops*
begin

lemma *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-restart-stgy*:
‹*cdcl-twl-stgy*** *S T* ⟹ *twl-struct-invs S* ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-stgy*** (*state*$_W$*-of S*, *n*) (*state*$_W$*-of T*, *n*)›
using *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-stgy*[*of S T*]
by (*auto dest*: *cdcl*$_W$*-restart-mset.rtranclp-cdcl*$_W$*-restart-stgy-cdcl*$_W$*-restart*
cdcl$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-cdcl*$_W$*-restart-stgy*)

lemma *cdcl-twl-stgy-restart-cdcl*$_W$*-restart-stgy*:
‹*cdcl-twl-stgy-restart S T* ⟹ *twl-struct-invs* (*fst S*) ⟹ *twl-stgy-invs* (*fst S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-stgy*** (*state*$_W$*-of-restart S*) (*state*$_W$*-of-restart T*)›

apply (*induction rule*: *cdcl-twl-stgy-restart.induct*)
subgoal for S T n U
using *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-restart-stgy*[*of* S T n]
cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-stgy.intros* [*of* ‹*state*$_W$*-of-restart*(T, n)›
‹(-, *Suc* n)›]
cdcl$_W$*-restart-mset.rtranclp-cdcl*$_W$*-stgy-cdcl*$_W$*-restart-stgy*[*of* ‹-›
‹*state*$_W$*-of* U› ‹*Suc* n›]
cdcl-twl-restart-cdcl$_W$*-stgy*[*of* T U]
rtranclp-cdcl-twl-stgy-twl-struct-invs[*of* S T]
rtranclp-cdcl-twl-stgy-twl-stgy-invs[*of* S T]
apply (*auto dest*!: *tranclp-into-rtranclp*)
by (*meson r-into-rtranclp rtranclp-trans*)
subgoal for S T n
using *rtranclp-cdcl-twl-stgy-cdcl*$_W$*-restart-stgy*[*of* S T n]
rtranclp-cdcl-twl-stgy-twl-struct-invs[*of* S T]
rtranclp-cdcl-twl-stgy-twl-stgy-invs[*of* S T]
by (*auto dest*!: *tranclp-into-rtranclp simp*: *full1-def*)
done

lemma *rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs*:
assumes
‹*cdcl-twl-stgy-restart*** S T› **and**
‹*twl-struct-invs* (*fst* S)› **and**
‹*twl-stgy-invs* (*fst* S)›
shows ‹*twl-stgy-invs* (*fst* T)›
using *assms*
by (*induction rule*: *rtranclp-induct*)
(*auto intro*: *cdcl-twl-stgy-restart-twl-stgy-invs*
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs)

lemma *rtranclp-cdcl-twl-stgy-restart-cdcl*$_W$*-restart-stgy*:
‹*cdcl-twl-stgy-restart*** S T $\Longrightarrow$ *twl-struct-invs* (*fst* S) $\Longrightarrow$ *twl-stgy-invs* (*fst* S) $\Longrightarrow$
cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-stgy*** (*state*$_W$*-of-restart* S) (*state*$_W$*-of-restart* T)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for T U
using *rtranclp-cdcl-twl-stgy-restart-twl-struct-invs*[*of* S T]
rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs[*of* S T]
cdcl-twl-stgy-restart-cdcl$_W$*-restart-stgy*[*of* T U]
by (*auto dest*!: *tranclp-into-rtranclp*)
done

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-with-leftovers* **where**
‹*cdcl-twl-stgy-restart-with-leftovers* S U $\longleftrightarrow$
($\exists$ T. *cdcl-twl-stgy-restart*** S (T, *snd* U) $\wedge$ *cdcl-twl-stgy*** T (*fst* U))›

lemma *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart*:
‹*cdcl-twl-stgy-restart* (T, m) (V, *Suc* m) $\Longrightarrow$
cdcl-twl-stgy** S T $\Longrightarrow$ *cdcl-twl-stgy-restart* (S, m) (V, *Suc* m)›
by (*subst* (*asm*) *cdcl-twl-stgy-restart.simps*)
(*auto simp*: *intro*: *cdcl-twl-stgy-restart.intros*
dest: *rtranclp-tranclp-tranclp*)

lemma *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*:

```
  ‹cdcl-twl-stgy-restart (T, m) (V, m) ⟹
      cdcl-twl-stgy** S T ⟹ cdcl-twl-stgy-restart (S, m) (V, m)›
  by (subst (asm) cdcl-twl-stgy-restart.simps)
    (auto simp: intro: cdcl-twl-stgy-restart.intros
      dest: rtranclp-tranclp-tranclp rtranclp-full1I)


definition cdcl-twl-stgy-restart-with-leftovers1 where
  ‹cdcl-twl-stgy-restart-with-leftovers1 S U ⟷
     cdcl-twl-stgy-restart S U ∨
     (cdcl-twl-stgy++ (fst S) (fst U) ∧ snd S = snd U)›

lemma (in twl-restart) wf-cdcl-twl-stgy-restart-with-leftovers1:
  ‹wf {(T :: 'v twl-st × nat, S).
        twl-struct-invs (fst S) ∧ cdcl-twl-stgy-restart-with-leftovers1 S T}›
  (is ‹wf ?S›)
proof (rule ccontr)
  assume ‹¬ ?thesis›
  then obtain g :: ‹nat ⇒ 'v twl-st × nat› where
    g: ‹⋀i. cdcl-twl-stgy-restart-with-leftovers1 (g i) (g (Suc i))› and
    inv: ‹⋀i. twl-struct-invs (fst (g i))›
    unfolding wf-iff-no-infinite-down-chain by fast
  have ns: ‹¬no-step cdcl-twl-stgy (fst (g i))› for i
    using g[of i]
    by (fastforce simp: cdcl-twl-stgy-restart-with-leftovers1-def
        cdcl-twl-stgy-restart.simps full1-def tranclp-unfold-begin)

  define h where
    ‹h ≡ rec-nat (g 0) (λi Si. g (SOME k. cdcl-twl-stgy-restart Si (g k)))›
  have [simp]: ‹h 0 = g 0›
    unfolding h-def by auto

  have L: ‹cdcl-twl-stgy++ (fst (g i)) (fst (g (Suc (i + k)))) ∧
        cdcl-twl-stgy++ (fst (g (i + k))) (fst (g (Suc (i + k)))) ∧
        snd (g (Suc (i + k))) = snd (g i)›
    if ‹k < j› and
      H: ‹⋀k. k ≤ j ⟹ ¬cdcl-twl-stgy-restart (g i) (g (Suc i + k))›
    for k i j
    using that
  proof (induction j arbitrary: k)
    case 0
    then show ?case by auto
  next
    case (Suc j k)
    then have
      IH: ‹⋀k. k < j ⟹
        cdcl-twl-stgy++ (fst (g i)) (fst (g (Suc (i + k)))) ∧
        cdcl-twl-stgy++ (fst (g (i + k))) (fst (g (Suc (i + k)))) ∧
        snd (g (Suc (i + k))) = snd (g i)› and
      ‹k < Suc j› and
      H: ‹⋀k. k ≤ Suc j ⟹ ¬ cdcl-twl-stgy-restart (g i) (g (Suc i + k))›
      by auto
    show ?case
    proof (cases ‹k = j›)
      case False
      then show ?thesis
```

using *IH*[*of k*] ‹*k* < *Suc j*› **by** *simp*
next
case [*simp*]: *True*
consider
(*res*) ‹*cdcl-twl-stgy-restart* (*g* ((*i*+*k*))) (*g* ((*Suc* (*i*+*k*))))› |
(*stgy*) ‹*cdcl-twl-stgy*$^{++}$ (*fst* (*g* ((*i*+*k*)))) (*fst* (*g* ((*Suc* (*i*+*k*)))))› **and**
‹*snd* (*g* (*Suc* (*i* + *k*))) = *snd* (*g* (*i* + *k*))›
using *g*[*of* ‹*i*+*k*›] **unfolding** *cdcl-twl-stgy-restart-with-leftovers1-def*
by *auto*
then show *?thesis*
proof *cases*
case *stgy*
then show *?thesis*
using *IH*[*of* ‹*k* − *1*›]
by (*cases* ‹*0* < *j*›) *auto*
next
case *res*
have *Sucg*: ‹*Suc* (*snd* (*g* ((*i* + *k*)))) = *snd* (*g* (*Suc* ((*i* + *k*))))›
using *res*
ns[*of* ‹*Suc* ((*i* + *k*))›]
by (*auto simp*: *cdcl-twl-stgy-restart.simps full1-def*)

have ‹*cdcl-twl-stgy-restart* (*g i*) (*g* (*Suc* (*i* + *k*)))›
using *IH*[*of* ‹*k* − *1*›]
res cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart[*of* ‹*fst* (*g* ((*i* + *k*)))›
‹*snd* (*g* ((*i* + *k*)))› ‹*fst* (*g* (*Suc* ((*i* + *k*))))› ‹*fst* (*g i*)›]
unfolding *Sucg prod.collapse*
by (*cases* ‹*0* < *j*›) (*auto intro*!: *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart*
dest!: *tranclp-into-rtranclp*)
then show *?thesis*
using *H*[*of k*] ‹*k* < *Suc j*›
by *auto*
qed
qed
qed
have *Ex-cdcl-twl-stgy-restart*: ‹∃ *k* > *i*. *cdcl-twl-stgy-restart* (*g i*) (*g k*)› **for** *i*
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have *H*: ‹⋀*k*. *k* > *i* ⟹ ¬*cdcl-twl-stgy-restart* (*g i*) (*g k*)›
by *fast*

define *g'* **where**
‹*g'* = (λ*k*. *fst* (*g* (*k* + *i*)))›
have *L*: ‹*cdcl-twl-stgy*$^{++}$ (*fst* (*g i*)) (*fst* (*g* (*Suc* (*i* + *k*)))) ∧
cdcl-twl-stgy^{++} (*fst* (*g* (*i* + *k*))) (*fst* (*g* (*Suc* (*i* + *k*)))) ∧
snd (*g* (*Suc* (*i* + *k*))) = *snd* (*g i*)›
for *k*
using *L*[*of k* ‹*k*+*1*› *i*] *H*[*of* ‹*Suc i*+-›]
by *auto*
have ‹(*g'* (*Suc k*), *g' k*) ∈ {(*T*, *S*). *twl-struct-invs S* ∧ *cdcl-twl-stgy*$^{++}$ *S T*}› **for** *k*
using *L*[*of k*] *inv*
unfolding *g'-def*
by (*auto simp*: *ac-simps*)
then show *False*
using *tranclp-wf-cdcl-twl-stgy*
unfolding *wf-iff-no-infinite-down-chain*

by *fast*
qed
have *h-Suc*: ‹h (Suc i) = g ($SOME$ j. *cdcl-twl-stgy-restart* (h i) (g j))› **for** i
unfolding *h-def* **by** *auto*
have *h-g*: ‹$\exists j$. h i = g j› **for** i
proof (*induction* i)
case *0*
then show *?case* **by** *auto*
next
case (*Suc* i)
then obtain i' **where**
i': ‹h i = g i'›
by *blast*
define j **where** ‹$j \equiv$ $SOME$ j. *cdcl-twl-stgy-restart* (h i) (g j)›
obtain k **where**
k: ‹$k > i'$› **and**
i-k: ‹*cdcl-twl-stgy-restart* (g i') (g k)›
using *Ex-cdcl-twl-stgy-restart*[*of* i'] **by** *blast*
have ‹*cdcl-twl-stgy-restart* (h i) (g j)›
unfolding *j-def*
apply (*rule someI*[*of* ‹λS. *cdcl-twl-stgy-restart* (h i) (g S)›])
using k *i-k* **unfolding** i' **by** *fast*
then show *?case*
unfolding *h-Suc* **by** *auto*
qed

have ‹*cdcl-twl-stgy-restart* (h i) (h (Suc i))› **for** i
proof −
obtain i' **where**
h-g-i: ‹h i = g i'›
using *h-g* **by** *fast*
define j **where** ‹$j \equiv$ $SOME$ j. *cdcl-twl-stgy-restart* (h i) (g j)›
obtain k **where**
k: ‹$k > i'$› **and**
i-k: ‹*cdcl-twl-stgy-restart* (g i') (g k)›
using *Ex-cdcl-twl-stgy-restart*[*of* i'] **by** *blast*
have ‹*cdcl-twl-stgy-restart* (h i) (g j)›
unfolding *j-def*
apply (*rule someI*[*of* - k])
using k *i-k* **unfolding** *h-g-i* **by** *fast*
then show *?thesis*
unfolding *h-Suc j-def*[*symmetric*] **.**
qed
moreover have ‹$\bigwedge i$. *twl-struct-invs* (*fst* (h i))›
using *inv h-g* **by** *metis*
ultimately show *False*
using *wf-cdcl-twl-stgy-restart*
unfolding *wf-iff-no-infinite-down-chain* **by** *fast*
qed

lemma (**in** *twl-restart*) *wf-cdcl-twl-stgy-restart-measure*:
‹*wf* ({((*brkT*, T, n), *brkS*, S, m).
twl-struct-invs S ∧ *cdcl-twl-stgy-restart-with-leftovers1* (S, m) (T, n)} ∪
{((*brkT*, T), *brkS*, S). S = T ∧ *brkT* ∧ ¬ *brkS*})›
(**is** ‹*wf* (*?TWL* ∪ *?BOOL*)›)

proof (*rule wf-union-compatible*)
  show ‹*wf ?TWL*›
    apply (*rule wf-subset*)
     apply (*rule wf-snd-wf-pair*[*OF wf-cdcl-twl-stgy-restart-with-leftovers1*])
    by *auto*
  show ‹*?TWL O ?BOOL* ⊆ *?TWL*›
    by *auto*

  show ‹*wf ?BOOL*›
    unfolding *wf-iff-no-infinite-down-chain*
  proof *clarify*
    fix *f* :: ‹*nat* ⇒ *bool* × *'b*›
    assume *H*: ‹∀ *i*. (*f* (*Suc i*), *f i*) ∈ {((*brkT*, *T*), *brkS*, *S*). *S* = *T* ∧ *brkT* ∧ ¬ *brkS*}›
    then have ‹(*f* (*Suc 0*), *f 0*) ∈ {((*brkT*, *T*), *brkS*, *S*). *S* = *T* ∧ *brkT* ∧ ¬ *brkS*}› **and**
      ‹(*f* (*Suc 1*), *f 1*) ∈ {((*brkT*, *T*), *brkS*, *S*). *S* = *T* ∧ *brkT* ∧ ¬ *brkS*}›
      by *presburger+*
    then show *False*
      by *auto*
  qed
qed

lemma (**in** *twl-restart*) *wf-cdcl-twl-stgy-restart-measure-early*:
  ‹*wf* ({((*ebrk*, *brkT*, *T*, *n*), *ebrk*, *brkS*, *S*, *m*).
        twl-struct-invs S ∧ *cdcl-twl-stgy-restart-with-leftovers1* (*S*, *m*) (*T*, *n*)} ∪
       {((*ebrkT*, *brkT*, *T*), (*ebrkS*, *brkS*, *S*)). *S* = *T* ∧ (*ebrkT* ∨ *brkT*) ∧ (¬*brkS* ∧ ¬*ebrkS*)})›
  (**is** ‹*wf* (*?TWL* ∪ *?BOOL*)›)
proof (*rule wf-union-compatible*)
  show ‹*wf ?TWL*›
    apply (*rule wf-subset*)
    apply (*rule wf-snd-wf-pair*)
     apply (*rule wf-snd-wf-pair*[*OF wf-cdcl-twl-stgy-restart-with-leftovers1*])
    by *auto*
  show ‹*?TWL O ?BOOL* ⊆ *?TWL*›
    by *auto*

  show ‹*wf ?BOOL*›
    unfolding *wf-iff-no-infinite-down-chain*
  proof *clarify*
    fix *f* :: ‹*nat* ⇒ *bool* × *bool* × *-*›
    assume *H*: ‹∀ *i*. (*f* (*Suc i*), *f i*) ∈ *?BOOL*›
    then have ‹(*f* (*Suc 0*), *f 0*) ∈ *?BOOL*› **and**
      ‹(*f* (*Suc 1*), *f 1*) ∈ *?BOOL*›
      by *presburger+*
    then show *False*
      by *auto*
  qed
qed

lemma *cdcl-twl-stgy-restart-with-leftovers-cdcl*$_W$*-restart-stgy*:
  ‹*cdcl-twl-stgy-restart-with-leftovers S T* ⟹ *twl-struct-invs* (*fst S*) ⟹ *twl-stgy-invs* (*fst S*) ⟹
    cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-stgy*** (*state*$_W$*-of-restart S*) (*state*$_W$*-of-restart T*)›
  unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
  apply (*rule exE*)
   apply *assumption*
  subgoal for *S'*

using
rtranclp-cdcl-twl-stgy-restart-cdcl$_W$*-restart-stgy*[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-cdcl$_W$*-restart-stgy*[*of S'* ‹*fst T*› ‹*snd T*›]
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs[*of S* ‹(*S'*, *snd T*)›]
by (*cases T*) *auto*
done

lemma *cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs*:
‹*cdcl-twl-stgy-restart-with-leftovers S T* $\Longrightarrow$ *twl-struct-invs* (*fst S*) $\Longrightarrow$
twl-struct-invs (*fst T*)›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
apply (*rule exE*)
apply *assumption*
subgoal for *S'*
using
rtranclp-cdcl-twl-stgy-twl-struct-invs[*of* ‹*S'*› ‹(*fst T*)›]
rtranclp-cdcl-twl-stgy-restart-cdcl$_W$*-restart-stgy*[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-cdcl$_W$*-restart-stgy*[*of S'* ‹*fst T*› ‹*snd T*›]
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs[*of S* ‹(*S'*, *snd T*)›]
by *auto*
done

lemma *rtranclp-cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs*:
‹*cdcl-twl-stgy-restart-with-leftovers*** *S T* $\Longrightarrow$ *twl-struct-invs* (*fst S*) $\Longrightarrow$
twl-struct-invs (*fst T*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs*[*of T U*]
by *auto*
done

lemma *cdcl-twl-stgy-restart-with-leftovers-twl-stgy-invs*:
‹*cdcl-twl-stgy-restart-with-leftovers S T* $\Longrightarrow$ *twl-struct-invs* (*fst S*) $\Longrightarrow$
twl-stgy-invs (*fst S*) $\Longrightarrow$ *twl-stgy-invs* (*fst T*)›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
apply (*rule exE*)
apply *assumption*
subgoal for *S'*
using
rtranclp-cdcl-twl-stgy-twl-struct-invs[*of* ‹*S'*› ‹(*fst T*)›]
rtranclp-cdcl-twl-stgy-twl-stgy-invs[*of* ‹*S'*› ‹(*fst T*)›]
rtranclp-cdcl-twl-stgy-restart-cdcl$_W$*-restart-stgy*[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-cdcl$_W$*-restart-stgy*[*of S'* ‹*fst T*› ‹*snd T*›]
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-struct-invs[*of S* ‹(*S'*, *snd T*)›]
rtranclp-cdcl-twl-stgy-restart-twl-stgy-invs[*of S* ‹(*S'*, *snd T*)›]
by *auto*
done

lemma *rtranclp-cdcl-twl-stgy-restart-with-leftovers-twl-stgy-invs*:

⟨*cdcl-twl-stgy-restart-with-leftovers*** *S T* ⟹ *twl-struct-invs* (*fst S*) ⟹
twl-stgy-invs (*fst S*) ⟹ *twl-stgy-invs* (*fst T*)⟩
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-stgy-restart-with-leftovers-twl-stgy-invs*[*of T U*]
rtranclp-cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs[*of S T*]
by *auto*
done

lemma *rtranclp-cdcl-twl-stgy-restart-with-leftovers-cdcl*$_W$*-restart-stgy*:
⟨*cdcl-twl-stgy-restart-with-leftovers*** *S T* ⟹ *twl-struct-invs* (*fst S*) ⟹ *twl-stgy-invs* (*fst S*) ⟹
cdcl$_W$*-restart-mset.cdcl*$_W$*-restart-stgy*** (*state*$_W$*-of-restart S*) (*state*$_W$*-of-restart T*)⟩
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *T U*
using *cdcl-twl-stgy-restart-with-leftovers-cdcl*$_W$*-restart-stgy*[*of T U*]
rtranclp-cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs[*of S T*]
rtranclp-cdcl-twl-stgy-restart-with-leftovers-twl-stgy-invs[*of S T*]
by *auto*
done

end

end
theory *Watched-Literals-Algorithm-Restart*
imports *Watched-Literals-Algorithm Watched-Literals-Transition-System-Restart*
begin

context *twl-restart-ops*
begin

Restarts are never necessary

definition *restart-required* :: *'v twl-st* ⇒ *nat* ⇒ *bool nres* **where**
⟨*restart-required S n* = *SPEC* (λ*b*. *b* ⟶ *size* (*get-learned-clss S*) > *f n*)⟩

definition (**in** −) *restart-prog-pre* :: ⟨*'v twl-st* ⇒ *bool* ⇒ *bool*⟩ **where**
⟨*restart-prog-pre S brk* ⟷ *twl-struct-invs S* ∧ *twl-stgy-invs S* ∧
(¬*brk* ⟶ *get-conflict S* = *None*)⟩

definition *restart-prog*
:: *'v twl-st* ⇒ *nat* ⇒ *bool* ⇒ (*'v twl-st* × *nat*) *nres*
where
⟨*restart-prog S n brk* = *do* {
ASSERT(*restart-prog-pre S brk*);
b ← *restart-required S n*;
b2 ← *SPEC*(λ-. *True*);
if b2 ∧ *b* ∧ ¬*brk then do* {
T ← *SPEC*(λ*T*. *cdcl-twl-restart S T*);
RETURN (*T*, *n* + *1*)
}
else
if b ∧ ¬*brk then do* {
T ← *SPEC*(λ*T*. *cdcl-twl-restart S T*);
RETURN (*T*, *n* + *1*)
}

else
  RETURN $(S,\ n)$
}⟩

definition *cdcl-twl-stgy-restart-prog-inv* **where**
⟨*cdcl-twl-stgy-restart-prog-inv* S_0 *brk T n* ≡ *twl-struct-invs T* ∧ *twl-stgy-invs T* ∧
(*brk* ⟶ *final-twl-state T*) ∧ *cdcl-twl-stgy-restart-with-leftovers* (S_0, *0*) (*T*, *n*) ∧
clauses-to-update T = {#} ∧ (¬*brk* ⟶ *get-conflict T* = *None*)⟩

definition *cdcl-twl-stgy-restart-prog* :: *'v twl-st* ⇒ *'v twl-st nres* **where**
⟨*cdcl-twl-stgy-restart-prog* S_0 =
do {
  (*brk*, *T*, -) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S_0\ brk\ T\ n}$
    (λ(*brk*, -). ¬*brk*)
    (λ(*brk*, *S*, *n*).
    do {
      T ← *unit-propagation-outer-loop S*;
      (*brk*, *T*) ← *cdcl-twl-o-prog T*;
      (*T*, *n*) ← *restart-prog T n brk*;
      RETURN (*brk*, *T*, *n*)
    })
    (*False*, S_0, *0*);
  RETURN T
}⟩

lemma (**in** *twl-restart*)
assumes
inv: ⟨*case* (*brk*, *T*, *m*) *of* (*brk*, *T*, *m*) ⇒ *cdcl-twl-stgy-restart-prog-inv S brk T m*⟩ **and**
cond: ⟨*case* (*brk*, *T*, *m*) *of* (*brk*, *uu-*) ⇒ ¬ *brk*⟩ **and**
other-inv: ⟨*cdcl-twl-o-prog-spec S′* (*brk′*, *U*)⟩ **and**
struct-invs-S: ⟨*twl-struct-invs S′*⟩ **and**
cp: ⟨$cdcl\text{-}twl\text{-}cp^{**}$ *T S′*⟩ **and**
lits-to-update: ⟨*literals-to-update S′* = {#}⟩ **and**
⟨∀ *S′a*. ¬ *cdcl-twl-cp S′ S′a*⟩ **and**
⟨*twl-stgy-invs S′*⟩
shows *restart-prog-spec*:
⟨*restart-prog U m brk′*
  ≤ *SPEC*
    (λ*x*. (*case x of*
      (*T*, *na*) ⇒ *RETURN* (*brk′*, *T*, *na*))
     ≤ *SPEC*
      (λ*s′*. (*case s′ of*
(*brk*, *T*, *n*) ⇒
twl-struct-invs T ∧
twl-stgy-invs T ∧
(*brk* ⟶ *final-twl-state T*) ∧
cdcl-twl-stgy-restart-with-leftovers (*S*, *0*)
(*T*, *n*) ∧
clauses-to-update T = {#} ∧
(¬ *brk* ⟶ *get-conflict T* = *None*)) ∧
(*s′*, *brk*, *T*, *m*)
∈ {((*brkT*, *T*, *n*), *brkS*, *S*, *m*).
twl-struct-invs S ∧
cdcl-twl-stgy-restart-with-leftovers1 (*S*, *m*)
(*T*, *n*)} ∪
{((*brkT*, *T*), *brkS*, *S*). *S* = *T* ∧ *brkT* ∧ ¬ *brkS*}))⟩ (**is** *?A*)

proof −
  have *struct-invs'*: ‹*cdcl-twl-restart U T* $\Longrightarrow$ *twl-struct-invs T*› **for** *T*
    using *assms*(*3*) *cdcl-twl-restart-twl-struct-invs* **by** *blast*
  have *stgy-invs*: ‹*cdcl-twl-restart U V* $\Longrightarrow$*twl-stgy-invs V*› **for** *V*
    using *assms*(*3*) *cdcl-twl-restart-twl-stgy-invs* **by** *blast*
  have *res-no-clss-to-upd*: ‹*cdcl-twl-restart U V* $\Longrightarrow$ *clauses-to-update V* = {#}› **for** *V*
    by (*auto simp*: *cdcl-twl-restart.simps*)
  have *res-no-confl*: ‹*cdcl-twl-restart U V* $\Longrightarrow$ *get-conflict V* = *None*› **for** *V*
    by (*auto simp*: *cdcl-twl-restart.simps*)

  have
    struct-invs-T: ‹*twl-struct-invs T*› **and**
    ‹*twl-stgy-invs T*› **and**
    ‹*brk* $\longrightarrow$ *final-twl-state T*› **and**
    twl-res: ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *m*)› **and**
    ‹*clauses-to-update T* = {#}› **and**
    confl: ‹¬ *brk* $\longrightarrow$ *get-conflict T* = *None*›
    using *inv* **unfolding** *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast+*
  have
    cdcl-o: ‹*cdcl-twl-o*** *S' U*› **and**
    conflict-U: ‹*get-conflict U* ≠ *None* $\Longrightarrow$ *count-decided* (*get-trail U*) = *0*› **and**
    brk'-no-step: ‹*brk'* $\Longrightarrow$ *final-twl-state U*› **and**
    struct-invs-U: ‹*twl-struct-invs U*› **and**
    stgy-invs-U: ‹*twl-stgy-invs U*› **and**
    clss-to-upd-U: ‹*clauses-to-update U* = {#}› **and**
    lits-to-upd-U: ‹¬ *brk'* $\longrightarrow$ *literals-to-update U* ≠ {#}› **and**
    confl-U: ‹¬ *brk'* $\longrightarrow$ *get-conflict U* = *None*›
    using *other-inv* **unfolding** *final-twl-state-def* **by** *fast+*

  have ‹*cdcl-twl-stgy*** *T U*›
    by (*meson* ‹*cdcl-twl-o*** *S' U*› *assms*(*5*) *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD*
        rtranclp-trans)
  have
    twl-restart-after-restart:
      ‹*cdcl-twl-stgy-restart* (*T*, *m*) (*V*, *Suc m*)› **and**
    rtranclp-twl-restart-after-restart:
      ‹*cdcl-twl-stgy-restart*** (*S*, *0*) (*V*, *m* + *1*)› **and**
    cdcl-twl-stgy-restart-with-leftovers-after-restart:
      ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*V*, *m* + *1*)› **and**
    cdcl-twl-stgy-restart-with-leftovers-after-restart-T-V:
      ‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*V*, *Suc m*)› **and**
    cdcl-twl-stgy-restart-with-leftovers1-after-restart:
      ‹*cdcl-twl-stgy-restart-with-leftovers1* (*T*, *m*) (*V*, *Suc m*)›
    if
      f: ‹*True* $\longrightarrow$ *f* (*snd* (*U*, *m*)) < *size* (*get-learned-clss* (*fst* (*U*, *m*)))› **and**
      res: ‹*cdcl-twl-restart U V*› **and**
      [*simp*]: ‹*brk'* = *False*›
    for *V*
  proof −
    have ‹*S'* ≠ *U*›
      using *lits-to-update lits-to-upd-U* **by** *auto*
    then have ‹*cdcl-twl-o*$^{++}$ *S' U*›
      using ‹*cdcl-twl-o*** *S' U*› **unfolding** *rtranclp-unfold* **by** *auto*
    then have *st*: ‹*cdcl-twl-stgy*$^{++}$ *T U*›
      by (*meson local.cp rtranclp-cdcl-twl-cp-stgyD rtranclp-tranclp-tranclp*
          tranclp-cdcl-twl-o-stgyD)

show *twl-res-T-V*: ‹*cdcl-twl-stgy-restart* (*T*, *m*) (*V*, *Suc m*)›
apply (*rule cdcl-twl-stgy-restart.restart-step*[*of - U*])
subgoal by (*rule st*)
subgoal using *f* **by** *simp*
subgoal by (*rule res*)
done
show ‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*V*, *Suc m*)›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
by (*rule exI*[*of -* ‹*V*›])(*auto simp*: *twl-res-T-V*)
show ‹$cdcl\text{-}twl\text{-}stgy\text{-}restart^{**}$ (*S*, *0*) (*V*, *m* + *1*)›
using *twl-res twl-res-T-V*
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
by (*auto dest*: *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart*)
then show ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*V*, *m* + *1*)›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def* **apply** −
by (*rule exI*[*of - V*]) *auto*
show ‹*cdcl-twl-stgy-restart-with-leftovers1* (*T*, *m*) (*V*, *Suc m*)›
using *twl-res-T-V*
unfolding *cdcl-twl-stgy-restart-with-leftovers1-def*
by *fast*
qed
have
rtranclp-twl-restart-after-restart-S-U:
‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)› **and**
rtranclp-twl-restart-after-restart-T-U:
‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*U*, *m*)›
proof −
obtain *Ta* **where**
S-Ta: ‹$cdcl\text{-}twl\text{-}stgy\text{-}restart^{**}$ (*S*, *0*) (*Ta*, *snd* (*T*, *m*))›
‹$cdcl\text{-}twl\text{-}stgy^{**}$ *Ta* (*fst* (*T*, *m*))›
using *twl-res* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
by *auto*
then have ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *Ta* (*fst* (*U*, *m*))›
using ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *T U*› **by** *auto*
then show ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)›
using *S-Ta* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
by *fastforce*
show ‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*U*, *m*)›
using ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *T U*› **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
by *fastforce*
qed
have
rtranclp-twl-restart-after-restart-brk:
‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)›
if
[*simp*]: ‹$brk' = True$›
proof −
have ‹*full1 cdcl-twl-stgy T U* ∨ *T* = *U* ∨ *get-conflict U* ≠ *None*›
using *brk'-no-step* ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *T U*›
unfolding *rtranclp-unfold full1-def final-twl-state-def* **by** *auto*
then consider
(*step*) ‹*cdcl-twl-stgy-restart* (*T*, *m*) (*U*, *m*)› |
(*TU*) ‹*T* = *U*› |
(*final*) ‹*get-conflict U* ≠ *None*›
by (*auto dest*!: *cdcl-twl-stgy-restart.intros*)

then show ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)›
proof *cases*
case *step*
then show *?thesis*
using *twl-res* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
using *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*[*of T m U*] **apply** −
by (*rule exI*[*of - U*]) (*fastforce dest*!:)
next
case [*simp*]: *TU*
then show *?thesis*
using *twl-res* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
using *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*[*of T m U*] **apply** −
by *fastforce*
next
case *final*
then show *?thesis*
using *twl-res* ‹*cdcl-twl-stgy*** *T U*› **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
using *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*[*of T m U*] **apply** −
by *fastforce*
qed
qed
have *cdcl-twl-stgy-restart-with-leftovers1-T-U*:
‹*cdcl-twl-stgy-restart-with-leftovers1* (*T*, *m*) (*U*, *m*) ⟷ *T* ≠ *U*›
proof −
have ‹*cdcl-twl-stgy*$^{++}$ *T U* ∨ *T* = *U*›
using ‹*cdcl-twl-stgy*** *T U*› **unfolding** *rtranclp-unfold* **by** *auto*
then show *?thesis*
using *wf-not-refl*[*OF wf-cdcl-twl-stgy-restart*, *of* ‹(*U*, *m*)›]
using *wf-not-refl*[*OF tranclp-wf-cdcl-twl-stgy*, *of* ‹*U*›]
struct-invs-U
unfolding *cdcl-twl-stgy-restart-with-leftovers1-def* **by** *auto*
qed
have *brk'-eq*: ‹¬*cdcl-twl-stgy-restart-with-leftovers1* (*T*, *m*) (*U*, *m*) ⟹ *brk'*›
using *cdcl-o lits-to-upd-U lits-to-update local.cp*
unfolding *cdcl-twl-stgy-restart-with-leftovers1-def*
unfolding *rtranclp-unfold*
by (*auto dest*!: *tranclp-cdcl-twl-o-stgyD tranclp-cdcl-twl-cp-stgyD*
simp: *rtranclp-unfold*
dest: *rtranclp-tranclp-tranclp tranclp-trans*)

have [*simp*]: ‹*brk* = *False*›
using *cond* **by** *auto*
show *?A*
unfolding *restart-prog-def restart-required-def*
apply (*refine-vcg*; *remove-dummy-vars*)
subgoal using *struct-invs-U stgy-invs-U confl-U* **unfolding** *restart-prog-pre-def* **by** *fast*
subgoal by (*rule struct-invs'*)
subgoal by (*rule stgy-invs*)
subgoal by (*rule cdcl-twl-stgy-restart-with-leftovers-after-restart*) *simp*
subgoal by (*rule res-no-clss-to-upd*)
subgoal by (*rule res-no-confl*)
subgoal by (*auto intro*!: *struct-invs-S struct-invs-T*
cdcl-twl-stgy-restart-with-leftovers1-after-restart)
subgoal using *struct-invs'* **by** *blast*
subgoal using *stgy-invs* **by** *blast*
subgoal by (*rule cdcl-twl-stgy-restart-with-leftovers-after-restart*) *simp*

subgoal by (*rule res-no-clss-to-upd*)
subgoal by (*rule res-no-confl*)
subgoal by (*auto intro*!: *struct-invs-S struct-invs-T*
cdcl-twl-stgy-restart-with-leftovers1-after-restart)
subgoal by (*rule struct-invs-U*)
subgoal by (*rule stgy-invs-U*)
subgoal by (*rule brk′-no-step*) *simp*
subgoal
by (*auto intro*: *rtranclp-twl-restart-after-restart-brk*
rtranclp-twl-restart-after-restart-S-U)
subgoal by (*rule clss-to-upd-U*)
subgoal using *struct-invs-U conflict-U lits-to-upd-U*
by (*cases* ‹*get-conflict U*›)(*auto simp*: *twl-struct-invs-def*)
subgoal
using *cdcl-twl-stgy-restart-with-leftovers1-T-U brk′-eq*
by (*auto simp*: *twl-restart-after-restart struct-invs-S struct-invs-T*
cdcl-twl-stgy-restart-with-leftovers-after-restart-T-V struct-invs-U
rtranclp-twl-restart-after-restart-brk rtranclp-twl-restart-after-restart-T-U
cdcl-twl-stgy-restart-with-leftovers1-after-restart)
done
qed

lemma (**in** *twl-restart*)
assumes
inv: ‹*case* (*ebrk*, *brk*, *T*, *m*) *of* (*ebrk*, *brk*, *T*, *m*) ⇒ *cdcl-twl-stgy-restart-prog-inv S brk T m*› **and**
cond: ‹*case* (*ebrk*, *brk*, *T*, *m*) *of* (*ebrk*, *brk*, -) ⇒ ¬ *brk* ∧ ¬*ebrk*› **and**
other-inv: ‹*cdcl-twl-o-prog-spec S′* (*brk′*, *U*)› **and**
struct-invs-S: ‹*twl-struct-invs S′*› **and**
cp: ‹*cdcl-twl-cp*** *T S′*› **and**
lits-to-update: ‹*literals-to-update S′* = {#}› **and**
‹∀ *S′a*. ¬ *cdcl-twl-cp S′ S′a*› **and**
‹*twl-stgy-invs S′*›
shows *restart-prog-early-spec*:
‹*restart-prog U m brk′*
≤ *SPEC*
(λ*x*. (*case x of* (*T*, *n*) ⇒ *RES UNIV* ≫= (λ*ebrk*. *RETURN* (*ebrk*, *brk′*, *T*, *n*)))
≤ *SPEC*
(λ*s′*. (*case s′ of* (*ebrk*, *brk*, *x*, *xb*) ⇒
cdcl-twl-stgy-restart-prog-inv S brk x xb) ∧
(*s′*, *ebrk*, *brk*, *T*, *m*)
∈ {((*ebrk*, *brkT*, *T*, *n*), *ebrk*, *brkS*, *S*, *m*).
twl-struct-invs S ∧
cdcl-twl-stgy-restart-with-leftovers1 (*S*, *m*) (*T*, *n*)} ∪
{((*ebrkT*, *brkT*, *T*), *ebrkS*, *brkS*, *S*).
S = *T* ∧ (*ebrkT* ∨ *brkT*) ∧ ¬ *brkS* ∧ ¬ *ebrkS*}))› (**is** ‹*?B*›)
proof −
have *struct-invs′*: ‹*cdcl-twl-restart U T* ⟹ *twl-struct-invs T*› **for** *T*
using *assms*(*3*) *cdcl-twl-restart-twl-struct-invs* **by** *blast*
have *stgy-invs*: ‹*cdcl-twl-restart U V* ⟹*twl-stgy-invs V*› **for** *V*
using *assms*(*3*) *cdcl-twl-restart-twl-stgy-invs* **by** *blast*
have *res-no-clss-to-upd*: ‹*cdcl-twl-restart U V* ⟹ *clauses-to-update V* = {#}› **for** *V*
by (*auto simp*: *cdcl-twl-restart.simps*)
have *res-no-confl*: ‹*cdcl-twl-restart U V* ⟹ *get-conflict V* = *None*› **for** *V*
by (*auto simp*: *cdcl-twl-restart.simps*)

have

struct-invs-T: ‹*twl-struct-invs T*› **and**
‹*twl-stgy-invs T*› **and**
‹*brk* ⟶ *final-twl-state T*› **and**
twl-res: ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *m*)› **and**
‹*clauses-to-update T* = {#}› **and**
confl: ‹¬ *brk* ⟶ *get-conflict T* = *None*›
using *inv* **unfolding** *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast+*
have
cdcl-o: ‹*cdcl-twl-o*** *S′ U*› **and**
conflict-U: ‹*get-conflict U* ≠ *None* ⟹ *count-decided* (*get-trail U*) = *0*› **and**
brk′-no-step: ‹*brk′* ⟹ *final-twl-state U*› **and**
struct-invs-U: ‹*twl-struct-invs U*› **and**
stgy-invs-U: ‹*twl-stgy-invs U*› **and**
clss-to-upd-U: ‹*clauses-to-update U* = {#}› **and**
lits-to-upd-U: ‹¬ *brk′* ⟶ *literals-to-update U* ≠ {#}› **and**
confl-U: ‹¬ *brk′* ⟶ *get-conflict U* = *None*›
using *other-inv* **unfolding** *final-twl-state-def* **by** *fast+*

have ‹*cdcl-twl-stgy*** *T U*›
by (*meson* ‹*cdcl-twl-o*** *S′ U*› *assms*(*5*) *rtranclp-cdcl-twl-cp-stgyD rtranclp-cdcl-twl-o-stgyD rtranclp-trans*)
have
twl-restart-after-restart:
‹*cdcl-twl-stgy-restart* (*T*, *m*) (*V*, *Suc m*)› **and**
rtranclp-twl-restart-after-restart:
‹*cdcl-twl-stgy-restart*** (*S*, *0*) (*V*, *m* + *1*)› **and**
cdcl-twl-stgy-restart-with-leftovers-after-restart:
‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*V*, *m* + *1*)› **and**
cdcl-twl-stgy-restart-with-leftovers-after-restart-T-V:
‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*V*, *Suc m*)› **and**
cdcl-twl-stgy-restart-with-leftovers1-after-restart:
‹*cdcl-twl-stgy-restart-with-leftovers1* (*T*, *m*) (*V*, *Suc m*)›
if
f: ‹*True* ⟶ *f* (*snd* (*U*, *m*)) < *size* (*get-learned-clss* (*fst* (*U*, *m*)))› **and**
res: ‹*cdcl-twl-restart U V*› **and**
[*simp*]: ‹*brk′* = *False*›
for *V*
proof −
have ‹*S′* ≠ *U*›
using *lits-to-update lits-to-upd-U* **by** *auto*
then have ‹*cdcl-twl-o*++ *S′ U*›
using ‹*cdcl-twl-o*** *S′ U*› **unfolding** *rtranclp-unfold* **by** *auto*
then have *st*: ‹*cdcl-twl-stgy*++ *T U*›
by (*meson local.cp rtranclp-cdcl-twl-cp-stgyD rtranclp-tranclp-tranclp tranclp-cdcl-twl-o-stgyD*)

show *twl-res-T-V*: ‹*cdcl-twl-stgy-restart* (*T*, *m*) (*V*, *Suc m*)›
apply (*rule cdcl-twl-stgy-restart.restart-step*[*of - U*])
subgoal by (*rule st*)
subgoal using *f* **by** *simp*
subgoal by (*rule res*)
done
show ‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*V*, *Suc m*)›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
by (*rule exI*[*of -* ‹*V*›])(*auto simp*: *twl-res-T-V*)
show ‹*cdcl-twl-stgy-restart*** (*S*, *0*) (*V*, *m* + *1*)›

using *twl-res twl-res-T-V*
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
by (*auto dest*: *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart*)
then show ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*V*, *m* + *1*)›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def* **apply** −
by (*rule exI*[*of* - *V*]) *auto*
show ‹*cdcl-twl-stgy-restart-with-leftovers1* (*T*, *m*) (*V*, *Suc m*)›
using *twl-res-T-V*
unfolding *cdcl-twl-stgy-restart-with-leftovers1-def*
by *fast*
qed
have
rtranclp-twl-restart-after-restart-S-U:
‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)› **and**
rtranclp-twl-restart-after-restart-T-U:
‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*U*, *m*)›
proof −
obtain *Ta* **where**
S-Ta: ‹*cdcl-twl-stgy-restart*** (*S*, *0*) (*Ta*, *snd* (*T*, *m*))›
‹*cdcl-twl-stgy*** *Ta* (*fst* (*T*, *m*))›
using *twl-res* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
by *auto*
then have ‹*cdcl-twl-stgy*** *Ta* (*fst* (*U*, *m*))›
using ‹*cdcl-twl-stgy*** *T U*› **by** *auto*
then show ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)›
using *S-Ta* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
by *fastforce*
show ‹*cdcl-twl-stgy-restart-with-leftovers* (*T*, *m*) (*U*, *m*)›
using ‹*cdcl-twl-stgy*** *T U*› **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
by *fastforce*
qed
have
rtranclp-twl-restart-after-restart-brk:
‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)›
if
[*simp*]: ‹*brk′* = *True*›
proof −
have ‹*full1 cdcl-twl-stgy T U* ∨ *T* = *U* ∨ *get-conflict U* ≠ *None*›
using *brk′-no-step* ‹*cdcl-twl-stgy*** *T U*›
unfolding *rtranclp-unfold full1-def final-twl-state-def* **by** *auto*
then consider
(*step*) ‹*cdcl-twl-stgy-restart* (*T*, *m*) (*U*, *m*)› |
(*TU*) ‹*T* = *U*› |
(*final*) ‹*get-conflict U* ≠ *None*›
by (*auto dest*!: *cdcl-twl-stgy-restart.intros*)
then show ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*U*, *m*)›
proof *cases*
case *step*
then show *?thesis*
using *twl-res* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
using *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*[*of T m U*] **apply** −
by (*rule exI*[*of* - *U*]) (*fastforce dest*!:)
next
case [*simp*]: *TU*
then show *?thesis*
using *twl-res* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*

using *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*[*of* T m U] **apply** −
by *fastforce*
next
case *final*
then show *?thesis*
using *twl-res* ‹*cdcl-twl-stgy*** T U› **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def*
using *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*[*of* T m U] **apply** −
by *fastforce*
qed
qed
have *cdcl-twl-stgy-restart-with-leftovers1-T-U*:
‹*cdcl-twl-stgy-restart-with-leftovers1* (T, m) (U, m) $\longleftrightarrow$ $T \neq U$›
proof −
have ‹*cdcl-twl-stgy*$^{++}$ T U $\vee$ $T = U$›
using ‹*cdcl-twl-stgy*** T U› **unfolding** *rtranclp-unfold* **by** *auto*
then show *?thesis*
using *wf-not-refl*[*OF wf-cdcl-twl-stgy-restart*, *of* ‹(U, m)›]
using *wf-not-refl*[*OF tranclp-wf-cdcl-twl-stgy*, *of* ‹U›]
struct-invs-U
unfolding *cdcl-twl-stgy-restart-with-leftovers1-def* **by** *auto*
qed
have *brk′-eq*: ‹¬*cdcl-twl-stgy-restart-with-leftovers1* (T, m) (U, m) $\Longrightarrow$ *brk′*›
using *cdcl-o lits-to-upd-U lits-to-update local.cp*
unfolding *cdcl-twl-stgy-restart-with-leftovers1-def*
unfolding *rtranclp-unfold*
by (*auto dest*!: *tranclp-cdcl-twl-o-stgyD tranclp-cdcl-twl-cp-stgyD*
simp: *rtranclp-unfold*
dest: *rtranclp-tranclp-tranclp tranclp-trans*)

have *H*[*simp*]: ‹*brk* = *False*› ‹*ebrk* = *False*›
using *cond* **by** *auto*
show *?B*
unfolding *restart-prog-def restart-required-def*
apply (*refine-vcg*; *remove-dummy-vars*)
subgoal using *struct-invs-U stgy-invs-U confl-U*
unfolding *restart-prog-pre-def cdcl-twl-stgy-restart-prog-inv-def H* **by** *fast*
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
apply (*intro conjI*)
subgoal by (*rule struct-invs′*)
subgoal by (*rule stgy-invs*)
subgoal unfolding *H* **by** *fast*
subgoal by (*rule cdcl-twl-stgy-restart-with-leftovers-after-restart*) *simp-all*
subgoal by (*rule res-no-clss-to-upd*)
subgoal by (*simp add*: *res-no-confl*)
done
subgoal
using *cdcl-twl-stgy-restart-with-leftovers1-T-U brk′-eq struct-invs-T*
by (*simp add*: *clss-to-upd-U confl-U rtranclp-twl-restart-after-restart-S-U*
stgy-invs-U struct-invs-U twl-restart-ops.cdcl-twl-stgy-restart-prog-inv-def
cdcl-twl-stgy-restart-with-leftovers1-after-restart)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
apply (*intro conjI*)
subgoal by (*rule struct-invs′*)
subgoal by (*rule stgy-invs*)

subgoal unfolding *H* by *fast*
subgoal by (*rule cdcl-twl-stgy-restart-with-leftovers-after-restart*) *simp-all*
subgoal by (*rule res-no-clss-to-upd*)
subgoal by (*simp add*: *res-no-confl*)
done
subgoal
using *cdcl-twl-stgy-restart-with-leftovers1-T-U brk′-eq*
by (*auto simp*: *twl-restart-after-restart struct-invs-S struct-invs-T cdcl-twl-stgy-restart-with-leftovers-after-restart-T-V struct-invs-U rtranclp-twl-restart-after-restart-brk rtranclp-twl-restart-after-restart-T-U cdcl-twl-stgy-restart-with-leftovers1-after-restart brk′-no-step clss-to-upd-U restart-prog-pre-def rtranclp-twl-restart-after-restart-S-U twl-restart-ops.cdcl-twl-stgy-restart-prog-inv-def*)
subgoal
using *cdcl-twl-stgy-restart-with-leftovers1-T-U brk′-eq*
by (*auto simp*: *twl-restart-after-restart struct-invs-S struct-invs-T cdcl-twl-stgy-restart-with-leftovers-after-restart-T-V struct-invs-U rtranclp-twl-restart-after-restart-brk rtranclp-twl-restart-after-restart-T-U cdcl-twl-stgy-restart-with-leftovers1-after-restart brk′-no-step clss-to-upd-U restart-prog-pre-def rtranclp-twl-restart-after-restart-S-U twl-restart-ops.cdcl-twl-stgy-restart-prog-inv-def*)
subgoal
using *cdcl-twl-stgy-restart-with-leftovers1-T-U brk′-eq*
by (*auto simp*: *twl-restart-after-restart struct-invs-S struct-invs-T cdcl-twl-stgy-restart-with-leftovers-after-restart-T-V struct-invs-U rtranclp-twl-restart-after-restart-brk rtranclp-twl-restart-after-restart-T-U cdcl-twl-stgy-restart-with-leftovers1-after-restart*)
done
qed

lemma *cdcl-twl-stgy-restart-with-leftovers-refl*: ‹*cdcl-twl-stgy-restart-with-leftovers S S*›
unfolding *cdcl-twl-stgy-restart-with-leftovers-def*
by (*rule exI*[*of* - ‹*fst S*›]) *auto*

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-spec*:
assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and** ‹*get-conflict S* = *None*›
shows
‹*cdcl-twl-stgy-restart-prog S* ≤ *SPEC*(λ*T*. ∃ *n*. *cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *n*) ∧ *final-twl-state T*)›
(**is** ‹- ≤ *SPEC*(λ*T*. *?P T*)›)
proof −
have *final-prop*: ‹*?P T*›
if
inv: ‹*case* (*brk*, *T*, *n*) *of* (*brk*, *T*, *m*) ⇒ *cdcl-twl-stgy-restart-prog-inv S brk T m*› **and**
‹¬ (*case* (*brk*, *T*, *n*) *of* (*brk*, *uu-*) ⇒ ¬ *brk*)›
for *brk T n*
proof −
have
‹*brk*› **and**
‹*twl-struct-invs T*› **and**
‹*twl-stgy-invs T*› **and**
ns: ‹*final-twl-state T*› **and**
twl-left-overs: ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *n*)› **and**
‹*clauses-to-update T* = {#}›

using *that* **unfolding** *cdcl-twl-stgy-restart-prog-inv-def* **by** *auto*
obtain S' **where**
st: ‹*cdcl-twl-stgy-restart*** $(S, 0)$ (S', n)› **and**
S'-T: ‹*cdcl-twl-stgy*** S' T›
using *twl-left-overs* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def* **by** *auto*
then show *?thesis*
using *ns* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def* **apply** −
apply (*rule-tac* $x=n$ **in** *exI*)
apply (*rule conjI*)
subgoal by (*rule-tac* $x=S'$ **in** *exI*) *auto*
subgoal by *auto*
done
qed
show *?thesis*
supply *RETURN-as-SPEC-refine*[*refine2 del*]
unfolding *cdcl-twl-stgy-restart-prog-def full-def cdcl-twl-stgy-restart-prog-inv-def*
apply (*refine-vcg WHILEIT-rule*[**where**
R = ‹{((*brkT*, T, n), (*brkS*, S, m)).
twl-struct-invs S ∧ *cdcl-twl-stgy-restart-with-leftovers1* (S, m) (T, n)} ∪
{((*brkT*, T), (*brkS*, S)). $S = T$ ∧ *brkT* ∧ ¬*brkS*}›];
remove-dummy-vars)
subgoal by (*rule wf-cdcl-twl-stgy-restart-measure*)
subgoal using *assms* **by** *fast*
subgoal using *assms* **by** *fast*
subgoal using *assms* **by** *fast*
subgoal by (*rule cdcl-twl-stgy-restart-with-leftovers-refl*)
subgoal using *assms* **by** *fast*
subgoal using *assms* **by** *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp*)
subgoal by *fast*
subgoal by (*rule restart-prog-spec*[*unfolded cdcl-twl-stgy-restart-prog-inv-def*])
subgoal by (*rule final-prop*[*unfolded cdcl-twl-stgy-restart-prog-inv-def*])
done
qed

definition *cdcl-twl-stgy-restart-prog-early* :: *'v twl-st* ⇒ *'v twl-st nres* **where**
‹*cdcl-twl-stgy-restart-prog-early* S_0 =
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, T, n) ← *WHILE*$_T$$^{\lambda(ebrk,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, S, n).
do {
T ← *unit-propagation-outer-loop* S;
(*brk*, T) ← *cdcl-twl-o-prog* T;
(T, n) ← *restart-prog* T n *brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk*, T, n)
})
(*ebrk*, *False*, S_0, *0*);
if ¬*brk then do* {

```
      (brk, T, -) ← WHILE_T^λ(brk, T, n). cdcl-twl-stgy-restart-prog-inv S₀ brk T n
  (λ(brk, -). ¬brk)
  (λ(brk, S, n).
  do {
    T ← unit-propagation-outer-loop S;
    (brk, T) ← cdcl-twl-o-prog T;
    (T, n) ← restart-prog T n brk;
    RETURN (brk, T, n)
  })
  (False, T, n);
      RETURN T
    }
    else RETURN T
  }›

lemma (in twl-restart) cdcl-twl-stgy-prog-early-spec:
  assumes ‹twl-struct-invs S› and ‹twl-stgy-invs S› and ‹clauses-to-update S = {#}› and
    ‹get-conflict S = None›
  shows
    ‹cdcl-twl-stgy-restart-prog-early S ≤ SPEC(λT. ∃n. cdcl-twl-stgy-restart-with-leftovers (S, 0) (T, n)
∧
        final-twl-state T)›
    (is ‹- ≤ SPEC(λT. ?P T)›)
proof −
  have final-prop: ‹?P T›
    if
     inv: ‹case (brk, T, n) of (brk, T, m) ⇒ cdcl-twl-stgy-restart-prog-inv S brk T m› and
      ‹¬ (case (brk, T, n) of (brk, uu-) ⇒ ¬ brk)›
    for brk T n
  proof −
    have
      ‹brk› and
      ‹twl-struct-invs T› and
      ‹twl-stgy-invs T› and
      ns: ‹final-twl-state T› and
      twl-left-overs: ‹cdcl-twl-stgy-restart-with-leftovers (S, 0) (T, n)› and
      ‹clauses-to-update T = {#}›
      using that unfolding cdcl-twl-stgy-restart-prog-inv-def by auto
    obtain S' where
       st: ‹cdcl-twl-stgy-restart** (S, 0) (S', n)› and
       S'-T: ‹cdcl-twl-stgy** S' T›
      using twl-left-overs unfolding cdcl-twl-stgy-restart-with-leftovers-def by auto
    then show ?thesis
      using ns unfolding cdcl-twl-stgy-restart-with-leftovers-def apply −
      apply (rule-tac x=n in exI)
      apply (rule conjI)
      subgoal by (rule-tac x=S' in exI) auto
      subgoal by auto
      done
  qed
  show ?thesis
    supply RETURN-as-SPEC-refine[refine2 del]
    unfolding cdcl-twl-stgy-restart-prog-early-def full-def
    apply (refine-vcg
        WHILEIT-rule[where
          R = ‹{((ebrk, brkT, T, n), (ebrk, brkS, S, m)).
```

twl-struct-invs S ∧ *cdcl-twl-stgy-restart-with-leftovers1* (S, m) $(T, n)\}$ ∪
$\{((ebrkT, brkT, T), (ebrkS, brkS, S)).\ S = T \wedge (ebrkT \vee brkT) \wedge (\neg brkS \wedge \neg ebrkS)\}$›]
WHILEIT-rule[**where**
R = ‹$\{((brkT, T, n), (brkS, S, m))$.
twl-struct-invs S ∧ *cdcl-twl-stgy-restart-with-leftovers1* (S, m) $(T, n)\}$ ∪
$\{((brkT, T), (brkS, S)).\ S = T \wedge brkT \wedge \neg brkS\}$›];
remove-dummy-vars)
subgoal by (*rule wf-cdcl-twl-stgy-restart-measure-early*)
subgoal using *assms* **unfolding** *cdcl-twl-stgy-restart-prog-inv-def*
by (*fast intro*: *cdcl-twl-stgy-restart-with-leftovers-refl*)
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def*
by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp*)
subgoal by *fast*
subgoal for *ebrk brk T m x ac bc*
by (*rule restart-prog-early-spec*)
subgoal by (*rule wf-cdcl-twl-stgy-restart-measure*)
subgoal by *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def*
by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp*)
subgoal by *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def*
by (*rule restart-prog-spec*[*unfolded cdcl-twl-stgy-restart-prog-inv-def*])
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def*
by (*rule final-prop*[*unfolded cdcl-twl-stgy-restart-prog-inv-def*])
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def*
by *auto*
done
qed

definition *cdcl-twl-stgy-restart-prog-bounded* :: *'v twl-st* ⇒ (*bool* × *'v twl-st*) *nres* **where**
‹*cdcl-twl-stgy-restart-prog-bounded* S_0 =
do {
ebrk ← *RES UNIV*;
$(ebrk, brk, T, n) \leftarrow WHILE_T{}^{\lambda(ebrk,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S_0\ brk\ T\ n}$
$(\lambda(ebrk, brk, \text{-}).\ \neg brk \wedge \neg ebrk)$
$(\lambda(ebrk, brk, S, n).$
do {
T ← *unit-propagation-outer-loop* S;
(brk, T) ← *cdcl-twl-o-prog* T;
(T, n) ← *restart-prog* T n *brk*;
ebrk ← *RES UNIV*;
RETURN $(ebrk, brk, T, n)$
})
$(ebrk, False, S_0, 0)$;
RETURN (brk, T)
}›

lemma (**in** *twl-restart*) *cdcl-twl-stgy-prog-bounded-spec*:

assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and**
‹*get-conflict S* = *None*›
shows
‹*cdcl-twl-stgy-restart-prog-bounded S* ≤ *SPEC*(λ(*brk*, *T*). ∃ *n*. *cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *n*) ∧
(*brk* ⟶ *final-twl-state T*))›
(**is** ‹- ≤ *SPEC ?P*›)
proof −
have *final-prop*: ‹*?P* (*brk*, *T*)›
if
inv: ‹*case* (*brk*, *T*, *n*) *of* (*brk*, *T*, *m*) ⇒ *cdcl-twl-stgy-restart-prog-inv S brk T m*›
for *brk T n*
proof −
have
‹*twl-struct-invs T*› **and**
‹*twl-stgy-invs T*› **and**
ns: ‹*brk* ⟶ *final-twl-state T*› **and**
twl-left-overs: ‹*cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *n*)› **and**
‹*clauses-to-update T* = {#}›
using *that* **unfolding** *cdcl-twl-stgy-restart-prog-inv-def* **by** *auto*
obtain *S′* **where**
st: ‹*cdcl-twl-stgy-restart*** (*S*, *0*) (*S′*, *n*)› **and**
S′-T: ‹*cdcl-twl-stgy*** *S′ T*›
using *twl-left-overs* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def* **by** *auto*
then show *?thesis*
using *ns* **unfolding** *cdcl-twl-stgy-restart-with-leftovers-def prod.case* **apply** −
apply (*rule-tac x*=*n* **in** *exI*)
apply (*rule conjI*)
subgoal by (*rule-tac x*=*S′* **in** *exI*) *auto*
subgoal by *auto*
done
qed
show *?thesis*
supply *RETURN-as-SPEC-refine*[*refine2 del*]
unfolding *cdcl-twl-stgy-restart-prog-bounded-def full-def*
apply (*refine-vcg*
WHILEIT-rule[**where**
R = ‹{((*ebrk*, *brkT*, *T*, *n*), (*ebrk*, *brkS*, *S*, *m*)).
twl-struct-invs S ∧ *cdcl-twl-stgy-restart-with-leftovers1* (*S*, *m*) (*T*, *n*)} ∪
{((*ebrkT*, *brkT*, *T*), (*ebrkS*, *brkS*, *S*)). *S* = *T* ∧ (*ebrkT* ∨ *brkT*) ∧ (¬*brkS* ∧ ¬*ebrkS*)}›];
remove-dummy-vars)
subgoal by (*rule wf-cdcl-twl-stgy-restart-measure-early*)
subgoal using *assms* **unfolding** *cdcl-twl-stgy-restart-prog-inv-def*
by (*fast intro*: *cdcl-twl-stgy-restart-with-leftovers-refl*)
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def* **by** *fast*
subgoal unfolding *cdcl-twl-stgy-restart-prog-inv-def*
by (*simp add*: *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp*)
subgoal by *fast*
subgoal for *ebrk brk T m x ac bc*
by (*rule restart-prog-early-spec*)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def prod.case*
by (*rule final-prop*[*unfolded prod.case cdcl-twl-stgy-restart-prog-inv-def*])

done
qed
end

end
theory *Watched-Literals-List*
imports *WB-More-Refinement-List Watched-Literals-Algorithm CDCL.DPLL-CDCL-W-Implementation Refine-Monadic.Refine-Monadic*
begin

lemma *mset-take-mset-drop-mset*: ‹$(\lambda x.\ mset\ (take\ 2\ x) + mset\ (drop\ 2\ x)) = mset$›
unfolding *mset-append*[*symmetric*] *append-take-drop-id* **..**
lemma *mset-take-mset-drop-mset'*: ‹$mset\ (take\ 2\ x) + mset\ (drop\ 2\ x) = mset\ x$›
unfolding *mset-append*[*symmetric*] *append-take-drop-id* **..**

lemma *uminus-lit-of-image-mset*:
‹$\{\#- \mathit{lit\text{-}of}\ x\ .\ x \in\#\ A\#\} = \{\#- \mathit{lit\text{-}of}\ x.\ x \in\#\ B\#\} \longleftrightarrow$
$\{\#\mathit{lit\text{-}of}\ x\ .\ x \in\#\ A\#\} = \{\#\mathit{lit\text{-}of}\ x.\ x \in\#\ B\#\}$›
for A :: ‹$('a\ literal,\ 'a\ literal,\ 'b)\ \mathit{annotated\text{-}lit}\ multiset$›
proof −
have *1*: ‹$(\lambda x.\ -\mathit{lit\text{-}of}\ x)$ ‘# A = *uminus* ‘# *lit-of* ‘# A›
for A :: ‹$('d::uminus,\ 'd,\ 'e)\ \mathit{annotated\text{-}lit}\ multiset$›
by *auto*
show *?thesis*
unfolding *1*
by (*rule inj-image-mset-eq-iff*) (*auto simp*: *inj-on-def*)
qed

1.3 Second Refinement: Lists as Clause

1.3.1 Types

type-synonym *'v clauses-to-update-l* = ‹*nat multiset*›

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

type-synonym *'v twl-st-l* =
‹(*'v*, *nat*) *ann-lits* × *'v clauses-l* ×
'v cconflict × *'v clauses* × *'v clauses* × *'v clauses-to-update-l* × *'v lit-queue*›

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

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

fun *set-clauses-to-update-l* :: ‹*'v clauses-to-update-l* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*set-clauses-to-update-l WS* (*M*, *N*, *D*, *NE*, *UE*, -, *Q*) = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›

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

fun *set-literals-to-update-l* :: ‹′*v clause* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l*› **where**
‹*set-literals-to-update-l Q* (*M*, *N*, *D*, *NE*, *UE*, *WS*, -) = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›

fun *get-conflict-l* :: ‹′*v twl-st-l* ⇒ ′*v cconflict*› **where**
‹*get-conflict-l* (-, -, *D*, -, -, -, -) = *D*›

fun *get-clauses-l* :: ‹′*v twl-st-l* ⇒ ′*v clauses-l*› **where**
‹*get-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *N*›

fun *get-unit-clauses-l* :: ‹′*v twl-st-l* ⇒ ′*v clauses*› **where**
‹*get-unit-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *NE* + *UE*›

fun *get-unit-init-clauses-l* :: ‹′*v twl-st-l* ⇒ ′*v clauses*› **where**
‹*get-unit-init-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *NE*›

fun *get-unit-learned-clauses-l* :: ‹′*v twl-st-l* ⇒ ′*v clauses*› **where**
‹*get-unit-learned-clauses-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *UE*›

fun *get-init-clauses* :: ‹′*v twl-st* ⇒ ′*v twl-clss*› **where**
‹*get-init-clauses* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) = *N*›

fun *get-unit-init-clauses* :: ‹′*v twl-st-l* ⇒ ′*v clauses*› **where**
‹*get-unit-init-clauses* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *NE*›

fun *get-unit-learned-clss* :: ‹′*v twl-st-l* ⇒ ′*v clauses*› **where**
‹*get-unit-learned-clss* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *UE*›

lemma *state-decomp-to-state*:
‹(*case S of* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⇒ *P M N U D NE UE WS Q*) =
P (*get-trail S*) (*get-init-clauses S*) (*get-learned-clss S*) (*get-conflict S*)
(*unit-init-clauses S*) (*get-init-learned-clss S*)
(*clauses-to-update S*)
(*literals-to-update S*)›
by (*cases S*) *auto*

lemma *state-decomp-to-state-l*:
‹(*case S of* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) ⇒ *P M N D NE UE WS Q*) =
P (*get-trail-l S*) (*get-clauses-l S*) (*get-conflict-l S*)
(*get-unit-init-clauses-l S*) (*get-unit-learned-clauses-l S*)
(*clauses-to-update-l S*)
(*literals-to-update-l S*)›
by (*cases S*) *auto*

definition *set-conflict′* :: ‹′*v clause option* ⇒ ′*v twl-st* ⇒ ′*v twl-st*› **where**
‹*set-conflict′* = (λ*C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*). (*M*, *N*, *U*, *C*, *NE*, *UE*, *WS*, *Q*))›

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

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

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

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

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

Taken from *fun-upd*.

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

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

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

inductive *convert-lit*
:: ‹′v clauses-l ⇒ ′v clauses ⇒ (′v, nat) ann-lit ⇒ (′v, ′v clause) ann-lit ⇒ bool›
where
‹convert-lit N E (Decided K) (Decided K)› |
‹convert-lit N E (Propagated K C) (Propagated K C′)›
if ‹C′ = mset (N ∝ C)› **and** ‹C ≠ 0› |
‹convert-lit N E (Propagated K C) (Propagated K C′)›
if ‹C = 0› **and** ‹C′ ∈# E›

definition *convert-lits-l* **where**
‹convert-lits-l N E = ⟨p2rel (convert-lit N E)⟩ list-rel›

lemma *convert-lits-l-nil*[*simp*]:
‹([], a) ∈ convert-lits-l N E ⟷ a = []›
‹(b, []) ∈ convert-lits-l N E ⟷ b = []›
by (*auto simp*: *convert-lits-l-def*)

lemma *convert-lits-l-cons*[*simp*]:
‹(L # M, L′ # M′) ∈ convert-lits-l N E ⟷
convert-lit N E L L′ ∧ (M, M′) ∈ convert-lits-l N E›
by (*auto simp*: *convert-lits-l-def p2rel-def*)

lemma *take-convert-lits-lD*:
‹(M, M′) ∈ convert-lits-l N E ⟹
(take n M, take n M′) ∈ convert-lits-l N E›
by (*auto simp*: *convert-lits-l-def list-rel-def*)

lemma *convert-lits-l-consE*:
‹(Propagated L C # M, x) ∈ convert-lits-l N E ⟹
(⋀L′ C′ M′. x = Propagated L′ C′ # M′ ⟹ (M, M′) ∈ convert-lits-l N E ⟹
convert-lit N E (Propagated L C) (Propagated L′ C′) ⟹ P) ⟹ P›
by (*cases x*) (*auto simp*: *convert-lit.simps*)

lemma *convert-lits-l-append*[*simp*]:
‹length M1 = length M1′ ⟹
(M1 @ M2, M1′ @ M2′) ∈ convert-lits-l N E ⟷ (M1, M1′) ∈ convert-lits-l N E ∧

$(M2, M2') \in$ *convert-lits-l N E* ›
by (*auto simp*: *convert-lits-l-def list-rel-append2 list-rel-imp-same-length*)

lemma *convert-lits-l-map-lit-of*: ‹$(ay, bq) \in$ *convert-lits-l N e* $\Longrightarrow$ *map lit-of ay* = *map lit-of bq*›
apply (*induction ay arbitrary*: *bq*)
subgoal by *auto*
subgoal for *L M bq* **by** (*cases bq*) (*auto simp*: *convert-lit.simps*)
done

lemma *convert-lits-l-tlD*:
‹$(M, M') \in$ *convert-lits-l N E* $\Longrightarrow$
(*tl M*, *tl M'*) $\in$ *convert-lits-l N E*›
by (*cases M*; *cases M'*) *auto*

lemma *get-clauses-l-set-clauses-to-update-l*[*simp*]:
‹*get-clauses-l* (*set-clauses-to-update-l WC S*) = *get-clauses-l S*›
by (*cases S*) *auto*

lemma *get-trail-l-set-clauses-to-update-l*[*simp*]:
‹*get-trail-l* (*set-clauses-to-update-l WC S*) = *get-trail-l S*›
by (*cases S*) *auto*

lemma *get-trail-set-clauses-to-update*[*simp*]:
‹*get-trail* (*set-clauses-to-update WC S*) = *get-trail S*›
by (*cases S*) *auto*

abbreviation *resolve-cls-l* **where**
‹*resolve-cls-l L D' E* $\equiv$ *union-mset-list* (*remove1* $(-L)$ *D'*) (*remove1 L E*)›

lemma *mset-resolve-cls-l-resolve-cls*[*iff*]:
‹*mset* (*resolve-cls-l L D' E*) = $cdcl_W$*-restart-mset.resolve-cls L* (*mset D'*) (*mset E*)›
by (*auto simp*: *union-mset-list*[*symmetric*])

lemma *resolve-cls-l-nil-iff*:
‹*resolve-cls-l L D' E* = [] $\longleftrightarrow$ $cdcl_W$*-restart-mset.resolve-cls L* (*mset D'*) (*mset E*) = {#}›
by (*metis mset-resolve-cls-l-resolve-cls mset-zero-iff*)

lemma *lit-of-convert-lit*[*simp*]:
‹*convert-lit N E L L'* $\Longrightarrow$ *lit-of L'* = *lit-of L*›
by (*auto simp*: *p2rel-def convert-lit.simps*)

lemma *is-decided-convert-lit*[*simp*]:
‹*convert-lit N E L L'* $\Longrightarrow$ *is-decided L'* $\longleftrightarrow$ *is-decided L*›
by (*cases L*) (*auto simp*: *p2rel-def convert-lit.simps*)

lemma *defined-lit-convert-lits-l*[*simp*]: ‹$(M, M') \in$ *convert-lits-l N E* $\Longrightarrow$
defined-lit M' = *defined-lit M*›
apply (*induction M arbitrary*: *M'*)
subgoal by *auto*
subgoal for *L M M'*
by (*cases M'*)
(*auto simp*: *defined-lit-cons*)
done

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

no-dup M′ ⟷ *no-dup M*›
apply (*induction M arbitrary*: *M′*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*) *auto*
done

lemma
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
count-decided-convert-lits-l[*simp*]:
‹*count-decided M′* = *count-decided M*›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def*)
done

lemma
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
get-level-convert-lits-l[*simp*]:
‹*get-level M′* = *get-level M*›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*fastforce simp*: *convert-lits-l-def p2rel-def get-level-cons-if split*: *if-splits*)+
subgoal for *L C M M′*
by (*cases M′*) (*auto simp*: *convert-lits-l-def p2rel-def get-level-cons-if*)
done

lemma
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
get-maximum-level-convert-lits-l[*simp*]:
‹*get-maximum-level M′* = *get-maximum-level M*›
by (*intro ext*, *rule get-maximum-level-cong*)
(*use assms* **in** *auto*)

lemma *list-of-l-convert-lits-l*[*simp*]:
assumes ‹(*M*, *M′*) ∈ *convert-lits-l N E*›
shows
‹*lits-of-l M′* = *lits-of-l M*›
using *assms*
apply (*induction M arbitrary*: *M′ rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for *L M M′*
by (*cases M′*)
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for *L C M M′*

by (*cases* M') (*auto simp*: *convert-lits-l-def p2rel-def*)
done

lemma *is-proped-hd-convert-lits-l*[*simp*]:
assumes ‹(M, M') ∈ *convert-lits-l* N E› **and** ‹$M \neq []$›
shows ‹*is-proped* (*hd* M') ⟷ *is-proped* (*hd* M)›
using *assms*
apply (*induction* M *arbitrary*: M' *rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for L M M'
by (*cases* M')
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for L C M M'
by (*cases* M') (*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*)
done

lemma *is-decided-hd-convert-lits-l*[*simp*]:
assumes ‹(M, M') ∈ *convert-lits-l* N E› **and** ‹$M \neq []$›
shows
‹*is-decided* (*hd* M') ⟷ *is-decided* (*hd* M)›
by (*meson assms*(*1*) *assms*(*2*) *is-decided-no-proped-iff is-proped-hd-convert-lits-l*)

lemma *lit-of-hd-convert-lits-l*[*simp*]:
assumes ‹(M, M') ∈ *convert-lits-l* N E› **and** ‹$M \neq []$›
shows
‹*lit-of* (*hd* M') = *lit-of* (*hd* M)›
by (*cases* M; *cases* M') (*use assms* **in** *auto*)

lemma *lit-of-l-convert-lits-l*[*simp*]:
assumes ‹(M, M') ∈ *convert-lits-l* N E›
shows
‹*lit-of* ‘ *set* M' = *lit-of* ‘ *set* M›
using *assms*
apply (*induction* M *arbitrary*: M' *rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for L M M'
by (*cases* M')
(*auto simp*: *convert-lits-l-def p2rel-def*)
subgoal for L C M M'
by (*cases* M') (*auto simp*: *convert-lits-l-def p2rel-def*)
done

The order of the assumption is important for simpler use.

lemma *convert-lits-l-extend-mono*:
assumes ‹(a,b) ∈ *convert-lits-l* N E›
‹∀ L i. *Propagated* L i ∈ *set* a ⟶ *mset* (N∝i) = *mset* (N'∝i)› **and** ‹E ⊆# E'›
shows
‹(a,b) ∈ *convert-lits-l* N' E'›
using *assms*
apply (*induction* a *arbitrary*: b *rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for l A b
by (*cases* b)
(*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*)
subgoal for l C A b
by (*cases* b)

(*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*)
done

lemma *convert-lits-l-nil-iff*[*simp*]:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows
‹*M'* = [] ⟷ *M* = []›
using *assms* **by** *auto*

lemma *convert-lits-l-atm-lits-of-l*:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows ‹*atm-of* ‘ *lits-of-l M* = *atm-of* ‘ *lits-of-l M'*›
using *assms* **by** *auto*

lemma *convert-lits-l-true-clss-clss*[*simp*]:
‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹ *M'* ⊨*as C* ⟷ *M* ⊨*as C*›
unfolding *true-annots-true-cls*
by (*auto simp*: *p2rel-def*)

lemma *convert-lit-propagated-decided*[*iff*]:
‹*convert-lit b d* (*Propagated x21 x22*) (*Decided x1*) ⟷ *False*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lit-decided*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Decided x2*) ⟷ *x1* = *x2*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lit-decided-propagated*[*iff*]:
‹*convert-lit b d* (*Decided x1*) (*Propagated x21 x22*) ⟷ *False*›
by (*auto simp*: *convert-lit.simps*)

lemma *convert-lits-l-lit-of-mset*[*simp*]:
‹(*a*, *af*) ∈ *convert-lits-l N E* ⟹ *lit-of* ‘# *mset af* = *lit-of* ‘# *mset a*›
apply (*induction a arbitrary*: *af*)
subgoal by *auto*
subgoal for *L M af*
by (*cases af*) *auto*
done

lemma *convert-lits-l-imp-same-length*:
‹(*a*, *b*) ∈ *convert-lits-l N E* ⟹ *length a* = *length b*›
by (*auto simp*: *convert-lits-l-def list-rel-imp-same-length*)

lemma *convert-lits-l-decomp-ex*:
assumes
H: ‹(*Decided K* # *a*, *M2*) ∈ *set* (*get-all-ann-decomposition x*)› **and**
xxa: ‹(*x*, *xa*) ∈ *convert-lits-l aa ac*›
shows ‹∃ *M2*. (*Decided K* # *drop* (*length xa* − *length a*) *xa*, *M2*)
∈ *set* (*get-all-ann-decomposition xa*)› (**is** *?decomp*) **and**
‹(*a*, *drop* (*length xa* − *length a*) *xa*) ∈ *convert-lits-l aa ac*› (**is** *?a*)
proof −
from *H* **obtain** *M3* **where**
x: ‹*x* = *M3* @ *M2* @ *Decided K* # *a*›
by *blast*
obtain *M3'* *M2'* *a'* **where**

xa: ‹*xa* = *M3'* @ *M2'* @ *Decided K* # *a'*› **and**
‹(*M3*, *M3'*) ∈ *convert-lits-l aa ac*› **and**
‹(*M2*, *M2'*) ∈ *convert-lits-l aa ac*› **and**
aa': ‹(*a*, *a'*) ∈ *convert-lits-l aa ac*›
using *xxa* **unfolding** *x*
by (*auto simp*: *list-rel-append1 convert-lits-l-def p2rel-def convert-lit.simps list-rel-split-right-iff*)
then have *a'*: ‹*a'* = *drop* (*length xa* − *length a*) *xa*› **and** [*simp*]: ‹*length xa* ≥ *length a*›
unfolding *xa* **by** (*auto simp*: *convert-lits-l-imp-same-length*)
show *?decomp*
using *get-all-ann-decomposition-ex*[*of K a'* ‹*M3'* @ *M2'*›]
unfolding *xa*
unfolding *a'*
by *auto*
show *?a*
using *aa'* **unfolding** *a'* **.**
qed

lemma *in-convert-lits-lD*:
‹*K* ∈ *set TM* ⟹
(*M*, *TM*) ∈ *convert-lits-l N NE* ⟹
∃ *K'*. *K'* ∈ *set M* ∧ *convert-lit N NE K' K*›
by (*auto 5 5 simp*: *convert-lits-l-def list-rel-append2 dest*!: *split-list p2relD elim*!: *list-relE*)

lemma *in-convert-lits-lD2*:
‹*K* ∈ *set M* ⟹
(*M*, *TM*) ∈ *convert-lits-l N NE* ⟹
∃ *K'*. *K'* ∈ *set TM* ∧ *convert-lit N NE K K'*›
by (*auto 5 5 simp*: *convert-lits-l-def list-rel-append1 dest*!: *split-list p2relD elim*!: *list-relE*)

lemma *convert-lits-l-filter-decided*: ‹(*S*, *S'*) ∈ *convert-lits-l M N* ⟹
map lit-of (*filter is-decided S'*) = *map lit-of* (*filter is-decided S*)›
apply (*induction S arbitrary*: *S'*)
subgoal by *auto*
subgoal for *L S S'*
by (*cases S'*) *auto*
done

lemma *convert-lits-lI*:
‹*length M* = *length M'* ⟹ (⋀*i*. *i* < *length M* ⟹ *convert-lit N NE* (*M*!*i*) (*M'*!*i*)) ⟹
(*M*, *M'*) ∈ *convert-lits-l N NE*›
by (*auto simp*: *convert-lits-l-def list-rel-def p2rel-def list-all2-conv-all-nth*)

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

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

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

definition *get-learned-clss-l* **where**
‹*get-learned-clss-l S* = *learned-clss-lf* (*get-clauses-l S*)›

abbreviation *init-clss-l* :: ⟨$'v$ *clauses-l* ⇒ ($'v$ *clause-l* × *bool*) *multiset*⟩ **where**
⟨*init-clss-l* $N \equiv \{\#C \in\# $ *ran-m* $N.$ *snd* $C\#\}$⟩

abbreviation *init-clss-lf* :: ⟨$'v$ *clauses-l* ⇒ $'v$ *clause-l multiset*⟩ **where**
⟨*init-clss-lf* $N \equiv$ *fst* ‘# *init-clss-l* N⟩

abbreviation *all-clss-l* :: ⟨$'v$ *clauses-l* ⇒ ($'v$ *clause-l* × *bool*) *multiset*⟩ **where**
⟨*all-clss-l* $N \equiv$ *init-clss-l* N + *learned-clss-l* N⟩

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

abbreviation *all-clss-lf* :: ⟨$'v$ *clauses-l* ⇒ $'v$ *clause-l multiset*⟩ **where**
⟨*all-clss-lf* $N \equiv$ *init-clss-lf* N + *learned-clss-lf* N⟩

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

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

definition *irred'* **where** ⟨*irred'* = *irred*⟩

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

fun *get-learned-clauses-l* :: ⟨$'v$ *twl-st-l* ⇒ $'v$ *clause-l multiset*⟩ **where**
⟨*get-learned-clauses-l* (M, N, D, NE, UE, WS, Q) = *learned-clss-lf* N⟩

lemma *ran-m-clause-upd*:
assumes
NC: ⟨$C \in\#$ *dom-m* N⟩
shows ⟨*ran-m* ($N(C \hookrightarrow C')$) =
add-mset (C', *irred* N C) (*remove1-mset* ($N \propto C$, *irred* N C) (*ran-m* N))⟩
proof −
define N' **where**
⟨N' = *fmdrop* C N⟩
have N-N': ⟨*dom-m* N = *add-mset* C (*dom-m* N')⟩
using NC **unfolding** N'-*def* **by** *auto*
have ⟨$C \notin\#$ *dom-m* N'⟩
using NC *distinct-mset-dom*[*of* N] **unfolding** N-N' **by** *auto*
then show *?thesis*
by (*auto simp*: N-N' *ran-m-def mset-set.insert-remove image-mset-remove1-mset-if*
intro!: *image-mset-cong*)
qed

lemma *ran-m-mapsto-upd*:
assumes
NC: ⟨$C \in\#$ *dom-m* N⟩
shows ⟨*ran-m* (*fmupd* C C' N) =
add-mset C' (*remove1-mset* ($N \propto C$, *irred* N C) (*ran-m* N))⟩

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

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

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

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

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

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

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

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

lemma *learned-clss-l-l-fmdrop*: ‹¬ *irred N C* ⟹ *C* ∈# *dom-m N* ⟹
  learned-clss-l (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*)) (*learned-clss-l N*)›
  using *distinct-mset-dom*[*of N*]

apply (*cases* ‹C ∈# *dom-m* N›)
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of* C - *the*] *dest*!: *multi-member-split*)

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

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

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

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

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

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

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

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

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

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

intro!: *image-mset-cong*)

definition *twl-st-l* :: ‹- ⇒ ($'v$ *twl-st-l* × $'v$ *twl-st*) *set*› **where**
‹*twl-st-l L* =
  {((*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*), (*M'*, *N'*, *U'*, *C'*, *NE'*, *UE'*, *WS'*, *Q'*)).
    (*M*, *M'*) ∈ *convert-lits-l N* (*NE*+*UE*) ∧
    N' = *twl-clause-of* '# *init-clss-lf N* ∧
    U' = *twl-clause-of* '# *learned-clss-lf N* ∧
    C' = *C* ∧
    NE' = *NE* ∧
    UE' = *UE* ∧
    WS' = (*case L of None* ⇒ {#} | *Some L* ⇒ *image-mset* (λ*j*. (*L*, *twl-clause-of* (*N* ∝ *j*))) *WS*) ∧
    Q' = *Q*
  }›

lemma *clss-state*$_W$*-of*[*twl-st*]:
  assumes ‹(*S*, *R*) ∈ *twl-st-l L*›
  shows
  ‹*init-clss* (*state*$_W$*-of R*) = *mset* '# (*init-clss-lf* (*get-clauses-l S*)) +
    get-unit-init-clauses-l S›
  ‹*learned-clss* (*state*$_W$*-of R*) = *mset* '# (*learned-clss-lf* (*get-clauses-l S*)) +
    get-unit-learned-clauses-l S›
 using *assms*
 by (*cases S*; *cases L*; *auto simp*: *init-clss.simps learned-clss.simps twl-st-l-def*
  mset-take-mset-drop-mset'; *fail*)+

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

lemma [*twl-st-l*]:
  assumes ‹(*S*, *T*) ∈ *twl-st-l L*›
  shows
    ‹(*get-trail-l S*, *get-trail T*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-unit-clauses-l S*)› **and**
    ‹*get-clauses T* = *twl-clause-of* '# *fst* '# *ran-m* (*get-clauses-l S*)› **and**
    ‹*get-conflict T* = *get-conflict-l S*› **and**
    ‹*L* = *None* ⟹ *clauses-to-update T* = {#}›
    ‹*L* ≠ *None* ⟹ *clauses-to-update T* =
      (λ*j*. (*the L*, *twl-clause-of* (*get-clauses-l S* ∝ *j*))) '# *clauses-to-update-l S*› **and**
    ‹*literals-to-update T* = *literals-to-update-l S*›
    ‹*backtrack-lvl* (*state*$_W$*-of T*) = *count-decided* (*get-trail-l S*)›
    ‹*unit-clss T* = *get-unit-clauses-l S*›
    ‹*cdcl*$_W$*-restart-mset.clauses* (*state*$_W$*-of T*) =
      mset '# *ran-mf* (*get-clauses-l S*) + *get-unit-clauses-l S*› **and**
    ‹*no-dup* (*get-trail T*) ⟷ *no-dup* (*get-trail-l S*)› **and**
    ‹*lits-of-l* (*get-trail T*) = *lits-of-l* (*get-trail-l S*)› **and**
    ‹*count-decided* (*get-trail T*) = *count-decided* (*get-trail-l S*)› **and**
    ‹*get-trail T* = [] ⟷ *get-trail-l S* = []› **and**
    ‹*get-trail T* ≠ [] ⟷ *get-trail-l S* ≠ []› **and**
    ‹*get-trail T* ≠ [] ⟹ *is-proped* (*hd* (*get-trail T*)) ⟷ *is-proped* (*hd* (*get-trail-l S*))›
    ‹*get-trail T* ≠ [] ⟹ *is-decided* (*hd* (*get-trail T*)) ⟷ *is-decided* (*hd* (*get-trail-l S*))›
    ‹*get-trail T* ≠ [] ⟹ *lit-of* (*hd* (*get-trail T*)) = *lit-of* (*hd* (*get-trail-l S*))›
    ‹*get-level* (*get-trail T*) = *get-level* (*get-trail-l S*)›
    ‹*get-maximum-level* (*get-trail T*) = *get-maximum-level* (*get-trail-l S*)›
    ‹*get-trail T* ⊨*as D* ⟷ *get-trail-l S* ⊨*as D*›
  using *assms* **unfolding** *twl-st-l-def all-clss-lf-ran-m*[*symmetric*]
  by (*auto split*: *option.splits simp*: *trail.simps clauses-def mset-take-mset-drop-mset'*)

lemma (**in** −) [*twl-st-l*]:
‹$(S, T) \in$ *twl-st-l b* $\Longrightarrow$ *get-all-init-clss T* = *mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses S*›
  by (*cases S*; *cases T*; *cases b*) (*auto simp*: *twl-st-l-def mset-take-mset-drop-mset′*)

lemma [*twl-st-l*]:
  assumes ‹$(S, T) \in$ *twl-st-l L*›
  shows ‹*lit-of* ‘ *set* (*get-trail T*) = *lit-of* ‘ *set* (*get-trail-l S*)›
  using *twl-st-l*[*OF assms*] **unfolding** *lits-of-def*
  by *simp*

lemma [*twl-st-l*]:
  ‹*get-trail-l* (*set-literals-to-update-l D S*) = *get-trail-l S*›
  by (*cases S*) *auto*

fun *remove-one-lit-from-wq* :: ‹*nat* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l*› **where**
  ‹*remove-one-lit-from-wq L* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = (*M*, *N*, *D*, *NE*, *UE*, *remove1-mset L WS*, *Q*)›

lemma [*twl-st-l*]: ‹*get-conflict-l* (*set-clauses-to-update-l W S*) = *get-conflict-l S*›
  by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*get-conflict-l* (*remove-one-lit-from-wq L S*) = *get-conflict-l S*›
  by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*literals-to-update-l* (*set-clauses-to-update-l Cs S*) = *literals-to-update-l S*›
  by (*cases S*) *auto*

lemma [*twl-st-l*]: ‹*get-unit-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-clauses-l S*›
  by (*cases S*) *auto*

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

lemma *init-clss-state-to-l*[*twl-st-l*]: ‹$(S, S') \in$ *twl-st-l L* $\Longrightarrow$
  init-clss ($state_W$*-of S′*) = *mset* ‘# *init-clss-lf* (*get-clauses-l S*) + *get-unit-init-clauses-l S*›
  by (*cases S*) (*auto simp*: *twl-st-l-def init-clss.simps mset-take-mset-drop-mset′*)

lemma [*twl-st-l*]:
  ‹*get-unit-init-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-init-clauses-l S*›
  by (*cases S*; *auto*; *fail*)+

lemma [*twl-st-l*]:
  ‹*get-unit-init-clauses-l* (*remove-one-lit-from-wq L S*) = *get-unit-init-clauses-l S*›
  by (*cases S*; *auto*; *fail*)+

lemma [*twl-st-l*]:
  ‹*get-clauses-l* (*remove-one-lit-from-wq L S*) = *get-clauses-l S*›
  ‹*get-trail-l* (*remove-one-lit-from-wq L S*) = *get-trail-l S*›
  by (*cases S*; *auto*; *fail*)+

lemma [*twl-st-l*]:
  ‹*get-unit-learned-clauses-l* (*set-clauses-to-update-l Cs S*) = *get-unit-learned-clauses-l S*›
  by (*cases S*) *auto*

lemma [*twl-st-l*]:
‹*get-unit-learned-clauses-l* (*remove-one-lit-from-wq L S*) = *get-unit-learned-clauses-l S*›
by (*cases S*) *auto*
lemma *literals-to-update-l-remove-one-lit-from-wq*[*simp*]:
‹*literals-to-update-l* (*remove-one-lit-from-wq L T*) = *literals-to-update-l T*›
by (*cases T*) *auto*

lemma *clauses-to-update-l-remove-one-lit-from-wq*[*simp*]:
‹*clauses-to-update-l* (*remove-one-lit-from-wq L T*) = *remove1-mset L* (*clauses-to-update-l T*)›
by (*cases T*) *auto*

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

lemma *unit-init-clauses-get-unit-init-clauses-l*[*twl-st-l*]:
‹(S, T) ∈ *twl-st-l L* ⟹ *unit-init-clauses T* = *get-unit-init-clauses-l S*›
by (*cases S*) (*auto simp*: *twl-st-l-def init-clss.simps*)

lemma *clauses-state-to-l*[*twl-st-l*]: ‹(S, S') ∈ *twl-st-l L* ⟹
$cdcl_W$*-restart-mset.clauses* ($state_W$*-of* S') = *mset* ‘# *ran-mf* (*get-clauses-l S*) +
get-unit-init-clauses-l S + *get-unit-learned-clauses-l S*›
apply (*subst all-clss-l-ran-m*[*symmetric*])
unfolding *image-mset-union*
by (*cases S*) (*auto simp*: *twl-st-l-def init-clss.simps mset-take-mset-drop-mset′ clauses-def*)

lemma *clauses-to-update-l-set-clauses-to-update-l*[*twl-st-l*]:
‹*clauses-to-update-l* (*set-clauses-to-update-l WS S*) = *WS*›
by (*cases S*) *auto*

lemma *hd-get-trail-twl-st-of-get-trail-l*:
‹(S, T) ∈ *twl-st-l L* ⟹ *get-trail-l S* ≠ [] ⟹
lit-of (*hd* (*get-trail T*)) = *lit-of* (*hd* (*get-trail-l S*))›
by (*cases S*; *cases* ‹*get-trail-l S*›; *cases* ‹*get-trail T*›) (*auto simp*: *twl-st-l-def*)

lemma *twl-st-l-mark-of-hd*:
‹(x, y) ∈ *twl-st-l b* ⟹
get-trail-l x ≠ [] ⟹
is-proped (*hd* (*get-trail-l x*)) ⟹
mark-of (*hd* (*get-trail-l x*)) > *0* ⟹
mark-of (*hd* (*get-trail y*)) = *mset* (*get-clauses-l x* ∝ *mark-of* (*hd* (*get-trail-l x*)))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;
cases ‹*hd* (*get-trail y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma *twl-st-l-lits-of-tl*:
‹(x, y) ∈ *twl-st-l b* ⟹
lits-of-l (*tl* (*get-trail y*)) = (*lits-of-l* (*tl* (*get-trail-l x*)))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;
cases ‹*hd* (*get-trail y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma *twl-st-l-mark-of-is-decided*:
‹(x, y) ∈ *twl-st-l b* ⟹
get-trail-l x ≠ [] ⟹
is-decided (*hd* (*get-trail y*)) = *is-decided* (*hd* (*get-trail-l x*))›
by (*cases* ‹*get-trail-l x*›; *cases* ‹*get-trail y*›; *cases* ‹*hd* (*get-trail-l x*)›;

cases ‹*hd* (*get-trail* *y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

lemma *twl-st-l-mark-of-is-proped*:
‹$(x, y) \in$ *twl-st-l* $b \Longrightarrow$
get-trail-l $x \neq [] \Longrightarrow$
is-proped (*hd* (*get-trail* *y*)) = *is-proped* (*hd* (*get-trail-l* *x*))›
by (*cases* ‹*get-trail-l* *x*›; *cases* ‹*get-trail* *y*›; *cases* ‹*hd* (*get-trail-l* *x*)›;
cases ‹*hd* (*get-trail* *y*)›)
(*auto simp*: *twl-st-l-def convert-lit.simps*)

fun *equality-except-trail* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*equality-except-trail* (M, N, D, NE, UE, WS, Q) $(M', N', D', NE', UE', WS', Q') \longleftrightarrow$
$N = N' \wedge D = D' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$›

fun *equality-except-conflict-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l* ⇒ *bool*› **where**
‹*equality-except-conflict-l* (M, N, D, NE, UE, WS, Q) $(M', N', D', NE', UE', WS', Q') \longleftrightarrow$
$M = M' \wedge N = N' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$›

lemma *equality-except-conflict-l-rewrite*:
assumes ‹*equality-except-conflict-l* *S* *T*›
shows
‹*get-trail-l* *S* = *get-trail-l* *T*› **and**
‹*get-clauses-l* *S* = *get-clauses-l* *T*›
using *assms* **by** (*cases* *S*; *cases* *T*; *auto*; *fail*)+

lemma *equality-except-conflict-l-alt-def*:
‹*equality-except-conflict-l* *S* *T* ⟷
get-trail-l *S* = *get-trail-l* *T* ∧ *get-clauses-l* *S* = *get-clauses-l* *T* ∧
get-unit-init-clauses-l *S* = *get-unit-init-clauses-l* *T* ∧
get-unit-learned-clauses-l *S* = *get-unit-learned-clauses-l* *T* ∧
literals-to-update-l *S* = *literals-to-update-l* *T* ∧
clauses-to-update-l *S* = *clauses-to-update-l* *T*›
by (*cases* *S*, *cases* *T*) *auto*

lemma *equality-except-conflict-alt-def*:
‹*equality-except-conflict* *S* *T* ⟷
get-trail *S* = *get-trail* *T* ∧ *get-init-clauses* *S* = *get-init-clauses* *T* ∧
get-learned-clss *S* = *get-learned-clss* *T* ∧
get-init-learned-clss *S* = *get-init-learned-clss* *T* ∧
unit-init-clauses *S* = *unit-init-clauses* *T* ∧
literals-to-update *S* = *literals-to-update* *T* ∧
clauses-to-update *S* = *clauses-to-update* *T*›
by (*cases* *S*, *cases* *T*) *auto*

1.3.2 Additional Invariants and Definitions

definition *twl-list-invs* **where**
‹*twl-list-invs* *S* ⟷
(∀ *C* ∈# *clauses-to-update-l* *S*. *C* ∈# *dom-m* (*get-clauses-l* *S*)) ∧
0 ∉# *dom-m* (*get-clauses-l* *S*) ∧
(∀ *L* *C*. *Propagated* *L* *C* ∈ *set* (*get-trail-l* *S*) ⟶ (*C* > *0* ⟶ *C* ∈# *dom-m* (*get-clauses-l* *S*) ∧
(*C* > *0* ⟶ *L* ∈ *set* (*watched-l* (*get-clauses-l* *S* ∝ *C*)) ∧
(*length* (*get-clauses-l* *S* ∝ *C*) > *2* ⟶ *L* = *get-clauses-l* *S* ∝ *C* ! *0*)))) ∧
distinct-mset (*clauses-to-update-l* *S*)›

definition *polarity* **where**
⟨*polarity M L* =
(*if undefined-lit M L then None else if L* ∈ *lits-of-l M then Some True else Some False*)⟩

lemma *polarity-None-undefined-lit*: ⟨*is-None* (*polarity M L*) $\Longrightarrow$ *undefined-lit M L*⟩
by (*auto simp*: *polarity-def split*: *if-splits*)

lemma *polarity-spec*:
assumes ⟨*no-dup M*⟩
shows
⟨*RETURN* (*polarity M L*) ≤ *SPEC*(λ*v*. (*v* = *None* ⟷ *undefined-lit M L*) ∧
(*v* = *Some True* ⟷ *L* ∈ *lits-of-l M*) ∧ (*v* = *Some False* ⟷ −*L* ∈ *lits-of-l M*))⟩
unfolding *polarity-def*
by *refine-vcg*
(*use assms* **in** ⟨*auto simp*: *defined-lit-map lits-of-def atm-of-eq-atm-of uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus
split: *option.splits*⟩)

lemma *polarity-spec′*:
assumes ⟨*no-dup M*⟩
shows
⟨*polarity M L* = *None* ⟷ *undefined-lit M L*⟩ **and**
⟨*polarity M L* = *Some True* ⟷ *L* ∈ *lits-of-l M*⟩ **and**
⟨*polarity M L* = *Some False* ⟷ −*L* ∈ *lits-of-l M*⟩
unfolding *polarity-def*
by (*use assms* **in** ⟨*auto simp*: *defined-lit-map lits-of-def atm-of-eq-atm-of uminus-lit-swap*
no-dup-cannot-not-lit-and-uminus
split: *option.splits*⟩)

definition *find-unwatched-l* **where**
⟨*find-unwatched-l M C* = *SPEC* (λ(*found*).
(*found* = *None* ⟷ (∀ *L*∈*set* (*unwatched-l C*). −*L* ∈ *lits-of-l M*)) ∧
(∀ *j*. *found* = *Some j* ⟶ (*j* < *length C* ∧ (*undefined-lit M* (*C*!*j*) ∨ *C*!*j* ∈ *lits-of-l M*) ∧ *j* ≥ *2*)))⟩

definition *set-conflict-l* :: ⟨′*v clause-l* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l*⟩ **where**
⟨*set-conflict-l* = (λ*C* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). (*M*, *N*, *Some* (*mset C*), *NE*, *UE*, {#}, {#}))⟩

definition *propagate-lit-l* :: ⟨′*v literal* ⇒ *nat* ⇒ *nat* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l*⟩ **where**
⟨*propagate-lit-l* = (λ*L′ C i* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
let N = (*if length* (*N* ∝ *C*) > *2 then N*(*C* ↪ (*swap* (*N* ∝ *C*) *0* (*Suc 0* − *i*))) *else N*) *in*
(*Propagated L′ C* # *M*, *N*, *D*, *NE*, *UE*, *WS*, *add-mset* (−*L′*) *Q*))⟩

definition *update-clause-l* :: ⟨*nat* ⇒ *nat* ⇒ *nat* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l nres*⟩ **where**
⟨*update-clause-l* = (λ*C i f* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). *do* {
let N′ = *N* (*C* ↪ (*swap* (*N*∝*C*) *i f*));
RETURN (*M*, *N′*, *D*, *NE*, *UE*, *WS*, *Q*)
})⟩

definition *unit-propagation-inner-loop-body-l-inv*
:: ⟨′*v literal* ⇒ *nat* ⇒ ′*v twl-st-l* ⇒ *bool*⟩
where
⟨*unit-propagation-inner-loop-body-l-inv L C S* ⟷
(∃ *S′*. (*set-clauses-to-update-l* (*clauses-to-update-l S* + {#*C*#}) *S*, *S′*) ∈ *twl-st-l* (*Some L*) ∧
twl-struct-invs S′ ∧
twl-stgy-invs S′ ∧

$C \in\#$ *dom-m* (*get-clauses-l S*) $\wedge$
$C > 0$ $\wedge$
$0 <$ *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
no-dup (*get-trail-l S*) $\wedge$
(*if* (*get-clauses-l S* $\propto$ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
1 − (*if* (*get-clauses-l S* $\propto$ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* $\propto$ *C*) $\wedge$
$L \in$ *set* (*watched-l* (*get-clauses-l S* $\propto$ *C*)) $\wedge$
get-conflict-l S = *None*
)
›

definition *unit-propagation-inner-loop-body-l* :: ‹*'v literal* ⇒ *nat* ⇒
'v twl-st-l ⇒ *'v twl-st-l nres*› **where**
‹*unit-propagation-inner-loop-body-l L C S* = *do* {
  ASSERT(*unit-propagation-inner-loop-body-l-inv L C S*);
  K ← *SPEC*(λ*K*. *K* ∈ *set* (*get-clauses-l S* ∝ *C*));
  let val-K = *polarity* (*get-trail-l S*) *K*;
  if val-K = *Some True then RETURN S*
  else do {
    let i = (*if* (*get-clauses-l S* ∝ *C*) ! *0* = *L then 0 else 1*);
    let L' = (*get-clauses-l S* ∝ *C*) ! (*1* − *i*);
    let val-L' = *polarity* (*get-trail-l S*) *L'*;
    if val-L' = *Some True*
    then RETURN S
    else do {
      f ← *find-unwatched-l* (*get-trail-l S*) (*get-clauses-l S* ∝ *C*);
      case f of
        None ⇒
          if val-L' = *Some False*
          then RETURN (*set-conflict-l* (*get-clauses-l S* ∝ *C*) *S*)
          else RETURN (*propagate-lit-l L' C i S*)
      | *Some f* ⇒ *do* {
          ASSERT(*f* < *length* (*get-clauses-l S* ∝ *C*));
          let K = (*get-clauses-l S* ∝ *C*)!*f*;
          let val-K = *polarity* (*get-trail-l S*) *K*;
          if val-K = *Some True then*
            RETURN S
          else
            update-clause-l C i f S
        }
    }
  }
}›

lemma *refine-add-invariants*:
  assumes
    ‹(*f S*) ≤ *SPEC*(λ*S'*. *Q S'*)› **and**
    ‹*y* ≤ ⇓ {(*S*, *S'*). *P S S'*} (*f S*)›
  shows ‹*y* ≤ ⇓ {(*S*, *S'*). *P S S'* ∧ *Q S'*} (*f S*)›
  using *assms* **unfolding** *pw-le-iff pw-conc-inres pw-conc-nofail* **by** *force*

lemma *clauses-tuple*[*simp*]:
  ‹$cdcl_W$*-restart-mset.clauses* (*M*, {#*f x* . *x* ∈# *init-clss-l N*#} + *NE*,
    {#*f x* . *x* ∈# *learned-clss-l N*#} + *UE*, *D*) = {#*f x*. *x* ∈# *all-clss-l N*#} + *NE* + *UE*›
  by (*auto simp*: *clauses-def simp flip*: *image-mset-union*)

lemma *valid-enqueued-alt-simps*[*simp*]:
⟨*valid-enqueued* $S \longleftrightarrow$
$(\forall (L, C) \in\# \textit{clauses-to-update } S.\ L \in\# \textit{watched } C \wedge C \in\# \textit{get-clauses } S \wedge$
$-L \in \textit{lits-of-l } (\textit{get-trail } S) \wedge \textit{get-level } (\textit{get-trail } S)\ L = \textit{count-decided } (\textit{get-trail } S)) \wedge$
$(\forall L \in\# \textit{literals-to-update } S.$
$-L \in \textit{lits-of-l } (\textit{get-trail } S) \wedge \textit{get-level } (\textit{get-trail } S)\ L = \textit{count-decided } (\textit{get-trail } S))$⟩
by (*cases S*) *auto*

declare *valid-enqueued.simps*[*simp del*]

lemma *set-clauses-simp*[*simp*]:
⟨$f\ `\ \{a.\ a \in\# \textit{ran-m } N \wedge \neg\ \textit{snd } a\} \cup f\ `\ \{a.\ a \in\# \textit{ran-m } N \wedge \textit{snd } a\} \cup A =$
$f\ `\ \{a.\ a \in\# \textit{ran-m } N\} \cup A$⟩
by *auto*

lemma *init-clss-l-clause-upd*:
⟨$C \in\# \textit{dom-m } N \Longrightarrow \textit{irred } N\ C \Longrightarrow$
$\textit{init-clss-l } (N(C \hookrightarrow C')) =$
$\textit{add-mset } (C', \textit{irred } N\ C)\ (\textit{remove1-mset } (N \propto C, \textit{irred } N\ C)\ (\textit{init-clss-l } N))$⟩
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd*:
⟨$C \in\# \textit{dom-m } N \Longrightarrow \textit{irred } N\ C \Longrightarrow$
$\textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{True})\ N) =$
$\textit{add-mset } (C', \textit{irred } N\ C)\ (\textit{remove1-mset } (N \propto C, \textit{irred } N\ C)\ (\textit{init-clss-l } N))$⟩
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *learned-clss-l-mapsto-upd*:
⟨$C \in\# \textit{dom-m } N \Longrightarrow \neg\textit{irred } N\ C \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) =$
$\textit{add-mset } (C', \textit{irred } N\ C)\ (\textit{remove1-mset } (N \propto C, \textit{irred } N\ C)\ (\textit{learned-clss-l } N))$⟩
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd-irrel*: ⟨$C \in\# \textit{dom-m } N \Longrightarrow \neg\textit{irred } N\ C \Longrightarrow$
$\textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) = \textit{init-clss-l } N$⟩
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *init-clss-l-mapsto-upd-irrel-notin*: ⟨$C \notin\# \textit{dom-m } N \Longrightarrow$
$\textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) = \textit{init-clss-l } N$⟩
by (*auto simp*: *ran-m-mapsto-upd-notin*)

lemma *learned-clss-l-mapsto-upd-irrel*: ⟨$C \in\# \textit{dom-m } N \Longrightarrow \textit{irred } N\ C \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmupd } C\ (C', \textit{True})\ N) = \textit{learned-clss-l } N$⟩
by (*auto simp*: *ran-m-mapsto-upd*)

lemma *learned-clss-l-mapsto-upd-notin*: ⟨$C \notin\# \textit{dom-m } N \Longrightarrow$
$\textit{learned-clss-l } (\textit{fmupd } C\ (C', \textit{False})\ N) = \textit{add-mset } (C', \textit{False})\ (\textit{learned-clss-l } N)$⟩
by (*auto simp*: *ran-m-mapsto-upd-notin*)

lemma *in-ran-mf-clause-inI*[*intro*]:
⟨$C \in\# \textit{dom-m } N \Longrightarrow i = \textit{irred } N\ C \Longrightarrow (N \propto C, i) \in\# \textit{ran-m } N$⟩
by (*auto simp*: *ran-m-def dom-m-def*)

lemma *init-clss-l-mapsto-upd-notin*:
⟨$C \notin\# \textit{dom-m } N \Longrightarrow \textit{init-clss-l } (\textit{fmupd } C\ (C', \textit{True})\ N) =$
$\textit{add-mset } (C', \textit{True})\ (\textit{init-clss-l } N)$⟩

by (*auto simp*: *ran-m-mapsto-upd-notin*)

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

lemma *clause-twl-clause-of*: ‹*clause* (*twl-clause-of* C) = *mset* C› **for** C
by (*cases* C; *cases* ‹*tl* C›) *auto*

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

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

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

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

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

lemma *unit-propagation-inner-loop-body-l*:
fixes i C :: *nat* **and** S :: ‹$'v$ *twl-st-l*› **and** S' :: ‹$'v$ *twl-st*› **and** L :: ‹$'v$ *literal*›
defines
C'[*simp*]: ‹$C' \equiv$ *get-clauses-l* $S \propto C$›
assumes
SS': ‹(S, S') $\in$ *twl-st-l* (*Some* L)› **and**
WS: ‹$C \in\#$ *clauses-to-update-l* S› **and**
struct-invs: ‹*twl-struct-invs* S'› **and**
add-inv: ‹*twl-list-invs* S› **and**
stgy-inv: ‹*twl-stgy-invs* S'›
shows
‹*unit-propagation-inner-loop-body-l* L C
(*set-clauses-to-update-l* (*clauses-to-update-l* S − {#C#}) S) ≤
⇓ {(S, S''). (S, S'') ∈ *twl-st-l* (*Some* L) ∧ *twl-list-invs* S ∧ *twl-stgy-invs* S'' ∧
twl-struct-invs S''}
(*unit-propagation-inner-loop-body* L (*twl-clause-of* C')
(*set-clauses-to-update* (*clauses-to-update* (S') − {#(L, *twl-clause-of* C')#}) S'))›
(**is** ‹*?A* ≤ ⇓ - *?B*›)

proof −
  let *?S* = ‹*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*›
  obtain *M N D NE UE WS Q* **where** *S*: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
    by (*cases S*) *auto*

  have *C-N-U*: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
    using *add-inv WS SS'* **by** (*auto simp*: *twl-list-invs-def*)
  let *?M* = ‹*get-trail-l S*›
  let *?N* = ‹*get-clauses-l S*›
  let *?WS* = ‹*clauses-to-update-l S*›
  let *?Q* = ‹*literals-to-update-l S*›

  define *i* :: *nat* **where** ‹*i* ≡ (*if get-clauses-l S*∝*C*!*0* = *L then 0 else 1*)›
  let *?L* = ‹*C'* ! *i*›
  let *?L'* = ‹*C'* ! (*Suc 0* − *i*)›
  have *inv*: ‹*twl-st-inv S'*› **and**
    cdcl-inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S'*)› **and**
    valid: ‹*valid-enqueued S'*›
    using *struct-invs WS* **by** (*auto simp*: *twl-struct-invs-def*)
  have
    w-q-inv: ‹*clauses-to-update-inv S'*› **and**
    dist: ‹*distinct-queued S'*› **and**
    no-dup: ‹*no-duplicate-queued S'*› **and**
    confl: ‹*get-conflict S'* ≠ *None* ⟹ *clauses-to-update S'* = {#} ∧ *literals-to-update S'* = {#}›
    using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast+*
  have *n-d*: ‹*no-dup ?M*› **and** *confl-inv*: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting* ($state_W$*-of S'*)›
    using *cdcl-inv SS'* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
      $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
    by (*auto simp*: *trail.simps comp-def twl-st*)

  then have *consistent*: ‹− *L* ∉ *lits-of-l ?M*› **if** ‹*L* ∈ *lits-of-l ?M*› **for** *L*
    using *consistent-interp-def distinct-consistent-interp that* **by** *blast*

  have *cons-M*: ‹*consistent-interp* (*lits-of-l ?M*)›
    using *n-d distinct-consistent-interp* **by** *fast*
  let *?C'* = ‹*twl-clause-of C'*›
  have *C'-N-U-or*: ‹*?C'* ∈# *twl-clause-of* ‘# (*init-clss-lf ?N*) ∨
      ?C' ∈# *twl-clause-of* ‘# *learned-clss-lf ?N*›
    using *WS valid SS'*
    unfolding *union-iff*[*symmetric*] *image-mset-union*[*symmetric*] *mset-append*[*symmetric*]
    by (*auto simp*: *twl-struct-invs-def*
        split: *prod.splits simp del*: *twl-clause-of.simps*)
  have *struct*: ‹*struct-wf-twl-cls ?C'*›
    using *C-N-U inv SS' WS valid* **unfolding** *valid-enqueued-alt-simps*
    by (*auto simp*: *twl-st-inv-alt-def Ball-ran-m-dom-struct-wf*
      simp del: *twl-clause-of.simps*)
  have *C'-N-U*: ‹*?C'* ∈# *twl-clause-of* ‘# *all-clss-lf ?N*›
    using *C'-N-U-or*
    unfolding *union-iff*[*symmetric*] *image-mset-union*[*symmetric*] *mset-append*[*symmetric*] **.**
  have *watched-C'*: ‹*mset* (*watched-l C'*) = {#*?L*, *?L'*#}›
    using *struct i-def SS'* **by** (*cases C*) (*auto simp*: *length-list-2 take-2-if*)
  then have *mset-watched-C*: ‹*mset* (*watched-l C'*) = {#*watched-l C'* ! *i*, *watched-l C'* ! (*Suc 0* − *i*)#}›
    using *i-def* **by** (*cases* ‹*twl-clause-of* (*get-clauses-l S* ∝ *C*)›) (*auto simp*: *take-2-if*)
  have *two-le-length-C*: ‹*2* ≤ *length C'*›
    by (*metis length-take linorder-not-le min-less-iff-conj numeral-2-eq-2 order-less-irrefl*
        size-add-mset size-eq-0-iff-empty size-mset watched-C')

obtain *WS'* **where** *WS'-def*: ‹*?WS = add-mset C WS'*›
  using *multi-member-split*[*OF WS*] **by** *auto*
then have *WS'-def'*: ‹*WS = add-mset C WS'*›
  unfolding *S* **by** *auto*
have *L*: ‹*L ∈ set (watched-l C')*› **and** *uL-M*: ‹*−L ∈ lits-of-l (get-trail-l S)*›
  using *valid SS'* **by** (*auto simp*: *WS'-def*)
have *C'-i*[*simp*]: ‹*C'!i = L*›
  using *L two-le-length-C* **by** (*auto simp*: *take-2-if i-def split*: *if-splits*)
then have [*simp*]: ‹*?N∝C!i = L*›
  by *auto*
have *C-0*: ‹*C > 0*› **and** *C-neq-0*[*iff*]: ‹*C ≠ 0*›
  using *assms*(*3*,*5*) **unfolding** *twl-list-invs-def* **by** (*auto dest*!: *multi-member-split*)

have *pre-inv*: ‹*unit-propagation-inner-loop-body-l-inv L C ?S*›
  unfolding *unit-propagation-inner-loop-body-l-inv-def*
proof (*rule exI*[*of - S'*], *intro conjI*)
  have *S-readd-C-S*: ‹*set-clauses-to-update-l* (*clauses-to-update-l ?S* + {#*C*#}) *?S = S*›
   unfolding *S WS'-def'* **by** *auto*
  show ‹(*set-clauses-to-update-l*
    (*clauses-to-update-l ?S* + {#*C*#})
    (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*),
   S') ∈ *twl-st-l* (*Some L*)›
    using *SS'* **unfolding** *S-readd-C-S* **.**
  show ‹*twl-stgy-invs S'*› ‹*twl-struct-invs S'*›
    using *assms* **by** *fast+*
  show ‹*C ∈# dom-m* (*get-clauses-l ?S*)›
    using *assms C-N-U* **by** *auto*
  show ‹*C > 0*›
    by (*rule C-0*)
  show ‹(*if get-clauses-l ?S ∝ C ! 0 = L then 0 else 1*) < *length* (*get-clauses-l ?S ∝ C*)›
    using *two-le-length-C* **by** *auto*
  show ‹*1* − (*if get-clauses-l ?S ∝ C ! 0 = L then 0 else 1*) < *length* (*get-clauses-l ?S ∝ C*)›
    using *two-le-length-C* **by** *auto*
  show ‹*length* (*get-clauses-l ?S ∝ C*) > *0*›
    using *two-le-length-C* **by** *auto*
  show ‹*no-dup* (*get-trail-l ?S*)›
    using *n-d* **by** *auto*
  show ‹*L ∈ set* (*watched-l* (*get-clauses-l ?S ∝ C*))›
    using *L* **by** *auto*
  show ‹*get-conflict-l ?S = None*›
    using *confl SS' WS* **by** (*cases* ‹*get-conflict-l S*›) (*auto dest*: *in-diffD*)
qed
have *i-def'*: ‹*i* = (*if get-clauses-l ?S ∝ C ! 0 = L then 0 else 1*)›
  unfolding *i-def* **by** *auto*
have ‹*twl-list-invs ?S*›
  using *add-inv C-N-U* **unfolding** *twl-list-invs-def S*
  by (*auto dest*: *in-diffD*)
then have *upd-rel*: ‹(*?S*,
   set-clauses-to-update (*remove1-mset* (*L*, *twl-clause-of C'*) (*clauses-to-update S'*)) *S'*)
  ∈ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*}›
  using *SS' WS*
  by (*auto simp*: *twl-st-l-def image-mset-remove1-mset-if*)
have ‹*twl-list-invs* (*set-conflict-l* (*get-clauses-l ?S ∝ C*) *?S*)›
  using *add-inv C-N-U* **unfolding** *twl-list-invs-def*
  by (*auto dest*: *in-diffD simp*: *set-conflicting-def S*
    set-conflict-l-def mset-take-mset-drop-mset')

then have *confl-rel*: ⟨(*set-conflict-l* (*get-clauses-l ?S* $\propto$ *C*) *?S*,
set-conflicting (*twl-clause-of C'*)
(*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of C'*) (*clauses-to-update S'*)) *S'*))
$\in$ {(*S*, *S'*). (*S*, *S'*) $\in$ *twl-st-l* (*Some L*) $\wedge$ *twl-list-invs S*}⟩
using *SS' WS* **by** (*auto simp*: *twl-st-l-def image-mset-remove1-mset-if set-conflicting-def*
set-conflict-l-def mset-take-mset-drop-mset')
have *propa-rel*:
⟨(*propagate-lit-l* (*get-clauses-l ?S* $\propto$ *C* ! (*1* $-$ *i*)) *C i*
(*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*),
propagate-lit L' (*twl-clause-of C'*)
(*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of C'*) (*clauses-to-update S'*)) *S'*))
$\in$ {(*S*, *S'*). (*S*, *S'*) $\in$ *twl-st-l* (*Some L*) $\wedge$ *twl-list-invs S*}⟩
if
⟨(*get-clauses-l ?S* $\propto$ *C* ! (*1* $-$ *i*), *L'*) $\in$ *Id*⟩ **and**
L'-undef: ⟨$-$ *L'* $\notin$ *lits-of-l*
(*get-trail*
(*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of C'*) (*clauses-to-update S'*)) *S'*)) ⟩
⟨*L'* $\notin$ *lits-of-l*
(*get-trail*
(*set-clauses-to-update*
(*remove1-mset* (*L*, *twl-clause-of C'*) (*clauses-to-update S'*))
S'))⟩
for *L'*
proof $-$
have [*simp*]: ⟨*mset* (*swap* (*N* $\propto$ *C*) *0* (*Suc 0* $-$ *i*)) = *mset* (*N* $\propto$ *C*)⟩
apply (*subst swap-multiset*)
using *two-le-length-C* **unfolding** *i-def*
by (*auto simp*: *S*)
have *mset-un-watched-swap*:
⟨*mset* (*watched-l* (*swap* (*N* $\propto$ *C*) *0* (*Suc 0* $-$ *i*))) = *mset* (*watched-l* (*N* $\propto$ *C*))⟩
⟨*mset* (*unwatched-l* (*swap* (*N* $\propto$ *C*) *0* (*Suc 0* $-$ *i*))) = *mset* (*unwatched-l* (*N* $\propto$ *C*))⟩
using *two-le-length-C* **unfolding** *i-def*
apply (*auto simp*: *S take-2-if*)
by (*auto simp*: *S swap-def*)

have *irred-init*: ⟨*irred N C* $\Longrightarrow$ (*N* $\propto$ *C*, *True*) $\in\#$ *init-clss-l N*⟩
using *C-N-U* **by** (*auto simp*: *S ran-def*)
have *init-unchanged*: ⟨{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* $\in\#$ *init-clss-l* (*N*(*C* $\hookrightarrow$ *swap* (*N* $\propto$ *C*) *0* (*Suc 0* $-$ *i*)))#} =
{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* $\in\#$ *init-clss-l N*#}⟩
using *C-N-U*
by (*cases* ⟨*irred N C*⟩) (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
mset-un-watched-swap init-clss-l-mapsto-upd-irrel
dest: *multi-member-split*[*OF irred-init*])

have *irred-init*: ⟨¬*irred N C* $\Longrightarrow$ (*N* $\propto$ *C*, *False*) $\in\#$ *learned-clss-l N*⟩
using *C-N-U* **by** (*auto simp*: *S ran-def*)
have *learned-unchanged*: ⟨{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* $\in\#$ *learned-clss-l* (*N*(*C* $\hookrightarrow$ *swap* (*N* $\propto$ *C*) *0* (*Suc 0* $-$ *i*)))#} =
{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* $\in\#$ *learned-clss-l N*#}⟩

using *C-N-U*
by (*cases* ‹*irred N C*›) (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if mset-un-watched-swap learned-clss-l-mapsto-upd learned-clss-l-mapsto-upd-irrel dest*: *multi-member-split*[*OF irred-init*])
have [*simp*]: ‹{#(*L*, *TWL-Clause* (*mset* (*watched-l*
(*fst* (*the* (*if C* = *x*
then Some (*swap* ($N \propto C$) *0* (*Suc 0* − *i*), *irred N C*)
else fmlookup N x)))))
(*mset* (*unwatched-l*
(*fst* (*the* (*if C* = *x*
then Some (*swap* ($N \propto C$) *0* (*Suc 0* − *i*), *irred N C*)
else fmlookup N x))))))
. *x* ∈# *WS*#} = {#(*L*, *TWL-Clause* (*mset* (*watched-l* ($N \propto x$))) (*mset* (*unwatched-l* ($N \propto x$))))
. *x* ∈# *WS*#}›
by (*rule image-mset-cong*) (*auto simp*: *mset-un-watched-swap*)
have *C′-0i*: ‹*C′* ! (*Suc 0* − *i*) ∈ *set* (*watched-l C′*)›
using *two-le-length-C* **by** (*auto simp*: *take-2-if S i-def*)

have *nth-swap-isabelle*: ‹*length a* ≥ *2* ⟹ *swap a 0* (*Suc 0* − *i*) ! *0* = *a* ! (*Suc 0* − *i*)›
for *a* :: ‹*′a list*›
using *two-le-length-C that* **apply** (*auto simp*: *swap-def S i-def*)
by (*metis* (*full-types*) *le0 neq0-conv not-less-eq-eq nth-list-update-eq numeral-2-eq-2*)
have [*simp*]: ‹*Propagated La C* ∉ *set M*› **for** *La*
proof (*rule ccontr*)
assume *H*:‹¬ *?thesis*›
then have ‹*La* ∈ *set* (*watched-l* ($N \propto C$))› **and**
‹*2* < *length* ($N \propto C$) ⟶ *La* = $N \propto C$! *0*›
using *add-inv C-N-U two-le-length-C mset-un-watched-swap C′-0i*
unfolding *twl-list-invs-def S* **by** *auto*
moreover have ‹*La* ∈ *lits-of-l M*›
using *H* **by** (*force simp*: *lits-of-def*)
ultimately show *False*
using *L′-undef that SS′ uL-M n-d C′-i S watched-C′ that*(*1*)
apply (*auto simp*: *S i-def dest*: *no-dup-consistentD split*: *if-splits*)
apply (*metis in-multiset-nempty member-add-mset no-dup-consistentD set-mset-mset*)
by (*metis* (*full-types*) *in-multiset-nempty member-add-mset no-dup-consistentD set-mset-mset*)
qed
have ‹*twl-list-invs*
(*Propagated* ($N \propto C$! (*Suc 0* − *i*)) *C* # *M*, *N*(*C* ↪ *swap* ($N \propto C$) *0* (*Suc 0* − *i*)),
D, *NE*, *UE*, *remove1-mset C WS*, *add-mset* (− $N \propto C$! (*Suc 0* − *i*)) *Q*)›
using *add-inv C-N-U two-le-length-C mset-un-watched-swap C′-0i*
unfolding *twl-list-invs-def*
by (*auto dest*: *in-diffD simp*: *set-conflicting-def set-conflict-l-def mset-take-mset-drop-mset′ S nth-swap-isabelle dest*!: *mset-eq-setD*)
moreover have
‹*convert-lit* (*N*(*C* ↪ *swap* ($N \propto C$) *0* (*Suc 0* − *i*))) (*NE* + *UE*)
(*Propagated* ($N \propto C$! (*Suc 0* − *i*)) *C*)
(*Propagated* ($N \propto C$! (*Suc 0* − *i*)) (*mset* ($N \propto C$)))›
by (*auto simp*: *convert-lit.simps C-0*)
moreover have ‹(*M*, *x*) ∈ *convert-lits-l N* (*NE* + *UE*) ⟹
(*M*, *x*) ∈ *convert-lits-l* (*N*(*C* ↪ *swap* ($N \propto C$) *0* (*Suc 0* − *i*))) (*NE* + *UE*)› **for** *x*
apply (*rule convert-lits-l-extend-mono*)
apply *assumption*
apply *auto*

done

moreover have

‹*convert-lit N* (*NE* + *UE*)

(*Propagated* (*N* ∝ *C* ! (*Suc 0* − *i*)) *C*)

(*Propagated* (*N* ∝ *C* ! (*Suc 0* − *i*)) (*mset* (*N* ∝ *C*)))›

by (*auto simp*: *convert-lit.simps C-0*)

moreover have ‹*twl-list-invs*

(*Propagated* (*N* ∝ *C* ! (*Suc 0* − *i*)) *C* # *M*, *N*, *D*, *NE*, *UE*,

remove1-mset C WS, *add-mset* (− *N* ∝ *C* ! (*Suc 0* − *i*)) *Q*)›

if ‹¬ *2* < *length* (*N* ∝ *C*)›

using *add-inv C-N-U two-le-length-C mset-un-watched-swap C′-0i that*

unfolding *twl-list-invs-def*

by (*auto dest*: *in-diffD simp*: *set-conflicting-def*

set-conflict-l-def mset-take-mset-drop-mset′ S nth-swap-isabelle

dest!: *mset-eq-setD*)

ultimately show *?thesis*

using *SS′ WS that* **by** (*auto simp*: *twl-st-l-def image-mset-remove1-mset-if propagate-lit-def*

propagate-lit-l-def mset-take-mset-drop-mset′ S learned-unchanged

init-unchanged mset-un-watched-swap intro: *convert-lit.simps*)

qed

have *update-clause-rel*: ‹(*if polarity*

(*get-trail-l*

(*set-clauses-to-update-l*

(*remove1-mset C* (*clauses-to-update-l S*)) *S*))

(*get-clauses-l*

(*set-clauses-to-update-l*

(*remove1-mset C* (*clauses-to-update-l S*)) *S*) ∝

C !

the K) =

Some True

then RETURN (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*)

else update-clause-l C i (*the K*) (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*))

≤ ⇓ {(*S*, *S′*). (*S*, *S′*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*}

(*update-clauseS L* (*twl-clause-of C′*) (*set-clauses-to-update* (*remove1-mset* (*L*, *twl-clause-of C′*) (*clauses-to-update S′*)) *S′*))›

(**is** ‹*?update-clss* ≤ ⇓ - -›)

if

L′: ‹(*get-clauses-l ?S* ∝ *C* ! (*1* − *i*), *L′*) ∈ *Id*› **and**

L′-M: ‹*L′* ∉ *lits-of-l*

(*get-trail*

(*set-clauses-to-update*

(*remove1-mset* (*L*, *twl-clause-of C′*) (*clauses-to-update S′*))

S′))› **and**

K: ‹*K* ∈ {*found*. (*found* = *None*) =

(∀ *L*∈*set* (*unwatched-l* (*get-clauses-l ?S* ∝ *C*)).

− *L* ∈ *lits-of-l* (*get-trail-l ?S*)) ∧

(∀ *j*. *found* = *Some j* ⟶

j < *length* (*get-clauses-l ?S* ∝ *C*) ∧

(*undefined-lit* (*get-trail-l ?S*) (*get-clauses-l ?S* ∝ *C* ! *j*) ∨

get-clauses-l ?S ∝ *C* ! *j* ∈ *lits-of-l* (*get-trail-l ?S*)) ∧

2 ≤ *j*)}› **and**

K-None: ‹*K* ≠ *None*›

for *L′* **and** *K*

proof −

obtain *K′* **where** [*simp*]: ‹*K* = *Some K′*›

using *K-None* **by** *auto*
have
K′-le: ‹$K' <$ *length* $(N \propto C)$› **and**
K′-2: ‹$2 \leq K'$› **and**
K′-M: ‹*undefined-lit M* $(N \propto C ! K') \vee$
$N \propto C ! K' \in$ *lits-of-l* (*get-trail-l S*) ›
using *K* **by** (*auto simp*: *S*)
have [*simp*]: ‹$N \propto C ! K' \in$ *set* (*unwatched-l* $(N \propto C)$)›
using *K′-le K′-2* **by** (*auto simp*: *set-drop-conv S*)
have [*simp*]: ‹$- N \propto C ! K' \notin$ *lits-of-l M* ›
using *n-d K′-M* **by** (*auto simp*: *S Decided-Propagated-in-iff-in-lits-of-l*
dest: *no-dup-consistentD*)

have *irred-init*: ‹*irred N C* $\Longrightarrow (N \propto C,$ *True*$) \in\#$ *init-clss-l N*›
using *C-N-U* **by** (*auto simp*: *S*)
have *init-unchanged*: ‹*update-clauses*
({#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. $x \in\#$ *init-clss-l N*#},
{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. $x \in\#$ *learned-clss-l N*#})
(*TWL-Clause* (*mset* (*watched-l* $(N \propto C)$)) (*mset* (*unwatched-l* $(N \propto C)$))) *L*
$(N \propto C ! K')$
({#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. $x \in\#$ *init-clss-l* $(N(C \hookrightarrow$ *swap* $(N \propto C)$ *i* $K'))$#},
{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. $x \in\#$ *learned-clss-l* $(N(C \hookrightarrow$ *swap* $(N \propto C)$ *i* $K'))$#})›
proof (*cases* ‹*irred N C*›)
case *J-NE*: *True*
have *L-L′-UW-N*: ‹$C' \in\#$ *init-clss-lf N*›
using *C-N-U J-NE* **unfolding** *take-set*
by (*auto simp*: *S ran-m-def*)

let *?UW* = ‹*unwatched-l* C'›
have *TWL-L-L′-UW-N*: ‹*TWL-Clause* {#*?L*, *?L′*#} (*mset ?UW*) $\in\#$ *twl-clause-of* ‘# *init-clss-lf* *N*›
using *imageI*[*OF L-L′-UW-N*, *of twl-clause-of*] *watched-C′* **by** *force*
let *?k′* = ‹*the K* − *2*›
have ‹*?k′* < *length* (*unwatched-l* C')›
using *K′-le two-le-length-C K′-2* **by** (*auto simp*: *S*)
then have *H0*: ‹*TWL-Clause* {#*?UW* ! *?k′*, *?L′*#} (*mset* (*list-update ?UW ?k′ ?L*)) =
update-clause (*TWL-Clause* {#*?L*, *?L′*#} (*mset ?UW*)) *?L* (*?UW* ! *?k′*)›
by (*auto simp*: *mset-update*)

have *H3*: ‹{#*L*, C' ! (*Suc 0* − *i*)#} = *mset* (*watched-l* $(N \propto C)$)›
using *K′-2 K′-le* ‹$C > 0$› *C′-i* **by** (*auto simp*: *S take-2-if C-N-U nth-tl i-def*)
have *H4*: ‹*mset* (*unwatched-l* C') = *mset* (*unwatched-l* $(N \propto C)$)›
by (*auto simp*: *S take-2-if C-N-U nth-tl*)

let *?New-C* = ‹(*TWL-Clause* {#*L*, C' ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l* C')))›

have *wo*: $a = a' \Longrightarrow b = b' \Longrightarrow L = L' \Longrightarrow K = K' \Longrightarrow c = c' \Longrightarrow$
update-clauses a K L b c $\Longrightarrow$
update-clauses a' K' L' b' c' **for** *a* a' *b* b' *K L* K' L' *c* c'
by *auto*
have [*simp*]: ‹$C' \in$ *fst* ‘ {*a*. $a \in\#$ *ran-m N* $\wedge$ *snd a*} $\longleftrightarrow$ *irred N C*›
using *C-N-U J-NE* **unfolding** C' *S ran-m-def*

by *auto*
have *C′-ran-N*: ‹(*C′*, *True*) ∈# *ran-m N*›
using *C-N-U J-NE* **unfolding** *C′ S S*
by *auto*
have *upd*: ‹*update-clauses*
(*twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*)
(*TWL-Clause* {#*C′* ! *i*, *C′* ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l C′*))) (*C′* ! *i*) (*C′* ! *the K*)
(*add-mset* (*update-clause* (*TWL-Clause* {#*C′* ! *i*, *C′* ! (*Suc 0* − *i*)#}
(*mset* (*unwatched-l C′*))) (*C′* ! *i*) (*C′* ! *the K*))
(*remove1-mset*
(*TWL-Clause* {#*C′* ! *i*, *C′* ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l C′*)))
(*twl-clause-of* ‘# *init-clss-lf N*)), *twl-clause-of* ‘# *learned-clss-lf N*)›
by (*rule update-clauses.intros*(*1*)[*OF TWL-L-L′-UW-N*])
have *K1*: ‹*mset* (*watched-l* (*swap* (*N*∝*C*) *i K′*)) = {#*N*∝*C*!*K′*, *N*∝*C*!(*1* − *i*)#}›
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
take-2-if swap-def i-def)
have *K2*: ‹*mset* (*unwatched-l* (*swap* (*N*∝*C*) *i K′*)) = *add-mset* (*N*∝*C* ! *i*)
(*remove1-mset* (*N*∝*C* ! *K′*) (*mset* (*unwatched-l* (*N*∝*C*))))›
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if mset-update*
take-2-if swap-def i-def drop-upd-irrelevant drop-Suc drop-update-swap)
have *K3*: ‹*mset* (*watched-l* (*N*∝*C*)) = {#*N*∝*C*!*i*, *N*∝*C*!(*1* − *i*)#}›
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
take-2-if swap-def i-def)

show *?thesis*
apply (*rule wo*[*OF - - - - - upd*])
subgoal by *auto*
subgoal by (*auto simp*: *S*)
subgoal by *auto*
subgoal unfolding *S H3*[*symmetric*] *H4*[*symmetric*] **by** *auto*
subgoal
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C K1 K2 K3 C′-ran-N*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
learned-clss-l-mapsto-upd-irrel)
done
next
assume *J-NE*: ‹¬*irred N C*›
have *L-L′-UW-N*: ‹*C′* ∈# *learned-clss-lf N*›
using *C-N-U J-NE* **unfolding** *take-set*
by (*auto simp*: *S ran-m-def*)

let *?UW* = ‹*unwatched-l C′*›
have *TWL-L-L′-UW-N*: ‹*TWL-Clause* {#*?L*, *?L′*#} (*mset ?UW*) ∈# *twl-clause-of* ‘# *learned-clss-lf N*›
using *imageI*[*OF L-L′-UW-N*, *of twl-clause-of*] *watched-C′* **by** *force*
let *?k′* = ‹*the K* − *2*›
have ‹*?k′* < *length* (*unwatched-l C′*)›
using *K′-le two-le-length-C K′-2* **by** (*auto simp*: *S*)
then have *H0*: ‹*TWL-Clause* {#*?UW* ! *?k′*, *?L′*#} (*mset* (*list-update ?UW ?k′ ?L*)) =
update-clause (*TWL-Clause* {#*?L*, *?L′*#} (*mset ?UW*)) *?L* (*?UW* ! *?k′*)›
by (*auto simp*: *mset-update*)

have *H3*: ‹{#*L*, *C′* ! (*Suc 0* − *i*)#} = *mset* (*watched-l* (*N* ∝ *C*))›

using *K′-2 K′-le* ‹$C > 0$› *C′-i* **by** (*auto simp*: *S take-2-if C-N-U nth-tl i-def*)
have *H4*: ‹*mset* (*unwatched-l C′*) = *mset* (*unwatched-l* ($N \propto C$))›
by (*auto simp*: *S take-2-if C-N-U nth-tl*)

let *?New-C* = ‹(*TWL-Clause* {#*L*, *C′* ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l C′*)))›

have *wo*: $a = a' \Longrightarrow b = b' \Longrightarrow L = L' \Longrightarrow K = K' \Longrightarrow c = c' \Longrightarrow$
update-clauses a K L b c $\Longrightarrow$
update-clauses a′ K′ L′ b′ c′ **for** *a a′ b b′ K L K′ L′ c c′*
by *auto*
have [*simp*]: ‹$C' \in$ *fst* ‘ {*a*. *a* ∈# *ran-m N* ∧ ¬*snd a*} ⟷ ¬*irred N C*›
using *C-N-U J-NE* **unfolding** *C′ S ran-m-def*
by *auto*
have *C′-ran-N*: ‹(*C′*, *False*) ∈# *ran-m N*›
using *C-N-U J-NE* **unfolding** *C′ S S*
by *auto*
have *upd*: ‹*update-clauses*
(*twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*)
(*TWL-Clause* {#*C′* ! *i*, *C′* ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l C′*))) (*C′* ! *i*)
(*C′* ! *the K*)
(*twl-clause-of* ‘# *init-clss-lf N*,
add-mset
(*update-clause*
(*TWL-Clause* {#*C′* ! *i*, *C′* ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l C′*))) (*C′* ! *i*)
(*C′* ! *the K*))
(*remove1-mset*
(*TWL-Clause* {#*C′* ! *i*, *C′* ! (*Suc 0* − *i*)#} (*mset* (*unwatched-l C′*)))
(*twl-clause-of* ‘# *learned-clss-lf N*)))
›
by (*rule update-clauses.intros*(*2*)[*OF TWL-L-L′-UW-N*])
have *K1*: ‹*mset* (*watched-l* (*swap* ($N \propto C$) *i K′*)) = {#$N \propto C$!*K′*, $N \propto C$!(*1* − *i*)#}›
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
take-2-if swap-def i-def)
have *K2*: ‹*mset* (*unwatched-l* (*swap* ($N \propto C$) *i K′*)) = *add-mset* ($N \propto C$! *i*)
(*remove1-mset* ($N \propto C$! *K′*) (*mset* (*unwatched-l* ($N \propto C$))))›
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if mset-update*
take-2-if swap-def i-def drop-upd-irrelevant drop-Suc drop-update-swap)
have *K3*: ‹*mset* (*watched-l* ($N \propto C$)) = {#$N \propto C$!*i*, $N \propto C$!(*1* − *i*)#}›
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C*
by (*auto simp*: *init-clss-l-mapsto-upd S image-mset-remove1-mset-if*
take-2-if swap-def i-def)

show *?thesis*
apply (*rule wo*[*OF - - - - - upd*])
subgoal by *auto*
subgoal by (*auto simp*: *S*)
subgoal by *auto*
subgoal unfolding *S H3*[*symmetric*] *H4*[*symmetric*] **by** *auto*
subgoal
using *J-NE C-N-U C′ K′-2 K′-le two-le-length-C K1 K2 K3 C′-ran-N*
by (*auto simp*: *learned-clss-l-mapsto-upd S image-mset-remove1-mset-if*
init-clss-l-mapsto-upd-irrel)
done
qed

have ‹*distinct-mset WS*›
  by (*metis* (*full-types*) *WS′-def WS′-def′ add-inv twl-list-invs-def*)
then have [*simp*]: ‹$C \notin\# WS'$›
   by (*auto simp*: *WS′-def′*)
have *H*: ‹{#(*L*, *TWL-Clause*
       (*mset* (*watched-l*
              (*fst* (*the* (*if* $C = x$ *then Some* (*swap* ($N \propto C$) *i* K', *irred N C*)
                        else fmlookup N x)))))
       (*mset* (*unwatched-l*
              (*fst* (*the* (*if* $C = x$ *then Some* (*swap* ($N \propto C$) *i* K', *irred N C*)
                        else fmlookup N x)))))). $x \in\# WS'$#} =
 {#(*L*, *TWL-Clause* (*mset* (*watched-l* ($N \propto x$))) (*mset* (*unwatched-l* ($N \propto x$)))). $x \in\# WS'$#}›
  by (*rule image-mset-cong*) *auto*
have [*simp*]: ‹*Propagated La C* $\notin$ *set M*› **for** *La*
proof (*rule ccontr*)
  assume *H*:‹¬ *?thesis*›
  then have ‹*length* ($N \propto C$) $>$ *2* $\Longrightarrow$ *La* = $N \propto C$! *0*› **and**
    ‹*La* $\in$ *set* (*watched-l* ($N \propto C$))›
    using *add-inv C-N-U two-le-length-C*
    unfolding *twl-list-invs-def S* **by** *auto*
  moreover have ‹*La* $\in$ *lits-of-l M*›
    using *H* **by** (*force simp*: *lits-of-def*)
  ultimately show *False*
    using *L′ L′-M SS′ uL-M n-d K′-2 K′-le*
    by (*auto simp*: *S i-def dest*: *no-dup-consistentD split*: *if-splits*)
qed
have *A*: ‹*?update-clss* = *do* {*let* $x = N \propto C$! K';
     if $x \in$ *lits-of-l* (*get-trail-l* (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*))
    then RETURN (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*)
    else update-clause-l C
          (*if get-clauses-l* (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*) $\propto$
              C !
              0 =
              L
           then 0 else 1)
          (*the K*) (*set-clauses-to-update-l* (*remove1-mset C* (*clauses-to-update-l S*)) *S*)}›
  unfolding *i-def*
  by (*auto simp add*: *S polarity-def dest*: *in-lits-of-l-defined-litD*)
have *alt-defs*: ‹$C' = N \propto C$›
  unfolding *C′ S* **by** *auto*
have *list-invs-blit*: ‹*twl-list-invs* (*M*, *N*, *D*, *NE*, *UE*, *WS′*, *Q*)›
  using *add-inv C-N-U two-le-length-C*
  unfolding *twl-list-invs-def*
  by (*auto dest*: *in-diffD simp*: *S WS′-def′*)
have ‹*twl-list-invs* (*M*, *N*($C \hookrightarrow$ *swap* ($N \propto C$) *i* K'), *D*, *NE*, *UE*, *WS′*, *Q*)›
  using *add-inv C-N-U two-le-length-C*
  unfolding *twl-list-invs-def*
  by (*auto dest*: *in-diffD simp*: *set-conflicting-def*
  set-conflict-l-def mset-take-mset-drop-mset′ S WS′-def′
  dest!: *mset-eq-setD*)
moreover have ‹(*M*, *x*) $\in$ *convert-lits-l N* (*NE* + *UE*) $\Longrightarrow$
    (*M*, *x*) $\in$ *convert-lits-l* (*N*($C \hookrightarrow$ *swap* ($N \propto C$) *i* K')) (*NE* + *UE*)› **for** *x*
  apply (*rule convert-lits-l-extend-mono*)
  by *auto*
ultimately show *?thesis*
  apply (*cases S′*)

unfolding *update-clauseS-def*
apply (*clarsimp simp only*: *clauses-to-update.simps set-clauses-to-update.simps*)
apply (*subst A*)
apply *refine-vcg*
subgoal unfolding C' *S* **by** *auto*
subgoal using L'*-M* SS' K'*-M* **unfolding** C' *S* **by** (*auto simp*: *twl-st-l-def*)
subgoal using L'*-M* SS' K'*-M* **unfolding** C' *S* **by** (*auto simp*: *twl-st-l-def*)
subgoal using L'*-M* SS' K'*-M add-inv list-invs-blit* **unfolding** C' *S*
by (*auto simp*: *twl-st-l-def* WS'*-def'*)
subgoal
using SS' *init-unchanged* **unfolding** *i-def*[*symmetric*] *get-clauses-l-set-clauses-to-update-l*
by (*auto simp*: *S update-clause-l-def update-clauseS-def twl-st-l-def* WS'*-def'*
RETURN-SPEC-refine RES-RES-RETURN-RES RETURN-def RES-RES2-RETURN-RES H
intro!: *RES-refine exI*[*of* - ‹$N \propto C$! *the K*›])
done
qed
have *H*: ‹*?A* $\leq \Downarrow$ {(*S*, S'). (*S*, S') $\in$ *twl-st-l* (*Some L*) $\wedge$ *twl-list-invs S*} *?B*›
unfolding *unit-propagation-inner-loop-body-l-def unit-propagation-inner-loop-body-def*
option.case-eq-if find-unwatched-l-def
apply (*rewrite at* ‹*let* - = *if* - ! - = -*then* - *else* - *in* -› *Let-def*)
apply (*rewrite at* ‹*let* - = *polarity* - - *in* -› *Let-def*)
apply (*refine-vcg*
bind-refine-spec[**where** M' = ‹*RETURN* (*polarity* - -)›, *OF* - *polarity-spec*]
case-prod-bind[*of* - ‹*If* - -›]; *remove-dummy-vars*)
subgoal by (*rule pre-inv*)
subgoal unfolding C' *clause-twl-clause-of* **by** *auto*
subgoal using SS' **by** (*auto simp*: *polarity-def Decided-Propagated-in-iff-in-lits-of-l*)
subgoal by (*rule upd-rel*)
subgoal
using *mset-watched-C* **by** (*auto simp*: *i-def*)
subgoal for L'
using *assms* **by** (*auto simp*: *polarity-def Decided-Propagated-in-iff-in-lits-of-l*)
subgoal by (*rule upd-rel*)
subgoal using SS' **by** *auto*
subgoal using SS' **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l*
polarity-def)
subgoal by (*rule confl-rel*)
subgoal unfolding *i-def*[*symmetric*] *i-def'*[*symmetric*] **by** (*rule propa-rel*)
subgoal by *auto*
subgoal for L' *K* **unfolding** *i-def*[*symmetric*] *i-def'*[*symmetric*]
by (*rule update-clause-rel*)
done
have *D-None*: ‹*get-conflict-l S* = *None*›
using *confl* SS' **by** (*cases* ‹*get-conflict-l S*›) (*auto simp*: *S* WS'*-def'*)
have $*$: ‹*unit-propagation-inner-loop-body* (C' ! *i*) (*twl-clause-of* C')
(*set-clauses-to-update* (*remove1-mset* (C' ! *i*, *twl-clause-of* C') (*clauses-to-update* S')) S')
$\leq$ *SPEC* ($\lambda S''$. *twl-struct-invs* S'' $\wedge$
twl-stgy-invs S'' $\wedge$
cdcl-twl-cp** S' S'' $\wedge$
(S'', S') $\in$ *measure* (*size* $\circ$ *clauses-to-update*))›
apply (*rule unit-propagation-inner-loop-body*(*1*)[*of* S' ‹C' ! *i*› ‹*twl-clause-of* C'›])
using *imageI*[*OF WS*, *of* ‹(λj. (*L*, *twl-clause-of* ($N \propto j$)))›]
struct-invs stgy-inv C-N-U WS SS' *D-None* **by** *auto*
have H': ‹*?B* $\leq$ *SPEC* ($\lambda S'$. *twl-stgy-invs* S' $\wedge$ *twl-struct-invs* S')›
using $*$
by (*simp add*: *weaken-SPEC*)

have ‹*?A*
  ≤ ⇓ {(*S*, *S'*). ((*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S*) ∧
      (*twl-stgy-invs S'* ∧ *twl-struct-invs S'*)}
    ?B›
  apply (*rule refine-add-invariants*)
  apply (*rule H'*)
  by (*rule H*)
then show *?thesis* **by** *simp*
qed

lemma *unit-propagation-inner-loop-body-l2*:
assumes
  SS': ‹(*S*, *S'*) ∈ *twl-st-l* (*Some L*)› **and**
  WS: ‹*C* ∈# *clauses-to-update-l S*› **and**
  struct-invs: ‹*twl-struct-invs S'*› **and**
  add-inv: ‹*twl-list-invs S*› **and**
  stgy-inv: ‹*twl-stgy-invs S'*›
shows
  ‹(*unit-propagation-inner-loop-body-l L C*
      (*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*),
    unit-propagation-inner-loop-body L (*twl-clause-of* (*get-clauses-l S* ∝ *C*))
      (*set-clauses-to-update*
        (*remove1-mset* (*L*, *twl-clause-of* (*get-clauses-l S* ∝ *C*))
        (*clauses-to-update S'*)) *S'*))
  ∈ ⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'* ∧
      twl-struct-invs S'}⟩*nres-rel*›
using *unit-propagation-inner-loop-body-l*[*OF assms*]
by (*auto simp*: *nres-rel-def*)

This a work around equality: it allows to instantiate variables that appear in goals by hand in a reasonable way (*rule*\-*tac I=x in EQI*).

definition *EQ* **where**
  [*simp*]: ‹*EQ* = (=)›

lemma *EQI*: *EQ I I*
  by *auto*

lemma *unit-propagation-inner-loop-body-l-unit-propagation-inner-loop-body*:
‹*EQ L'' L''* ⟹
  (*uncurry2 unit-propagation-inner-loop-body-l*, *uncurry2 unit-propagation-inner-loop-body*) ∈
    {(((*L*, *C*), *S0*), ((*L'*, *C'*), *S0'*)). ∃ *S S'*. *L* = *L'* ∧ *C'* = (*twl-clause-of* (*get-clauses-l S* ∝ *C*)) ∧
      S0 = (*set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*) ∧
      S0' = (*set-clauses-to-update*
        (*remove1-mset* (*L*, *twl-clause-of* (*get-clauses-l S* ∝ *C*))
        (*clauses-to-update S'*)) *S'*) ∧
     (*S*, *S'*) ∈ *twl-st-l* (*Some L*) ∧ *L* = *L''* ∧
     C ∈# *clauses-to-update-l S* ∧ *twl-struct-invs S'* ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'*} $\to_f$
    ⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l* (*Some L''*) ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'* ∧
      twl-struct-invs S'}⟩*nres-rel*›
apply (*intro frefI nres-relI*)
using *unit-propagation-inner-loop-body-l*
by *fastforce*

definition *select-from-clauses-to-update* :: ‹*'v twl-st-l* ⇒ (*'v twl-st-l* × *nat*) *nres*› **where**
‹*select-from-clauses-to-update S* = *SPEC* (λ(*S'*, *C*). *C* ∈# *clauses-to-update-l S* ∧
  S' = *set-clauses-to-update-l* (*clauses-to-update-l S* − {#*C*#}) *S*)›

definition *unit-propagation-inner-loop-l-inv* **where**
‹*unit-propagation-inner-loop-l-inv* $L = (\lambda(S, n).$
$(\exists S'. (S, S') \in$ *twl-st-l* (*Some* L) $\wedge$ *twl-struct-invs* $S' \wedge$ *twl-stgy-invs* $S' \wedge$
twl-list-invs $S \wedge$ (*clauses-to-update* $S' \neq \{\#\} \vee n > 0 \longrightarrow$ *get-conflict* $S' =$ *None*) $\wedge$
$-L \in$ *lits-of-l* (*get-trail-l* S)))›

definition *unit-propagation-inner-loop-body-l-with-skip* **where**
‹*unit-propagation-inner-loop-body-l-with-skip* $L = (\lambda(S, n).$ *do* {
ASSERT (*clauses-to-update-l* $S \neq \{\#\} \vee n > 0$);
ASSERT(*unit-propagation-inner-loop-l-inv* L (S, n));
$b \leftarrow$ *SPEC*($\lambda b.$ ($b \longrightarrow n > 0$) $\wedge$ ($\neg b \longrightarrow$ *clauses-to-update-l* $S \neq \{\#\}$));
if $\neg b$ *then do* {
ASSERT (*clauses-to-update-l* $S \neq \{\#\}$);
(S', C) $\leftarrow$ *select-from-clauses-to-update* S;
$T \leftarrow$ *unit-propagation-inner-loop-body-l* L C S';
RETURN (T, *if get-conflict-l* T = *None then* n *else* 0)
} *else RETURN* (S, $n-1$)
})›

definition *unit-propagation-inner-loop-l* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*unit-propagation-inner-loop-l* L S_0 = *do* {
$n \leftarrow$ *SPEC*(λ-::*nat*. *True*);
(S, n) $\leftarrow$ $WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L}$
($\lambda(S, n).$ *clauses-to-update-l* $S \neq \{\#\} \vee n > 0$)
(*unit-propagation-inner-loop-body-l-with-skip* L)
(S_0, n);
RETURN S
}›

lemma *set-mset-clauses-to-update-l-set-mset-clauses-to-update-spec*:
assumes ‹(S, S') $\in$ *twl-st-l* (*Some* L)›
shows
‹*RES* (*set-mset* (*clauses-to-update-l* S)) $\leq \Downarrow \{(C, (L', C')).\ L' = L \wedge$
C' = *twl-clause-of* (*get-clauses-l* $S \propto C$)}
(*RES* (*set-mset* (*clauses-to-update* S')))›
proof −
obtain *M N D NE UE WS Q* **where**
S: ‹S = (M, N, D, *NE*, *UE*, *WS*, Q)›
by (*cases* S) *auto*
show *?thesis*
using *assms* **unfolding** *S* **by** (*auto simp add*: *Bex-def twl-st-l-def intro*!: *RES-refine*)
qed

lemma *refine-add-inv*:
fixes f :: ‹*'a* ⇒ *'a nres*› **and** f' :: ‹*'b* ⇒ *'b nres*› **and** h :: ‹*'b* ⇒ *'a*›
assumes
‹(f', f) $\in \{(S, S').\ S' = h\ S \wedge R\ S\} \rightarrow \langle\{(T, T').\ T' = h\ T \wedge P'\ T\}\rangle$ *nres-rel*›
(**is** ‹- $\in$ *?R* $\rightarrow \langle\{(T, T').$ *?H* $T\ T' \wedge P'\ T\}\rangle$ *nres-rel*›)
assumes
‹$\bigwedge S.\ R\ S \Longrightarrow f\ (h\ S) \leq$ *SPEC* ($\lambda T.\ Q\ T$)›
shows
‹(f', f) $\in$ *?R* $\rightarrow \langle\{(T, T').$ *?H* $T\ T' \wedge P'\ T \wedge Q\ (h\ T)\}\rangle$ *nres-rel*›
using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
by *fastforce*

lemma *refine-add-inv-generalised*:
fixes f :: ‹$'a \Rightarrow 'b$ *nres*› **and** f' :: ‹$'c \Rightarrow 'd$ *nres*›
assumes
‹$(f', f) \in A \to_f \langle B\rangle$ *nres-rel*›
assumes
‹$\bigwedge S\ S'.\ (S, S') \in A \Longrightarrow f\ S' \leq RES\ C$›
shows
‹$(f', f) \in A \to_f \langle\{(T, T').\ (T, T') \in B \wedge T' \in C\}\rangle$ *nres-rel*›
using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail fref-param1*[*symmetric*]
by *fastforce*

lemma *refine-add-inv-pair*:
fixes f :: ‹$'a \Rightarrow ('c \times 'a)$ *nres*› **and** f' :: ‹$'b \Rightarrow ('c \times 'b)$ *nres*› **and** h :: ‹$'b \Rightarrow 'a$›
assumes
‹$(f', f) \in \{(S, S').\ S' = h\ S \wedge R\ S\} \to \langle\{(S, S').\ (fst\ S' = h'\ (fst\ S) \wedge$
$snd\ S' = h\ (snd\ S)) \wedge P'\ S\}\rangle$ *nres-rel*› (**is** ‹$\text{-} \in ?R \to \langle\{(S, S').\ ?H\ S\ S' \wedge P'\ S\}\rangle$ *nres-rel*›)
assumes
‹$\bigwedge S.\ R\ S \Longrightarrow f\ (h\ S) \leq SPEC\ (\lambda T.\ Q\ (snd\ T))$›
shows
‹$(f', f) \in ?R \to \langle\{(S, S').\ ?H\ S\ S' \wedge P'\ S \wedge Q\ (h\ (snd\ S))\}\rangle$ *nres-rel*›
using *assms* **unfolding** *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*
by *fastforce*

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

lemma *cdcl-twl-cp-in-trail-stays-in*:
‹$cdcl\text{-}twl\text{-}cp^{**}\ S'\ aa \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\ S') \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\ aa)$›
by (*induction rule*: *rtranclp-induct*)
(*auto elim*!: *cdcl-twl-cpE*)

lemma *cdcl-twl-cp-in-trail-stays-in-l*:
‹$(x2, S') \in twl\text{-}st\text{-}l\ (Some\ x1) \Longrightarrow cdcl\text{-}twl\text{-}cp^{**}\ S'\ aa \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ x2) \Longrightarrow$
$(a, aa) \in twl\text{-}st\text{-}l\ (Some\ x1) \Longrightarrow -\ x1 \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ a)$›
using *cdcl-twl-cp-in-trail-stays-in*[*of* S' *aa* ‹*x1*›]
by (*auto simp*: *twl-st twl-st-l*)

lemma *unit-propagation-inner-loop-l*:
‹(*uncurry unit-propagation-inner-loop-l*, *unit-propagation-inner-loop*) $\in$
$\{((L, S), S').\ (S, S') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}struct\text{-}invs\ S' \wedge$
$twl\text{-}stgy\text{-}invs\ S' \wedge twl\text{-}list\text{-}invs\ S \wedge -L \in lits\text{-}of\text{-}l\ (get\text{-}trail\text{-}l\ S)\} \to_f$
$\langle\{(T, T').\ (T, T') \in twl\text{-}st\text{-}l\ None \wedge clauses\text{-}to\text{-}update\text{-}l\ T = \{\#\} \wedge$
$twl\text{-}list\text{-}invs\ T \wedge twl\text{-}struct\text{-}invs\ T' \wedge twl\text{-}stgy\text{-}invs\ T'\}\rangle$ *nres-rel*›
(**is** ‹$?unit\text{-}prop\text{-}inner \in ?A \to_f \langle ?B\rangle nres\text{-}rel$›)
proof –
have *SPEC-remove*: ‹*select-from-clauses-to-update* S
$\leq \Downarrow \{((T', C), C').$
$(T', set\text{-}clauses\text{-}to\text{-}update\ (clauses\text{-}to\text{-}update\ S'' - \{\#C'\#\})\ S'') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge$
$T' = set\text{-}clauses\text{-}to\text{-}update\text{-}l\ (clauses\text{-}to\text{-}update\text{-}l\ S - \{\#C\#\})\ S \wedge$
$C' \in\#\ clauses\text{-}to\text{-}update\ S'' \wedge$
$C \in\#\ clauses\text{-}to\text{-}update\text{-}l\ S \wedge$
$snd\ C' = twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C)\}$
$(SPEC\ (\lambda C.\ C \in\#\ clauses\text{-}to\text{-}update\ S''))$›
if ‹$(S, S'') \in \{(T, T').\ (T, T') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}list\text{-}invs\ T\}$›

for *S* :: ‹′*v twl-st-l*› **and** *S*′′ *L*
using *that* **unfolding** *select-from-clauses-to-update-def*
by (*auto simp*: *conc-fun-def image-mset-remove1-mset-if twl-st-l-def*)
show *?thesis*
unfolding *unit-propagation-inner-loop-l-def unit-propagation-inner-loop-def uncurry-def*
unit-propagation-inner-loop-body-l-with-skip-def
apply (*intro frefI nres-relI*)
subgoal for *LS S*′
apply (*rewrite* **in** ‹*let* - = *set-clauses-to-update* - - *in* -› *Let-def*)
apply (*refine-vcg set-mset-clauses-to-update-l-set-mset-clauses-to-update-spec*
WHILEIT-refine-genR[**where**
R = ‹{(T, T'). (T, T') ∈ *twl-st-l None* ∧ *twl-list-invs* T ∧ *clauses-to-update-l* T = {#}
∧ *twl-struct-invs* T' ∧ *twl-stgy-invs* T'}
$\times_f$ *nat-rel*› **and**
R' = ‹{(T, T'). (T, T') ∈ *twl-st-l* (*Some* (*fst LS*)) ∧ *twl-list-invs* T}
$\times_f$ *nat-rel*›]
unit-propagation-inner-loop-body-l-unit-propagation-inner-loop-body[*THEN fref-to-Down-curry2*]
SPEC-remove;
remove-dummy-vars)
subgoal by *simp*
subgoal for *x1 x2 n na x x*′ **unfolding** *unit-propagation-inner-loop-l-inv-def*
apply (*case-tac x*; *case-tac x*′)
apply (*simp only*: *prod.simps*)
by (*rule exI*[*of* - ‹*fst x*′›]) (*auto intro*: *cdcl-twl-cp-in-trail-stays-in-l*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
apply (*subst* (*asm*) *prod-rel-iff*)
apply *normalize-goal*
apply *assumption*
apply (*rule-tac I=x1* **in** *EQI*)
subgoal for *x1 x2 n na x1a x2a x1b x2b b ba x1c x2c x1d x2d*
apply (*subst in-pair-collect-simp*)
apply (*subst prod.case*)+
apply (*rule-tac x* = *x1b* **in** *exI*)
apply (*rule-tac x* = *x1a* **in** *exI*)
apply (*intro conjI*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

done
qed

definition *clause-to-update* :: ‹'*v literal* ⇒ '*v twl-st-l* ⇒ '*v clauses-to-update-l*›**where**
‹*clause-to-update L S* =
filter-mset
(λC::*nat*. $L \in$ *set* (*watched-l* (*get-clauses-l S* ∝ *C*)))
(*dom-m* (*get-clauses-l S*))›

lemma *distinct-mset-clause-to-update*: ‹*distinct-mset* (*clause-to-update L C*)›
unfolding *clause-to-update-def*
apply (*rule distinct-mset-filter*)
using *distinct-mset-dom* **by** *blast*

lemma *in-clause-to-updateD*: ‹$b \in\#$ *clause-to-update* L' $T \Longrightarrow b \in\#$ *dom-m* (*get-clauses-l T*)›
by (*auto simp*: *clause-to-update-def*)

lemma *in-clause-to-update-iff*:
‹$C \in\#$ *clause-to-update L S* ⟷
$C \in\#$ *dom-m* (*get-clauses-l S*) ∧ $L \in$ *set* (*watched-l* (*get-clauses-l S* ∝ *C*))›
by (*auto simp*: *clause-to-update-def*)

definition *select-and-remove-from-literals-to-update* :: ‹'*v twl-st-l* ⇒
('*v twl-st-l* × '*v literal*) *nres*› **where**
‹*select-and-remove-from-literals-to-update S* = *SPEC*(λ(S', L). $L \in\#$ *literals-to-update-l S* ∧
S' = *set-clauses-to-update-l* (*clause-to-update L S*)
(*set-literals-to-update-l* (*literals-to-update-l S* − {#*L*#}) *S*))›

definition *unit-propagation-outer-loop-l-inv* **where**
‹*unit-propagation-outer-loop-l-inv S* ⟷
(∃ S'. (S, S') ∈ *twl-st-l None* ∧ *twl-struct-invs* S' ∧ *twl-stgy-invs* S' ∧
clauses-to-update-l S = {#})›

definition *unit-propagation-outer-loop-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*unit-propagation-outer-loop-l* S_0 =
$WHILE_T{}^{unit\text{-}propagation\text{-}outer\text{-}loop\text{-}l\text{-}inv}$
(λS. *literals-to-update-l S* ≠ {#})
(λS. *do* {
ASSERT(*literals-to-update-l S* ≠ {#});
(S', L) ← *select-and-remove-from-literals-to-update S*;
unit-propagation-inner-loop-l L S'
})
(S_0 :: '*v twl-st-l*)
›

lemma *watched-twl-clause-of-watched*: ‹*watched* (*twl-clause-of x*) = *mset* (*watched-l x*)›
by (*cases x*) *auto*

lemma *twl-st-of-clause-to-update*:
assumes
TT': ‹(T, T') ∈ *twl-st-l None*› **and**
‹*twl-struct-invs* T'›
shows
‹(*set-clauses-to-update-l*
(*clause-to-update* L' T)
(*set-literals-to-update-l* (*remove1-mset* L' (*literals-to-update-l T*)) *T*),

set-clauses-to-update
  (*Pair L′ ‘# {#C ∈# get-clauses T′. L′ ∈# watched C#}*)
  (*set-literals-to-update* (*remove1-mset L′* (*literals-to-update T′*))
   T′))
∈ *twl-st-l* (*Some L′*)›
proof −
 obtain *M N D NE UE WS Q* **where**
  T: ‹*T* = (*M*, *N*, *D* , *NE*, *UE*, *WS*, *Q*)›
  by (*cases T*) *auto*

 have
  ‹{#(*L′*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *x*)))
     (*mset* (*unwatched-l* (*N* ∝ *x*)))).
   x ∈# {#*C* ∈# *dom-m N*. *L′* ∈ *set* (*watched-l* (*N* ∝ *C*))#}#} =
  Pair L′ ‘#
   {#*C* ∈# {#*TWL-Clause* (*mset* (*watched-l x*)) (*mset* (*unwatched-l x*)). *x* ∈# *init-clss-lf N*#} +
     {#*TWL-Clause* (*mset* (*watched-l x*)) (*mset* (*unwatched-l x*)). *x* ∈# *learned-clss-lf N*#}.
   L′ ∈# *watched C*#}›
  (**is** ‹{#(*L′*, *?C x*). *x* ∈# *?S*#} = *Pair L′* ‘# *?C′*›)
 proof −
  have *H*: ‹{#*f* (*N* ∝ *x*). *x* ∈# {#*x* ∈# *dom-m N*. *P* (*N* ∝ *x*)#}#} =
    {#*f* (*fst x*). *x* ∈# {#*C* ∈# *ran-m N*. *P* (*fst C*)#}#}› **for** *P* **and** *f* :: ‹*′a literal list* ⇒ *′b*›
    unfolding *ran-m-def image-mset-filter-swap2* **by** *auto*

  have *H*: ‹{#*f* (*N*∝*x*). *x* ∈# *?S*#} =
    {#*f* (*fst x*). *x* ∈# {#*C* ∈# *init-clss-l N*. *L′* ∈ *set* (*watched-l* (*fst C*))#}#} +
    {#*f* (*fst x*). *x* ∈# {#*C* ∈# *learned-clss-l N*. *L′* ∈ *set* (*watched-l* (*fst C*))#}#}›
   for *f* :: ‹*′a literal list* ⇒ *′b*›
   unfolding *image-mset-union*[*symmetric*] *filter-union-mset*[*symmetric*]
   apply *auto*
   apply (*subst H*)
   ..

  have *L″*: ‹{#(*L′*, *?C x*). *x* ∈# *?S*#} = *Pair L′* ‘# {#*?C x*. *x* ∈# *?S*#}›
   by *auto*
  also have ‹. . . = *Pair L′* ‘# *?C′*›
   apply (*rule arg-cong*[*of - -* ‹(‘#) (*Pair L′*)›])
   unfolding *image-mset-union*[*symmetric*] *mset-append*[*symmetric*] *drop-Suc H*
   apply *simp*
   apply (*subst H*)
   unfolding *image-mset-union*[*symmetric*] *mset-append*[*symmetric*] *drop-Suc H*
    filter-union-mset[*symmetric*] *image-mset-filter-swap2*
   by *auto*
  finally show *?thesis* **.**
 qed
 then show *?thesis*
  using *TT′*
  by (*cases T′*) (*auto simp del*: *filter-union-mset*
    simp: *T split-beta clause-to-update-def twl-st-l-def*
    split: *if-splits*)
qed

lemma *twl-list-invs-set-clauses-to-update-iff*:
 assumes ‹*twl-list-invs T*›
 shows ‹*twl-list-invs* (*set-clauses-to-update-l WS* (*set-literals-to-update-l Q T*)) ⟷
  ((∀ *x*∈#*WS*. *case x of C* ⇒ *C* ∈# *dom-m* (*get-clauses-l T*)) ∧

distinct-mset WS)›
proof −
obtain *M N C NE UE WS Q* **where**
T: ‹*T* = (*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*)›
by (*cases T*) *auto*
show *?thesis*
using *assms*
unfolding *twl-list-invs-def T* **by** *auto*
qed

lemma *unit-propagation-outer-loop-l-spec*:
‹(*unit-propagation-outer-loop-l*, *unit-propagation-outer-loop*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'* ∧
twl-stgy-invs S' ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#} ∧
get-conflict-l S = *None*} $\rightarrow_f$
⟨{(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧
(*twl-list-invs T* ∧ *twl-struct-invs T'* ∧ *twl-stgy-invs T'* ∧
clauses-to-update-l T = {#}) ∧
literals-to-update T' = {#} ∧ *clauses-to-update T'* = {#} ∧
no-step cdcl-twl-cp T'}⟩ *nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ *?I*› **is** ‹- ∈ - $\rightarrow_f$ ⟨*?B*⟩ *nres-rel*›)
proof −
have *H*:
‹*select-and-remove-from-literals-to-update x*
≤ ⇓ {((*S'*, *L'*), *L*). *L* = *L'* ∧ *S'* = *set-clauses-to-update-l* (*clause-to-update L x*)
(*set-literals-to-update-l* (*remove1-mset L* (*literals-to-update-l x*)) *x*)}
(*SPEC* (λ*L*. *L* ∈# *literals-to-update x'*))›
if ‹(*x*, *x'*) ∈ *twl-st-l None*› **for** *x* :: ‹*'v twl-st-l*› **and** *x'* :: ‹*'v twl-st*›
using *that* **unfolding** *select-and-remove-from-literals-to-update-def*
apply (*cases x*; *cases x'*)
unfolding *conc-fun-def* **by** (*clarsimp simp add*: *twl-st-l-def conc-fun-def*)
have *H'*: ‹*unit-propagation-outer-loop-l-inv T* ⟹
x2 ∈# *literals-to-update-l T* ⟹ − *x2* ∈ *lits-of-l* (*get-trail-l T*)›
for *S S' T T' L L' C x2*
by (*auto simp*: *unit-propagation-outer-loop-l-inv-def twl-st-l-def twl-struct-invs-def*)
have *H*:
‹(*unit-propagation-outer-loop-l*, *unit-propagation-outer-loop*) ∈*?R* $\rightarrow_f$
⟨{(*S*, *S'*).
(*S*, *S'*) ∈ *twl-st-l None* ∧
clauses-to-update-l S = {#} ∧
twl-list-invs S ∧
twl-struct-invs S' ∧
twl-stgy-invs S'}⟩ *nres-rel*›
unfolding *unit-propagation-outer-loop-l-def unit-propagation-outer-loop-def fref-param1*[*symmetric*]
apply (*refine-vcg unit-propagation-inner-loop-l*[*THEN fref-to-Down-curry-left*]
H)
subgoal by *simp*
subgoal unfolding *unit-propagation-outer-loop-l-inv-def* **by** *fastforce*
subgoal by *auto*
subgoal by *simp*
subgoal by *fast*
subgoal for *S S' T T' L L' C x2*
by (*auto simp add*: *twl-st-of-clause-to-update twl-list-invs-set-clauses-to-update-iff*
intro: *cdcl-twl-cp-twl-struct-invs cdcl-twl-cp-twl-stgy-invs*
distinct-mset-clause-to-update H'

dest: *in-clause-to-updateD*)
done
have *B*: ‹*?B* = {(*T*, *T'*). (*T*, *T'*) ∈ {(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧
twl-list-invs T ∧
twl-struct-invs T' ∧
twl-stgy-invs T' ∧ *clauses-to-update-l T* = {#} } ∧
T' ∈ {*T'*. *literals-to-update T'* = {#} ∧
clauses-to-update T' = {#} ∧
(∀ *S'*. ¬ *cdcl-twl-cp T' S'*)}}›
by *auto*
show *?thesis*
unfolding *B*
apply (*rule refine-add-inv-generalised*)
subgoal
using *H* **apply** −
apply (*match-spec*; (*match-fun-rel*; *match-fun-rel?*)+)
apply *blast+*
done
subgoal for *S S'*
apply (*rule weaken-SPEC*[*OF unit-propagation-outer-loop*[*of S'*]])
apply ((*solves auto*)+)[*4*]
using *no-step-cdcl-twl-cp-no-step-cdcl*$_W$*-cp* **by** *blast*
done
qed

lemma *get-conflict-l-get-conflict-state-spec*:
assumes ‹(*S*, *S'*) ∈ *twl-st-l None*› **and** ‹*twl-list-invs S*› **and** ‹*clauses-to-update-l S* = {#}›
shows ‹((*False*, *S*), (*False*, *S'*))
∈ {((*brk*, *S*), (*brk'*, *S'*)). *brk* = *brk'* ∧ (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}}›
using *assms* **by** *auto*

fun *lit-and-ann-of-propagated* **where**
‹*lit-and-ann-of-propagated* (*Propagated L C*) = (*L*, *C*)› |
‹*lit-and-ann-of-propagated* (*Decided -*) = *undefined*›
— we should never call the function in that context

definition *tl-state-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*tl-state-l* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). (*tl M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*))›

definition *resolve-cls-l'* :: ‹*'v twl-st-l* ⇒ *nat* ⇒ *'v literal* ⇒ *'v clause*› **where**
‹*resolve-cls-l' S C L* =
remove1-mset L (*remove1-mset* (−*L*) (*the* (*get-conflict-l S*) ∪# *mset* (*get-clauses-l S* ∝ *C*)))›

definition *update-confl-tl-l* :: ‹*nat* ⇒ *'v literal* ⇒ *'v twl-st-l* ⇒ *bool* × *'v twl-st-l*› **where**
‹*update-confl-tl-l* = (λ*C L* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
let D = *resolve-cls-l'* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) *C L in*
(*False*, (*tl M*, *N*, *Some D*, *NE*, *UE*, *WS*, *Q*)))›

definition *skip-and-resolve-loop-inv-l* **where**
‹*skip-and-resolve-loop-inv-l* S_0 *brk S* ⟷
(∃ *S'* S_0*'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ (S_0, S_0*'*) ∈ *twl-st-l None* ∧
skip-and-resolve-loop-inv S_0*'* (*brk*, *S'*) ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#} ∧
(¬*is-decided* (*hd* (*get-trail-l S*)) ⟶ *mark-of* (*hd*(*get-trail-l S*)) > *0*))›

definition *skip-and-resolve-loop-l* :: ⟨'v twl-st-l ⇒ 'v twl-st-l nres⟩ **where**
⟨*skip-and-resolve-loop-l* S_0 =
  do {
    ASSERT(*get-conflict-l* $S_0 \neq$ *None*);
    (-, S) ←
      $WHILE_T{}^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}inv\text{-}l\ S_0\ brk\ S}$
      (λ(*brk*, S). ¬*brk* ∧ ¬*is-decided* (*hd* (*get-trail-l* S)))
      (λ(-, S).
        do {
          let D' = *the* (*get-conflict-l* S);
          let (L, C) = *lit-and-ann-of-propagated* (*hd* (*get-trail-l* S));
          if $-L \notin\#$ D' *then*
            do {*RETURN* (*False*, *tl-state-l* S)}
          else
            if *get-maximum-level* (*get-trail-l* S) (*remove1-mset* ($-L$) D') = *count-decided* (*get-trail-l* S)
            then
              do {*RETURN* (*update-confl-tl-l* C L S)}
            else
              do {*RETURN* (*True*, S)}
        }
)
      (*False*, S_0);
    RETURN S
  }
⟩

context
begin

private lemma *skip-and-resolve-l-refines*:
⟨((*brkS*), *brk'S'*) ∈ {((*brk*, S), *brk'*, S'). *brk* = *brk'* ∧ (S, S') ∈ *twl-st-l None* ∧
    twl-list-invs S ∧ *clauses-to-update-l* S = {#}} ⟹
  brkS = (*brk*, S) ⟹ *brk'S'* = (*brk'*, S') ⟹
((*False*, *tl-state-l* S), *False*, *tl-state* S') ∈ {((*brk*, S), *brk'*, S'). *brk* = *brk'* ∧
    (S, S') ∈ *twl-st-l None* ∧ *twl-list-invs* S ∧ *clauses-to-update-l* S = {#}}⟩
by (*cases* S; *cases* ⟨*get-trail-l* S⟩)
 (*auto simp*: *twl-list-invs-def twl-st-l-def*
   resolve-cls-l-nil-iff tl-state-l-def tl-state-def dest: *convert-lits-l-tlD*)

private lemma *skip-and-resolve-skip-refine*:
assumes
  rel: ⟨((*brk*, S), *brk'*, S') ∈ {((*brk*, S), *brk'*, S'). *brk* = *brk'* ∧
      (S, S') ∈ *twl-st-l None* ∧ *twl-list-invs* S ∧ *clauses-to-update-l* S = {#}}⟩ **and**
  dec: ⟨¬ *is-decided* (*hd* (*get-trail* S'))⟩ **and**
  rel': ⟨((L, C), L', C') ∈ {((L, C), L', C'). L = L' ∧ C > *0* ∧
      C' = *mset* (*get-clauses-l* S ∝ C)}⟩ **and**
  LC: ⟨*lit-and-ann-of-propagated* (*hd* (*get-trail-l* S)) = (L, C)⟩ **and**
  tr: ⟨*get-trail-l* $S \neq$ []⟩ **and**
  struct-invs: ⟨*twl-struct-invs* S'⟩ **and**
  stgy-invs: ⟨*twl-stgy-invs* S'⟩ **and**
  lev: ⟨*count-decided* (*get-trail-l* S) > *0*⟩ **and**
  inv: ⟨*case* (*brk*, S) *of* (x, *xa*) ⇒ *skip-and-resolve-loop-inv-l S0 x xa*⟩
shows
 ⟨(*update-confl-tl-l* C L S, *False*,
   update-confl-tl (*Some* (*remove1-mset* (− L') (*the* (*get-conflict* S')) ∪# *remove1-mset* L' C')) S')
     ∈ {((*brk*, S), *brk'*, S').

brk = *brk'* ∧
(*S*, *S'*) ∈ *twl-st-l None* ∧
twl-list-invs S ∧
clauses-to-update-l S = {#}}›

proof –
obtain *M N D NE UE Q* **where** *S*: ‹*S* = (*Propagated L C* # *M*, *N*, *D*, *NE*, *UE*, {#}, *Q*)›
using *dec LC tr rel*
by (*cases S*; *cases* ‹*get-trail-l S*›; *cases* ‹*get-trail S'*›; *cases* ‹*hd* (*get-trail-l S*)›)
(*auto simp*: *twl-st-l-def*)
have *S'*: ‹(*S*, *S'*) ∈ *twl-st-l None*› **and** [*simp*]: ‹*L* = *L'*› **and**
C': ‹*C'* = *mset* (*get-clauses-l S* ∝ *C*)› **and**
[*simp*]: ‹*C* > *0*› ‹*C* ≠ *0*›**and**
invs-S: ‹*twl-list-invs S*›
using *rel rel'* **unfolding** *S* **by** *auto*
have *H*: ‹*L'* ∉# *the D* ⟹ *the D* ∪# {#*L'*, *aa*#} – {#*L'*, – *L'*#} =
the D ∪# {#*aa*#} – {#– *L'*#}›
‹*L'* ∉# *the D* ⟹ *the D* ∪# {#*aa*, *L'*#} – {#*L'*, – *L'*#} =
the D ∪# {#*aa*#} – {#– *L'*#}› **for** *aa*
by (*auto simp*: *add-mset-commute*)
have *H'*: ‹*a* ≠ –*L'* ⟹ *remove1-mset* (– *L'*) (*the D*) ∪# {#*a*#} =
remove1-mset (– *L'*) (*the D* ∪# {#*a*#})› **for** *a*
by (*auto simp*: *sup-union-right-if*
dest: *in-diffD multi-member-split*)
have ‹*D* ≠ *None*›
using *inv* **by** (*auto simp*: *skip-and-resolve-loop-inv-l-def S*
skip-and-resolve-loop-inv-def twl-st-l-def)
have ‹$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of S'*)› **and**
struct: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S'*)›
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast+*
moreover have ‹*Suc 0* ≤ *backtrack-lvl* ($state_W$*-of S'*)›
using *lev S'* **by** (*cases S*) (*auto simp*: *trail.simps twl-st-l-def*)
moreover have ‹*is-proped* ($cdcl_W$*-restart-mset.hd-trail* ($state_W$*-of S'*))›
using *dec tr S'* **by** (*cases* ‹*get-trail-l S*›)
(*auto simp*: *trail.simps is-decided-no-proped-iff twl-st-l-def*)
moreover have ‹*mark-of* ($cdcl_W$*-restart-mset.hd-trail* ($state_W$*-of S'*)) = *C'*›
using *dec S'* **unfolding** *C'* **by** (*cases* ‹*get-trail S'*›)
(*auto simp*: *S trail.simps twl-st-l-def*
convert-lit.simps)
ultimately have *False*: ‹*C* = *0* ⟹ *False*›
using *C'* $cdcl_W$*-restart-mset.hd-trail-level-ge-1-length-gt-1*[*of* ‹$state_W$*-of S'*›]
by (*auto simp*: *is-decided-no-proped-iff*)
then have *L*: ‹*length* (*N* ∝ *C*) > *2* ⟶ *L* = *N* ∝ *C* ! *0*› **and**
C-dom: ‹*C* ∈# *dom-m N*› **and**
L: ‹*L* ∈ *set*(*watched-l* (*N* ∝ *C*))›
using *invs-S*
unfolding *S C'* **by** (*auto simp*: *twl-list-invs-def*)
moreover {
have ‹*twl-st-inv S'*›
using *struct-invs* **unfolding** *S twl-struct-invs-def*
by *fast*
then have
‹∀ *x*∈#*ran-m N*. *struct-wf-twl-cls* (*twl-clause-of* (*fst x*))›
using *struct-invs S'* **unfolding** *S twl-st-inv-alt-def*
by *simp*
then have ‹*Multiset.Ball* (*dom-m N*) (λ*C*. *length* (*N* ∝ *C*) ≥ *2*)›
by (*subst* (*asm*) *Ball-ran-m-dom-struct-wf*) *auto*

then have ‹*length* $(N \propto C) \geq 2$›
  using ‹$C \in\#$ *dom-m N*› **unfolding** *S* **by** (*auto simp*: *twl-list-invs-def*)
}
moreover {
  have
    lev-confl: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting* ($state_W$*-of* S')› **and**
    M-lev: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv* ($state_W$*-of* S')›
    using *struct* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def* **by** *fast*+
  then have ‹$M \models as$ *CNot* (*remove1-mset L* (*mset* $(N \propto C)$))›
    using S' *False*
    by (*force simp*: *S twl-st-l-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting-def*
      $cdcl_W$*-restart-mset-state convert-lit.simps*
      elim!: *convert-lits-l-consE*)
  then have *1*: ‹$-L' \notin\#$ *mset* $(N \propto C)$›
    apply − **apply** (*rule*, *drule multi-member-split*)
    using S' *M-lev False* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def*
    by (*auto simp*: *S twl-st-l-def* $cdcl_W$*-restart-mset-state split*: *if-splits*
      dest: *in-lits-of-l-defined-litD*)
  then have *2*:‹*remove1-mset* $(-\ L')$ (*the D*) $\cup\#$ *mset* (*tl* $(N \propto C)$) =
    remove1-mset $(-\ L')$ (*the D* $\cup\#$ *mset* (*tl* $(N \propto C)$))›
    using *L* **by**(*cases* ‹$N \propto C$›; *cases* ‹$-L' \in\#$ *mset* $(N \propto C)$›)
      (*auto simp*: *remove1-mset-union-distrib*)
  have ‹*Propagated L C* # $M \models as$ *CNot* (*the D*)›
    using S' *False lev-confl* ‹$D \neq$ *None*›
    by (*force simp*: *S twl-st-l-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting-def*
      $cdcl_W$*-restart-mset-state convert-lit.simps*)
  then have *3*: ‹$L' \notin\#$ (*the D*)›
    apply − **apply** (*rule*, *drule multi-member-split*)
    using S' *M-lev False* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def*
    by (*auto simp*: *S twl-st-l-def* $cdcl_W$*-restart-mset-state split*: *if-splits*
      dest: *in-lits-of-l-defined-litD*)

  note *1* **and** *2* **and** *3*
}
ultimately show *?thesis*
  using *invs-S* S'
  by (*cases* ‹$N \propto C$›; *cases* ‹*tl* $(N \propto C)$›)
    (*auto simp*: *skip-and-resolve-loop-inv-def twl-list-invs-def resolve-cls-l*$'$*-def*
      resolve-cls-l-nil-iff update-confl-tl-l-def update-confl-tl-def twl-st-l-def H H'
      S S' C' *dest*!: *False dest*: *convert-lits-l-tlD*)
qed

lemma *get-level-same-lits-cong*:
  assumes
    ‹*map* (*atm-of o lit-of*) *M* = *map* (*atm-of o lit-of*) M'› **and**
    ‹*map is-decided M* = *map is-decided* M'›
  shows ‹*get-level M L* = *get-level* M' *L*›
proof −
  have [*dest*]: ‹*map is-decided M* = *map is-decided zsa* $\Longrightarrow$
      length (*filter is-decided M*) = *length* (*filter is-decided zsa*)›
    for *M* :: ‹('*d*, '*e*, '*f*) *annotated-lit list*› **and** *zsa* :: ‹('*g*, '*h*, '*i*) *annotated-lit list*›
    by (*induction M arbitrary*: *zsa*) (*auto simp*: *get-level-def*)

  show *?thesis*
    using *assms*
    by (*induction M arbitrary*: M') (*auto simp*: *get-level-def*)

qed

lemma *clauses-in-unit-clss-have-level0*:
assumes
struct-invs: ‹*twl-struct-invs T*› **and**
C: ‹*C* ∈# *unit-clss T*› **and**
LC-T: ‹*Propagated L C* ∈ *set* (*get-trail T*)› **and**
count-dec: ‹*0* < *count-decided* (*get-trail T*)›
shows
‹*get-level* (*get-trail T*) *L* = *0*› (**is** *?lev-L*) **and**
‹∀ *K*∈# *C*. *get-level* (*get-trail T*) *K* = *0*› (**is** *?lev-K*)
proof −
have
all-struct: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* ($state_W$*-of T*)› **and**
ent: ‹*entailed-clss-inv T*›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
by *fast+*
obtain *K* **where**
‹*K* ∈# *C*› **and** *lev-K*: ‹*get-level* (*get-trail T*) *K* = *0*› **and** *K-M*: ‹*K* ∈ *lits-of-l* (*get-trail T*)›
using *ent C count-dec* **by** (*cases T*; *cases* ‹*get-conflict T*›) *auto*
thm *entailed-clss-inv.simps*
obtain *M1 M2* **where**
M: ‹*get-trail T* = *M2* @ *Propagated L C* # *M1*›
using *LC-T* **by** (*blast elim*: *in-set-list-format*)
have ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting* ($state_W$*-of T*)› **and**
lev-inv: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv* ($state_W$*-of T*) ›
using *all-struct* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
by *fast+*
then have *M1*: ‹*M1* ⊨*as CNot* (*remove1-mset L C*)› **and** ‹*L* ∈# *C*›
using *M* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting-def*
by (*auto simp*: *twl-st*)
moreover have *n-d*: ‹*no-dup* (*get-trail T*)›
using *lev-inv* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def* **by** (*simp add*: *twl-st*)
ultimately have ‹*L* = *K*›
using ‹*K* ∈# *C*› *M K-M*
by (*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset*
dest: *in-lits-of-l-defined-litD no-dup-uminus-append-in-atm-notin*
no-dup-appendD no-dup-consistentD)
then show *?lev-L*
using *lev-K* **by** *simp*
have *count-dec-M1*: ‹*count-decided M1* = *0*›
using *M n-d* ‹*?lev-L*› **by** *auto*
have ‹*get-level* (*get-trail T*) *K* = *0*› **if** ‹*K* ∈# *C*› **for** *K*
proof −
have ‹−*K* ∈ *lits-of-l* (*Propagated* (−*L*) *C* # *M1*)›
using *M1 M that* **by** (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model remove1-mset-add-mset-If*
dest!: *multi-member-split dest*: *in-diffD split*: *if-splits*)
then have ‹*get-level* (*get-trail T*) *K* = *get-level* (*Propagated* (−*L*) *C* # *M1*) *K*›
apply −
apply (*subst* (*2*) *get-level-skip*[*symmetric*, *of M2*])
using *n-d M* **by** (*auto dest*: *no-dup-uminus-append-in-atm-notin*
intro: *get-level-same-lits-cong*)
then show *?thesis*
using *count-decided-ge-get-level*[*of* ‹*Propagated* (−*L*) *C* # *M1*› *K*] *count-dec-M1*
by (*auto simp*: *get-level-cons-if split*: *if-splits*)
qed

then show *?lev-K*
by *fast*
qed

lemma *clauses-clss-have-level1-notin-unit*:
assumes
struct-invs: ‹*twl-struct-invs T*› **and**
LC-T: ‹*Propagated L C* ∈ *set* (*get-trail T*)› **and**
count-dec: ‹*0* < *count-decided* (*get-trail T*)› **and**
‹*get-level* (*get-trail T*) *L* > *0*›
shows
‹*C* ∉# *unit-clss T*›
using *clauses-in-unit-clss-have-level0*[*of T C*, *OF struct-invs - LC-T count-dec*] *assms*
by *linarith*

lemma *skip-and-resolve-loop-l-spec*:
‹(*skip-and-resolve-loop-l*, *skip-and-resolve-loop*) ∈
{(*S*::′*v twl-st-l*, *S*′). (*S*, *S*′) ∈ *twl-st-l None* ∧ *twl-struct-invs S*′ ∧
twl-stgy-invs S′ ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#} ∧ *literals-to-update-l S* = {#} ∧
get-conflict S′ ≠ *None* ∧
0 < *count-decided* (*get-trail-l S*)} $\rightarrow_f$
⟨{(*T*, *T*′). (*T*, *T*′) ∈ *twl-st-l None* ∧ *twl-list-invs T* ∧
(*twl-struct-invs T*′ ∧ *twl-stgy-invs T*′ ∧
no-step $cdcl_W$*-restart-mset.skip* ($state_W$*-of T*′) ∧
no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of T*′) ∧
literals-to-update T′ = {#} ∧
clauses-to-update-l T = {#} ∧ *get-conflict T*′ ≠ *None*)}⟩ *nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ -›)
proof −
have *is-proped*[*iff*]: ‹*is-proped* (*hd* (*get-trail S*′)) ⟷ *is-proped* (*hd* (*get-trail-l S*))›
if ‹*get-trail-l S* ≠ []› **and**
‹(*S*, *S*′) ∈ *twl-st-l None*›
for *S* :: ‹′*v twl-st-l*› **and** *S*′
by (*cases S*, *cases* ‹*get-trail-l S*›; *cases* ‹*hd* (*get-trail-l S*)›)
(*use that* **in** ‹*auto split*: *if-splits simp*: *twl-st-l-def*›)
have
mark-ge-0: ‹*0* < *mark-of* (*hd* (*get-trail-l T*))› (**is** *?ge*) **and**
nempty: ‹*get-trail-l T* ≠ []› ‹*get-trail* (*snd brkT*′) ≠ []› (**is** *?nempty*)
if
SS′: ‹(*S*, *S*′) ∈ *?R*› **and**
‹*get-conflict-l S* ≠ *None*› **and**
brk-TT′: ‹(*brkT*, *brkT*′)
∈ {((*brk*, *S*), *brk*′, *S*′). *brk* = *brk*′ ∧ (*S*, *S*′) ∈ *twl-st-l None* ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#}}› (**is** ‹- ∈ *?brk*›) **and**
loop-inv: ‹*skip-and-resolve-loop-inv S*′ *brkT*′› **and**
brkT: ‹*brkT* = (*brk*, *T*)› **and**
dec: ‹¬ *is-decided* (*hd* (*get-trail-l T*))›
for *S S*′ *brkT brkT*′ *brk T*
proof −
obtain *brk*′ *T*′ **where** *brkT*′: ‹*brkT*′ = (*brk*′, *T*′)› **by** (*cases brkT*′)
have ‹$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of T*′)› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of T*′)› **and**
tr: ‹*get-trail T*′ ≠ []› ‹*get-trail-l T* ≠ []› **and**
count-dec: ‹*count-decided* (*get-trail-l T*) ≠ *0*› ‹*count-decided* (*get-trail T*′) ≠ *0*› **and**
TT′: ‹(*T*,*T*′) ∈ *twl-st-l None*› **and**

struct-invs: ‹*twl-struct-invs* T'›
using *loop-inv brk-TT'* **unfolding** *twl-struct-invs-def skip-and-resolve-loop-inv-def brkT brkT'*
by *auto*
moreover have ‹*Suc 0* ≤ *backtrack-lvl* (*state*$_W$*-of* T')›
using *count-dec TT'* **by** (*auto simp*: *trail.simps*)
moreover have *proped*: ‹*is-proped* (*cdcl*$_W$*-restart-mset.hd-trail* (*state*$_W$*-of* T'))›
using *dec tr TT'* **by** (*cases* ‹*get-trail-l T*›)
(*auto simp*: *trail.simps is-decided-no-proped-iff twl-st*)
moreover have ‹*mark-of* (*hd* (*get-trail* T')) ∉# *unit-clss* T'›
using *clauses-clss-have-level1-notin-unit*(*1*)[*of* T' ‹*lit-of* (*hd* (*get-trail* T'))›
‹*mark-of* (*hd* (*get-trail* T'))›] *dec struct-invs count-dec tr proped TT'*
by (*cases* ‹*get-trail* T'›; *cases* ‹*hd* (*get-trail* T')›)
(*auto simp*: *twl-st*)
moreover have ‹*convert-lit* (*get-clauses-l T*) (*unit-clss* T') (*hd* (*get-trail-l T*))
(*hd* (*get-trail* T'))›
using *tr dec TT'*
by (*cases* ‹*get-trail* T'›; *cases* ‹*get-trail-l T*›)
(*auto simp*: *twl-st-l-def*)
ultimately have ‹*mark-of* (*hd* (*get-trail-l T*)) = *0* ⟹ *False*›
using *tr dec TT'* **by** (*cases* ‹*get-trail-l T*›; *cases* ‹*hd* (*get-trail-l T*)›)
(*auto simp*: *trail.simps twl-st convert-lit.simps*)
then show *?ge* **by** *blast*
show ‹*get-trail-l T* ≠ []› ‹*get-trail* (*snd brkT'*) ≠ []›
using *tr TT' brkT'* **by** *auto*
qed
have *H*: ‹*RETURN* (*lit-and-ann-of-propagated* (*hd* (*get-trail-l T*)))
≤ ⇓ {((*L*, *C*), (*L'*, *C'*)). *L* = *L'* ∧ *C*> *0* ∧ *C'* = *mset* (*get-clauses-l T* ∝ *C*)}
(*SPEC* (λ(*L*, *C*). *Propagated L C* = *hd* (*get-trail* T')))›
if
SS': ‹(*S*, *S'*) ∈ *?R*› **and**
confl: ‹*get-conflict-l S* ≠ *None*› **and**
brk-TT': ‹(*brkT*, *brkT'*) ∈ *?brk*› **and**
loop-inv: ‹*skip-and-resolve-loop-inv S' brkT'*› **and**
brkT: ‹*brkT* = (*brk*, *T*)› **and**
dec: ‹¬ *is-decided* (*hd* (*get-trail-l T*))› **and**
brkT': ‹*brkT'* = (*brk'*, T')›
for *S* :: ‹*'v twl-st-l*› **and** *S'* :: ‹*'v twl-st*› **and** *T T' brk brk' brkT' brkT*
using *confl brk-TT' loop-inv brkT dec mark-ge-0*[*OF SS' confl brk-TT' loop-inv brkT dec*]
nempty[*OF SS' confl brk-TT' loop-inv brkT dec*] **unfolding** *brkT'*
apply (*cases T*; *cases T'*; *cases* ‹*get-trail-l T*›; *cases* ‹*hd* (*get-trail-l T*)› ;
cases ‹*get-trail* T'›; *cases* ‹*hd* (*get-trail* T')›)
apply ((*solves* ‹*force split*: *if-splits*›)+)[*15*]
unfolding *RETURN-def*
by (*rule RES-refine*; *solves* ‹*auto split*: *if-splits simp*: *twl-st-l-def convert-lit.simps*›)+
have *skip-and-resolve-loop-inv-trail-nempty*: ‹*skip-and-resolve-loop-inv S'* (*False*, *S*) ⟹
get-trail S ≠ []› **for** *S* :: ‹*'v twl-st*› **and** *S'*
unfolding *skip-and-resolve-loop-inv-def*
by *auto*

have *twl-list-invs-tl-state-l*: ‹*twl-list-invs S* ⟹ *twl-list-invs* (*tl-state-l S*)›
for *S* :: ‹*'v twl-st-l*›
by (*cases S*, *cases* ‹*get-trail-l S*›) (*auto simp*: *tl-state-l-def twl-list-invs-def*)
have *clauses-to-update-l-tl-state*: ‹*clauses-to-update-l* (*tl-state-l S*) = *clauses-to-update-l S*›
for *S* :: ‹*'v twl-st-l*›
by (*cases S*, *cases* ‹*get-trail-l S*›) (*auto simp*: *tl-state-l-def*)

have H:
‹(*skip-and-resolve-loop-l*, *skip-and-resolve-loop*) ∈ $?R \rightarrow_f$
⟨{(T::'v *twl-st-l*, T'). (T, T') ∈ *twl-st-l None* ∧ *twl-list-invs* T ∧
clauses-to-update-l T = {#}}⟩ *nres-rel*›
supply [[*goals-limit=1*]]
unfolding *skip-and-resolve-loop-l-def skip-and-resolve-loop-def fref-param1*[*symmetric*]
apply (*refine-vcg H*)
subgoal by *auto* — conflict is not none
apply (*rule get-conflict-l-get-conflict-state-spec*)
subgoal by *auto* — loop invariant init: *skip-and-resolve-loop-inv*
subgoal by *auto* — loop invariant init: *twl-list-invs*
subgoal by *auto* — loop invariant init: *clauses-to-update* S = {#}
subgoal for S S' *brkT brkT'*
unfolding *skip-and-resolve-loop-inv-l-def*
apply(*rule exI*[*of* - ‹*snd brkT'*›])
apply(*rule exI*[*of* - S'])
apply (*intro conjI impI*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule mark-ge-0*)
done
— align loop conditions
subgoal by (*auto dest*!: *skip-and-resolve-loop-inv-trail-nempty*)
apply *assumption+*
subgoal by *auto*
apply *assumption+*
subgoal by *auto*
subgoal by (*drule skip-and-resolve-l-refines*) *blast+*
subgoal by (*auto simp*: *twl-list-invs-tl-state-l*)
subgoal by (*rule skip-and-resolve-skip-refine*)
(*auto simp*: *skip-and-resolve-loop-inv-def*)
— annotations are valid
subgoal by *auto*
subgoal by *auto*
done
have H: ‹(*skip-and-resolve-loop-l*, *skip-and-resolve-loop*)
∈ $?R \rightarrow_f$
⟨{(T::'v *twl-st-l*, T').
(T, T') ∈ {(T, T'). (T, T') ∈ *twl-st-l None* ∧ (*twl-list-invs* T ∧
clauses-to-update-l T = {#})} ∧
T' ∈ {T'. *twl-struct-invs* T' ∧ *twl-stgy-invs* T' ∧
(*no-step* $cdcl_W$*-restart-mset.skip* ($state_W$*-of* T')) ∧
(*no-step* $cdcl_W$*-restart-mset.resolve* ($state_W$*-of* T')) ∧
literals-to-update T' = {#} ∧
get-conflict T' ≠ *None*}}⟩*nres-rel*›
apply (*rule refine-add-inv-generalised*)
subgoal by (*rule H*)
subgoal for S S'
apply (*rule order-trans*)
apply (*rule skip-and-resolve-loop-spec*[*of* S'])
by *auto*
done
show *?thesis*

using H **apply** $-$
apply (*match-spec*; (*match-fun-rel*; *match-fun-rel?*)+)
by *blast+*
qed

end

definition *find-decomp* :: ‹$'v$ *literal* $\Rightarrow$ $'v$ *twl-st-l* $\Rightarrow$ $'v$ *twl-st-l nres*› **where**
‹*find-decomp* $=$ $(\lambda L\ (M,\ N,\ D,\ NE,\ UE,\ WS,\ Q).$
$SPEC(\lambda S.\ \exists K\ M2\ M1.\ S = (M1,\ N,\ D,\ NE,\ UE,\ WS,\ Q) \wedge$
$(Decided\ K\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M) \wedge$
$get\text{-}level\ M\ K = get\text{-}maximum\text{-}level\ M\ (the\ D - \{\#-L\#\}) + 1))$›

lemma *find-decomp-alt-def*:
‹*find-decomp* L S $=$
$SPEC(\lambda T.\ \exists K\ M2\ M1.\ equality\text{-}except\text{-}trail\ S\ T \wedge get\text{-}trail\text{-}l\ T = M1 \wedge$
$(Decided\ K\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (get\text{-}trail\text{-}l\ S)) \wedge$
$get\text{-}level\ (get\text{-}trail\text{-}l\ S)\ K =$
$get\text{-}maximum\text{-}level\ (get\text{-}trail\text{-}l\ S)\ (the\ (get\text{-}conflict\text{-}l\ S) - \{\#-L\#\}) + 1)$›
unfolding *find-decomp-def*
by (*cases* S) *force*

definition *find-lit-of-max-level* :: ‹$'v$ *twl-st-l* $\Rightarrow$ $'v$ *literal* $\Rightarrow$ $'v$ *literal nres*› **where**
‹*find-lit-of-max-level* $=$ $(\lambda(M,\ N,\ D,\ NE,\ UE,\ WS,\ Q)\ L.$
$SPEC(\lambda L'.\ L' \in\#\ the\ D - \{\#-L\#\} \wedge get\text{-}level\ M\ L' = get\text{-}maximum\text{-}level\ M\ (the\ D - \{\#-L\#\})))$›

definition *ex-decomp-of-max-lvl* :: ‹$('v,\ nat)$ *ann-lits* $\Rightarrow$ $'v$ *cconflict* $\Rightarrow$ $'v$ *literal* $\Rightarrow$ *bool*› **where**
‹*ex-decomp-of-max-lvl* $M\ D\ L \longleftrightarrow$
$(\exists K\ M1\ M2.\ (Decided\ K\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M) \wedge$
$get\text{-}level\ M\ K = get\text{-}maximum\text{-}level\ M\ (remove1\text{-}mset\ (-L)\ (the\ D)) + 1)$›

fun *add-mset-list* :: ‹$'a$ *list* $\Rightarrow$ $'a$ *multiset multiset* $\Rightarrow$ $'a$ *multiset multiset*› **where**
‹*add-mset-list* L UE $=$ *add-mset* (*mset* L) UE›

definition (**in** $-$)*list-of-mset* :: ‹$'v$ *clause* $\Rightarrow$ $'v$ *clause-l nres*› **where**
‹*list-of-mset* $D = SPEC(\lambda D'.\ D = mset\ D')$›

fun *extract-shorter-conflict-l* :: ‹$'v$ *twl-st-l* $\Rightarrow$ $'v$ *twl-st-l nres*›
where
‹*extract-shorter-conflict-l* $(M,\ N,\ D,\ NE,\ UE,\ WS,\ Q) = SPEC(\lambda S.$
$\exists D'.\ D' \subseteq\#\ the\ D \wedge S = (M,\ N,\ Some\ D',\ NE,\ UE,\ WS,\ Q) \wedge$
$clause\ `\#\ twl\text{-}clause\text{-}of\ `\#\ ran\text{-}mf\ N + NE + UE \models pm\ D' \wedge -(lit\text{-}of\ (hd\ M)) \in\#\ D')$›

declare *extract-shorter-conflict-l.simps*[*simp del*]
lemmas *extract-shorter-conflict-l-def* $=$ *extract-shorter-conflict-l.simps*

lemma *extract-shorter-conflict-l-alt-def*:
‹*extract-shorter-conflict-l* $S = SPEC(\lambda T.$
$\exists D'.\ D' \subseteq\#\ the\ (get\text{-}conflict\text{-}l\ S) \wedge equality\text{-}except\text{-}conflict\text{-}l\ S\ T \wedge$
$get\text{-}conflict\text{-}l\ T = Some\ D' \wedge$
$clause\ `\#\ twl\text{-}clause\text{-}of\ `\#\ ran\text{-}mf\ (get\text{-}clauses\text{-}l\ S) + get\text{-}unit\text{-}clauses\text{-}l\ S \models pm\ D' \wedge$
$-lit\text{-}of\ (hd\ (get\text{-}trail\text{-}l\ S)) \in\#\ D')$›
by (*cases* S) (*auto simp*: *extract-shorter-conflict-l-def ac-simps*)

definition *backtrack-l-inv* **where**

⟨*backtrack-l-inv S* ⟷
    $(\exists S'.\ (S, S') \in$ *twl-st-l None* $\wedge$
    get-trail-l S $\neq [] \wedge$
    no-step $cdcl_W$*-restart-mset.skip* ($state_W$*-of* S')$\wedge$
    no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of* S') $\wedge$
    get-conflict-l S $\neq$ *None* $\wedge$
    twl-struct-invs S' $\wedge$
    twl-stgy-invs S' $\wedge$
    twl-list-invs S $\wedge$
    get-conflict-l S $\neq$ *Some* {#})
⟩

definition *get-fresh-index* :: ⟨'*v clauses-l* ⇒ *nat nres*⟩ **where**
⟨*get-fresh-index N* = *SPEC*($\lambda i.\ i > 0 \wedge i \notin\#$ *dom-m N*)⟩

definition *propagate-bt-l* :: ⟨'*v literal* ⇒ '*v literal* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*⟩ **where**
⟨*propagate-bt-l* = ($\lambda L\ L'$ (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). *do* {
  D'' ← *list-of-mset* (*the D*);
  i ← *get-fresh-index N*;
  RETURN (*Propagated* ($-L$) *i* # *M*,
    fmupd i ([$-L$, L'] @ (*remove1* ($-L$) (*remove1* L' D'')), *False*) *N*,
      None, *NE*, *UE*, *WS*, {#*L*#})
  })⟩

definition *propagate-unit-bt-l* :: ⟨'*v literal* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l*⟩ **where**
⟨*propagate-unit-bt-l* = (λL (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
  (*Propagated* ($-L$) *0* # *M*, *N*, *None*, *NE*, *add-mset* (*the D*) *UE*, *WS*, {#*L*#}))⟩

definition *backtrack-l* :: ⟨'*v twl-st-l* ⇒ '*v twl-st-l nres*⟩ **where**
⟨*backtrack-l S* =
  do {
    ASSERT(*backtrack-l-inv S*);
    let L = *lit-of* (*hd* (*get-trail-l S*));
    S ← *extract-shorter-conflict-l S*;
    S ← *find-decomp L S*;

    if size (*the* (*get-conflict-l S*)) > *1*
    then do {
      L' ← *find-lit-of-max-level S L*;
      propagate-bt-l L L' *S*
    }
    else do {
      RETURN (*propagate-unit-bt-l L S*)
   }
})⟩

lemma *backtrack-l-spec*:
⟨(*backtrack-l*, *backtrack*) ∈
  {(*S*::'*v twl-st-l*, S'). (*S*, S') ∈ *twl-st-l None* ∧ *get-conflict-l S* ≠ *None* ∧
    get-conflict-l S ≠ *Some* {#} ∧
    clauses-to-update-l S = {#} ∧ *literals-to-update-l S* = {#} ∧ *twl-list-invs S* ∧
    no-step $cdcl_W$*-restart-mset.skip* ($state_W$*-of* S') ∧
    no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of* S') ∧
    twl-struct-invs S' ∧ *twl-stgy-invs* S'} $\rightarrow_f$
  ⟨{(*T*::'*v twl-st-l*, T'). (*T*, T') ∈ *twl-st-l None* ∧ *get-conflict-l T* = *None* ∧ *twl-list-invs T* ∧
    twl-struct-invs T' ∧ *twl-stgy-invs* T' ∧ *clauses-to-update-l T* = {#} ∧

literals-to-update-l $T \neq \{\#\}\}\rangle$ *nres-rel*›
(**is** ‹ - $\in$ *?R* $\rightarrow_f$ *?I*›)
proof –
have *H*: ‹*find-decomp L S*
$\leq \Downarrow \{(T,\ T').\ (T,\ T') \in$ *twl-st-l None* $\wedge$ *equality-except-trail S T* $\wedge$
$(\exists M.$ *get-trail-l S* $= M$ @ *get-trail-l T*$)\}$
(*reduce-trail-bt L′ S′*)›
(**is** ‹- $\leq$ $\Downarrow$ *?find-decomp* -›)
if
SS′: ‹$(S,\ S') \in$ *twl-st-l None*› **and** ‹$L =$ *lit-of* (*hd* (*get-trail-l S*))› **and**
‹$L' =$ *lit-of* (*hd* (*get-trail S′*))› ‹*get-trail-l S* $\neq$ []›
for *S* :: ‹*′v twl-st-l*› **and** *S′* **and** *L′ L*
unfolding *find-decomp-alt-def reduce-trail-bt-def*
state-decomp-to-state
apply (*subst RES-RETURN-RES*)
apply (*rule RES-refine*)
unfolding *in-pair-collect-simp bex-simps*
using *that* **apply** (*auto 5 5 intro*!: *RES-refine convert-lits-l-decomp-ex*)
apply (*rule-tac x*=‹*drop* (*length* (*get-trail S′*) $-$ *length a*) (*get-trail S′*)› **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac x*=*K* **in** *exI*)
apply (*auto simp*: *twl-st-l-def*
intro: *convert-lits-l-decomp-ex*)
done

have *list-of-mset*: ‹*list-of-mset D′* $\leq$ *SPEC* ($\lambda c.\ (c,\ D'') \in \{(c,\ D).\ D =$ *mset c*$\}$)›
if ‹$D' = D''$› **for** *D′* :: ‹*′v clause*› **and** *D″*
using *that* **by** (*cases D″*) (*auto simp*: *list-of-mset-def*)
have *ext*: ‹*extract-shorter-conflict-l T*
$\leq \Downarrow \{(S,\ S').\ (S,\ S') \in$ *twl-st-l None* $\wedge$
$-$*lit-of* (*hd* (*get-trail-l S*)) $\in\#$ *the* (*get-conflict-l S*) $\wedge$
the (*get-conflict-l S*) $\subseteq\#$ *the* D_0 $\wedge$ *equality-except-conflict-l T S* $\wedge$ *get-conflict-l S* $\neq$ *None*$\}$
(*extract-shorter-conflict T′*)›
(**is** ‹- $\leq$ $\Downarrow$ *?extract* -›)
if ‹$(T,\ T') \in$ *twl-st-l None*› **and**
‹$D_0 =$ *get-conflict-l T*› **and**
‹*get-trail-l T* $\neq$ []›
for *T* :: ‹*′v twl-st-l*› **and** *T′* **and** D_0
unfolding *extract-shorter-conflict-l-alt-def extract-shorter-conflict-alt-def*
apply (*rule RES-refine*)
unfolding *in-pair-collect-simp bex-simps*
apply *clarify*
apply (*rule-tac x*=‹*set-conflict′* (*Some D′*) *T′*› **in** *bexI*)
using *that*
apply (*auto simp del*: *split-paired-Ex equality-except-conflict-l.simps*
simp: *set-conflict′-def*[*unfolded state-decomp-to-state*]
intro!: *RES-refine equality-except-conflict-alt-def*[*THEN iffD2*]
del: *split-paired-all*)
apply (*auto simp*: *twl-st-l-def equality-except-conflict-l-alt-def*)
done

have *uhd-in-D*: ‹$L \in\#$ *the D*›
if
inv-s: ‹*twl-stgy-invs S′*› **and**
inv: ‹*twl-struct-invs S′*› **and**
ns: ‹*no-step* $cdcl_W$*-restart-mset.skip* ($state_W$*-of S′*)› **and**

confl:
⟨*conflicting* (*state*$_W$*-of* S') ≠ *None*⟩
⟨*conflicting* (*state*$_W$*-of* S') ≠ *Some* {#}⟩ **and**
M-nempty: ⟨*get-trail-l* S ≠ []⟩ **and**
D: ⟨D = *get-conflict-l* S⟩
⟨L = − *lit-of* (*hd* (*get-trail-l* S))⟩ **and**
SS': ⟨(S, S') ∈ *twl-st-l None*⟩
for L M D **and** S :: ⟨*'v twl-st-l*⟩ **and** S' :: ⟨*'v twl-st*⟩
unfolding D
using *cdcl*$_W$*-restart-mset.no-step-skip-hd-in-conflicting*[*of* ⟨*state*$_W$*-of* S'⟩,
OF - - *ns confl*] *that*
by (*auto simp*: *cdcl*$_W$*-restart-mset-state twl-stgy-invs-def*
twl-struct-invs-def twl-st)

have *find-lit*:
⟨*find-lit-of-max-level* U (*lit-of* (*hd* (*get-trail-l* S)))
≤ *SPEC* (λL''. L'' ∈# *remove1-mset* (− *lit-of* (*hd* (*get-trail* S'))) (*the* (*get-conflict* U')) ∧
lit-of (*hd* (*get-trail* S')) ≠ − L'' ∧
get-level (*get-trail* U') L'' = *get-maximum-level* (*get-trail* U')
(*remove1-mset* (− *lit-of* (*hd* (*get-trail* S'))) (*the* (*get-conflict* U'))))⟩
(**is** ⟨- ≤ *RES ?find-lit-of-max-level*⟩)
if
UU': ⟨(S, S') ∈ *?R*⟩ **and**
bt-inv: ⟨*backtrack-l-inv* S⟩ **and**
RR': ⟨(T, T') ∈ *?extract* S (*get-conflict-l* S)⟩ **and**
T: ⟨(U, U') ∈ *?find-decomp* T⟩
for S S' T T' U U'
proof −
have *SS'*: ⟨(S, S') ∈ *twl-st-l None*⟩ ⟨*get-trail-l* S ≠ []⟩ **and**
struct-invs: ⟨*twl-struct-invs* S'⟩ ⟨*get-conflict-l* S ≠ *None*⟩
using *UU' bt-inv* **by** (*auto simp*: *backtrack-l-inv-def*)
have ⟨*cdcl*$_W$*-restart-mset.distinct-cdcl*$_W$*-state* (*state*$_W$*-of* S')⟩
using *struct-invs* **unfolding** *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
by *fast*
then have *dist*: ⟨*distinct-mset* (*the* (*get-conflict-l* S))⟩
using *struct-invs SS'* **unfolding** *cdcl*$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def*
by (*cases* S) (*auto simp*: *cdcl*$_W$*-restart-mset-state twl-st*)
then have *dist*: ⟨*distinct-mset* (*the* (*get-conflict-l* U))⟩
using *UU' RR' T* **by** (*cases* S, *cases* T, *cases* U, *auto intro*: *distinct-mset-mono*)
show *?thesis*
using *T distinct-mem-diff-mset*[*OF dist*, *of* - ⟨{#-#}⟩] *SS'*
unfolding *find-lit-of-max-level-def*
state-decomp-to-state-l
by (*force simp*: *uminus-lit-swap*)
qed

have *propagate-bt*:
⟨*propagate-bt-l* (*lit-of* (*hd* (*get-trail-l* S))) L U
≤ *SPEC* (λc. (c, *propagate-bt* (*lit-of* (*hd* (*get-trail* S'))) L' U') ∈
{(T, T'). (T, T') ∈ *twl-st-l None* ∧ *clauses-to-update-l* T = {#} ∧ *twl-list-invs* T})⟩
if
SS': ⟨(S, S') ∈ *?R*⟩ **and**
bt-inv: ⟨*backtrack-l-inv* S⟩ **and**
TT': ⟨(T, T') ∈ *?extract* S (*get-conflict-l* S)⟩ **and**
UU': ⟨(U, U') ∈ *?find-decomp* T⟩ **and**
L': ⟨L' ∈ *?find-lit-of-max-level* S' U'⟩ **and**

LL': ‹(*L*, *L'*) ∈ *Id*› **and**
size: ‹*size* (*the* (*get-conflict-l U*)) > *1*›
for *S S' T T' U U' L L'*
proof −
obtain *MS NS DS NES UES* **where**
S: ‹*S* = (*MS*, *NS*, *Some DS*, *NES*, *UES*, {#}, {#})› **and**
S-S': ‹(*S*, *S'*) ∈ *twl-st-l None*› **and**
add-invs: ‹*twl-list-invs S*› **and**
struct-inv: ‹*twl-struct-invs S'*› **and**
stgy-inv: ‹*twl-stgy-invs S'*› **and**
nss: ‹*no-step* $cdcl_W$*-restart-mset.skip* ($state_W$*-of S'*)› **and**
nsr: ‹*no-step* $cdcl_W$*-restart-mset.resolve* ($state_W$*-of S'*)› **and**
confl: ‹*get-conflict-l S* ≠ *None*› ‹*get-conflict-l S* ≠ *Some* {#}›
using *SS'* **by** (*cases S*; *cases* ‹*get-conflict-l S*›) *auto*
then obtain *DT* **where**
T: ‹*T* = (*MS*, *NS*, *Some DT*, *NES*, *UES*, {#}, {#})› **and**
T-T': ‹(*T*, *T'*) ∈ *twl-st-l None*›
using *TT'* **by** (*cases T*; *cases* ‹*get-conflict-l T*›) *auto*
then obtain *MU MU'* **where**
U: ‹*U* = (*MU*, *NS*, *Some DT*, *NES*, *UES*, {#}, {#})› **and**
MU: ‹*MS* = *MU'* @ *MU*› **and**
U-U': ‹(*U*, *U'*) ∈ *twl-st-l None*›
using *UU'* **by** (*cases U*) *auto*
have [*simp*]: ‹*L* = *L'*›
using *LL'* **by** *simp*

have [*simp*]: ‹*MS* ≠ []› **and** *add-invs*: ‹*twl-list-invs S*›
using *SS' bt-inv* **unfolding** *twl-list-invs-def backtrack-l-inv-def S* **by** *auto*
have ‹*Suc 0* < *size DT*›
using *size* **by** (*auto simp*: *U*)
then have ‹*DS* ≠ {#}›
using *TT'* **by** (*auto simp*: *S T*)
moreover have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant* ($state_W$*-of S'*)›
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S'*)›
using *struct-inv stgy-inv* **unfolding** *twl-struct-invs-def twl-stgy-invs-def*
by *fast+*
ultimately have ‹− *lit-of* (*hd MS*) ∈# *DS*›
using *bt-inv* $cdcl_W$*-restart-mset.no-step-skip-hd-in-conflicting*[*of* ‹$state_W$*-of S'*›]
size struct-inv stgy-inv nss nsr confl SS'
unfolding *backtrack-l-inv-def*
by (*auto simp*: $cdcl_W$*-restart-mset-state S twl-st*)
then have ‹− *lit-of* (*hd MS*) ∈# *DT*›
using *TT'* **by** (*auto simp*: *T*)
moreover have ‹*L'* ∈# *remove1-mset* (− *lit-of* (*hd MS*)) *DT*›
using *L' S-S' U-U'* **by** (*auto simp*: *S U*)
ultimately have *DT*:
‹*DT* = *add-mset* (− *lit-of* (*hd MS*)) (*add-mset L'* (*DT* − {#− *lit-of* (*hd MS*), *L'*#}))›
by (*metis* (*no-types*, *lifting*) *add-mset-diff-bothsides diff-single-eq-union*)
have [*simp*]: ‹*Propagated L i* ∉ *set MU*›
if
i-dom: ‹*i* ∉# *dom-m NS*› **and**
‹*i* > *0*›
for *L i*
using *add-invs that* **unfolding** *S MU twl-list-invs-def*
by *auto*
have *Propa*:

⟨((*Propagated* (− *lit-of* (*hd MS*)) *i* # *MU*,
  fmupd i (− *lit-of* (*hd MS*) # *L* # *remove1* (− *lit-of* (*hd MS*)) (*remove1 L xa*), *False*) *NS*,
    None, *NES*, *UES*, {#}, *unmark* (*hd MS*)),
   case U′ of
   (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⇒
    (*Propagated* (− *lit-of* (*hd* (*get-trail S′*))) (*the D*) # *M*, *N*,
     add-mset
      (*TWL-Clause* {#− *lit-of* (*hd* (*get-trail S′*)), *L′*#}
       (*the D* − {#− *lit-of* (*hd* (*get-trail S′*)), *L′*#}))
      U,
     None, *NE*, *UE*, *WS*, *unmark* (*hd* (*get-trail S′*))))
  ∈ *twl-st-l None*⟩
if
 [*symmetric*, *simp*]: ⟨*DT* = *mset xa*⟩ **and**
 i-dom: ⟨*i* ∉# *dom-m NS*⟩ **and**
 ⟨*i* > *0*⟩
for *i xa*
using *U-U′ S-S′ T-T′ i-dom* ⟨*i* > *0*⟩ *DT* **apply** (*cases U′*)
apply (*auto simp*: *U twl-st-l-def hd-get-trail-twl-st-of-get-trail-l S*
  init-clss-l-mapsto-upd-irrel-notin learned-clss-l-mapsto-upd-notin convert-lit.simps
  intro: *convert-lits-l-extend-mono*)
 apply (*rule convert-lits-l-extend-mono*)
  apply *assumption*
apply *auto*
done
have [*simp*]: ⟨*Ex Not*⟩
 by *auto*
show *?thesis*
 unfolding *propagate-bt-l-def list-of-mset-def propagate-bt-def U RES-RETURN-RES*
  get-fresh-index-def RES-RES-RETURN-RES
 apply *clarify*
 apply (*rule RES-rule*)
 apply (*subst in-pair-collect-simp*)
 apply (*intro conjI*)
 subgoal using *Propa*
   by (*auto simp*: *hd-get-trail-twl-st-of-get-trail-l S T U*)
 subgoal by *auto*
 subgoal using *add-invs* ⟨*L* = *L′*⟩ **by** (*auto simp*: *S twl-list-invs-def MU simp del*: ⟨*L* = *L′*⟩)
 done
qed

have *propagate-unit-bt*:
 ⟨(*propagate-unit-bt-l* (*lit-of* (*hd* (*get-trail-l S*))) *U*,
  propagate-unit-bt (*lit-of* (*hd* (*get-trail S′*))) *U′*)
 ∈ {(*T*, *T′*). (*T*, *T′*) ∈ *twl-st-l None* ∧ *clauses-to-update-l T* = {#} ∧ *twl-list-invs T*}⟩
 if
  SS′: ⟨(*S*, *S′*) ∈ *?R*⟩ **and**
  bt-inv: ⟨*backtrack-l-inv S*⟩ **and**
  TT′: ⟨(*T*, *T′*) ∈ *?extract S* (*get-conflict-l S*)⟩ **and**
  UU′: ⟨(*U*, *U′*) ∈ *?find-decomp T*⟩ **and**
  size: ⟨¬*size* (*the* (*get-conflict-l U*)) > *1*⟩
 for *S T* :: ⟨*′v twl-st-l*⟩ **and** *S′ T′ U U′*
proof −
 obtain *MS NS DS NES UES* **where**
  S: ⟨*S* = (*MS*, *NS*, *Some DS*, *NES*, *UES*, {#}, {#})⟩
  using *SS′* **by** (*cases S*; *cases* ⟨*get-conflict-l S*⟩) *auto*

then obtain *DT* **where**
T: ‹$T = (MS, NS, Some\ DT, NES, UES, \{\#\}, \{\#\})$›
using TT' **by** (*cases T*; *cases* ‹*get-conflict-l T*›) *auto*
then obtain $MU\ MU'$ **where**
U: ‹$U = (MU, NS, Some\ DT, NES, UES, \{\#\}, \{\#\})$› **and**
MU: ‹$MS = MU'\ @\ MU$›
using UU' **by** (*cases U*) *auto*
have S'-S: ‹$(S, S') \in$ *twl-st-l None*›
using SS' **by** *simp*
have U'-U: ‹$(U, U') \in$ *twl-st-l None*›
using UU' **by** *simp*

have [*simp*]: ‹$MS \neq []$› **and** *add-invs*: ‹*twl-list-invs S*›
using SS' *bt-inv* **unfolding** *twl-list-invs-def backtrack-l-inv-def S* **by** *auto*
have *DT*: ‹$DT = \{\#-$ *lit-of* (*hd MS*)$\#\}$›
using TT' *size* **by** (*cases DT*, *auto simp*: *U T*)
show *?thesis*
apply (*subst in-pair-collect-simp*)
apply (*intro conjI*)
subgoal
using S'-S U'-U **apply** (*auto simp*: *twl-st-l-def propagate-unit-bt-def propagate-unit-bt-l-def S T U DT convert-lit.simps intro*: *convert-lits-l-extend-mono*)
apply (*rule convert-lits-l-extend-mono*)
apply *assumption*
by *auto*
subgoal by (*auto simp*: *propagate-unit-bt-def propagate-unit-bt-l-def S T U DT*)
subgoal using *add-invs* S'-S **unfolding** *S T U twl-list-invs-def propagate-unit-bt-l-def*
by (*auto 5 5 simp*: *propagate-unit-bt-l-def DT*
twl-list-invs-def MU twl-st-l-def)
done
qed

have *bt*:
‹(*backtrack-l*, *backtrack*) $\in$ *?R* $\rightarrow_f$
$\langle\{(T::'v$ *twl-st-l*, $T')$. $(T, T') \in$ *twl-st-l None* $\wedge$ *clauses-to-update-l* $T = \{\#\} \wedge$
twl-list-invs $T\}\rangle$ *nres-rel*›
(**is** ‹- $\in$ - $\rightarrow_f$ $\langle$*?I*$'\rangle$*nres-rel*›)
supply [[*goals-limit=1*]]
unfolding *backtrack-l-def backtrack-def fref-param1*[*symmetric*]
apply (*refine-vcg H list-of-mset ext*; *remove-dummy-vars*)
subgoal for $S\ S'$
unfolding *backtrack-l-inv-def*
apply (*rule-tac* $x=S'$ **in** *exI*)
by (*auto simp*: *backtrack-inv-def backtrack-l-inv-def twl-st-l*)
subgoal by (*auto simp*: *convert-lits-l-def elim*: *neq-NilE*)
subgoal unfolding *backtrack-inv-def* **by** *auto*
subgoal by *simp*
subgoal by (*auto simp*: *backtrack-inv-def equality-except-conflict-l-rewrite*)
subgoal by (*auto simp*: *hd-get-trail-twl-st-of-get-trail-l backtrack-l-inv-def*
equality-except-conflict-l-rewrite)
subgoal by (*auto simp*: *propagate-bt-l-def propagate-bt-def backtrack-l-inv-def*
equality-except-conflict-l-rewrite)
subgoal by *auto*
subgoal by (*rule find-lit*) *assumption+*
subgoal by (*rule propagate-bt*) *assumption+*
subgoal by (*rule propagate-unit-bt*) *assumption+*

done
have *SPEC-Id*: ‹*SPEC* Φ = ⇓ {(*T*, *T'*). Φ *T*} (*SPEC* Φ)› **for** Φ
unfolding *conc-fun-RES*
by *auto*
have ‹(*backtrack-l S*, *backtrack S'*) ∈ *?I*› **if** ‹(*S*, *S'*) ∈ *?R*› **for** *S S'*
proof −
have ‹*backtrack-l S* ≤ ⇓ *?I'* (*backtrack S'*)›
by (*rule bt*[*unfolded fref-param1*[*symmetric*], *to-*⇓, *rule-format*, *of S S'*])
(*use that* **in** *auto*)
moreover have ‹*backtrack S'* ≤ *SPEC* (λ*T*. *cdcl-twl-o S' T* ∧
get-conflict T = *None* ∧
(∀ *S'*. ¬ *cdcl-twl-o T S'*) ∧
twl-struct-invs T ∧
twl-stgy-invs T ∧ *clauses-to-update T* = {#} ∧ *literals-to-update T* ≠ {#})›
by (*rule backtrack-spec*[*to-*⇓, *of S'*]) (*use that* **in** ‹*auto simp*: *twl-st-l*›)
ultimately show *?thesis*
apply −
apply (*unfold refine-rel-defs nres-rel-def in-pair-collect-simp*;
(*unfold Ball2-split-def all-to-meta*)?;
(*intro allI impI*)?)
apply (*subst* (*asm*) *SPEC-Id*)
apply *unify-Down-invs2+*
unfolding *nofail-simps*
apply *unify-Down-invs2-normalisation-post*
apply (*rule weaken-*⇓)
prefer *2* **apply** *assumption*
subgoal premises *p* **by** (*auto simp*: *twl-st-l-def*)
done
qed
then show *?thesis*
by (*intro frefI*)
qed

definition *find-unassigned-lit-l* :: ‹*'v twl-st-l* ⇒ *'v literal option nres*› **where**
‹*find-unassigned-lit-l* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
SPEC (λ*L*.
(*L* ≠ *None* ⟶
undefined-lit M (*the L*) ∧
atm-of (*the L*) ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf N* + *NE*)) ∧
(*L* = *None* ⟶ (∄ *L'*. *undefined-lit M L'* ∧
atm-of L' ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf N* + *NE*))))
)›

definition *decide-l-or-skip-pre* **where**
‹*decide-l-or-skip-pre S* ⟷ (∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧
twl-struct-invs S' ∧
twl-stgy-invs S' ∧
twl-list-invs S ∧
get-conflict-l S = *None* ∧
clauses-to-update-l S = {#} ∧
literals-to-update-l S = {#})
›

definition *decide-lit-l* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*decide-lit-l* = (λ*L'* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).

$(Decided\ L'\ \#\ M,\ N,\ D,\ NE,\ UE,\ WS,\ \{\#-\ L'\#\}))\rangle$

definition *decide-l-or-skip* :: ‹′v twl-st-l ⇒ (bool × ′v twl-st-l) nres› **where**
‹*decide-l-or-skip S* = (*do* {
ASSERT(*decide-l-or-skip-pre S*);
L ← *find-unassigned-lit-l S*;
case L of
None ⇒ *RETURN* (*True*, *S*)
| *Some L* ⇒ *RETURN* (*False*, *decide-lit-l L S*)
})
›

method *match-⇓* =
(*match* **conclusion in** ‹$f \le \Downarrow R\ g$› **for** *f* :: ‹′a nres› **and** *R* :: ‹(′a × ′b) set› **and**
g :: ‹′b nres› ⇒
‹*match premises in*
I[*thin,uncurry*]: ‹$f \le \Downarrow R'\ g$› *for* R' :: ‹(′a × ′b) set›
⇒ ‹*rule refinement-trans-long*[*of f f g g R′ R*, *OF refl refl - I*]›
| *I*[*thin,uncurry*]: ‹- $\Longrightarrow f \le \Downarrow R'\ g$› *for* R' :: ‹(′a × ′b) set›
⇒ ‹*rule refinement-trans-long*[*of f f g g R′ R*, *OF refl refl - I*]›
›)

lemma *decide-l-or-skip-spec*:
‹(*decide-l-or-skip*, *decide-or-skip*) ∈
{(*S*, *S′*). (*S*, *S′*) ∈ *twl-st-l None* ∧ *get-conflict-l S* = *None* ∧
clauses-to-update-l S = {#} ∧ *literals-to-update-l S* = {#} ∧ *no-step cdcl-twl-cp S′* ∧
twl-struct-invs S′ ∧ *twl-stgy-invs S′* ∧ *twl-list-invs S*} $\rightarrow_f$
⟨{((*brk*, *T*), (*brk′*, *T′*)). (*T*, *T′*) ∈ *twl-st-l None* ∧ *brk* = *brk′* ∧ *twl-list-invs T* ∧
clauses-to-update-l T = {#} ∧
(*get-conflict-l T* ≠ *None* ⟶ *get-conflict-l T* = *Some* {#})∧
twl-struct-invs T′ ∧ *twl-stgy-invs T′* ∧
(¬*brk* ⟶ *literals-to-update-l T* ≠ {#})∧
(*brk* ⟶ *literals-to-update-l T* = {#})}⟩ *nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ ⟨*?S*⟩*nres-rel*›)
proof −
have *find-unassigned-lit-l*: ‹*find-unassigned-lit-l S* ≤ ⇓ *Id* (*find-unassigned-lit S′*)›
if *SS′*: ‹(*S*, *S′*) ∈ *?R*›
for *S S′*
using *that*
by (*cases S*)
(*auto simp*: *find-unassigned-lit-l-def find-unassigned-lit-def*
mset-take-mset-drop-mset′ image-image twl-st-l-def)

have *I*: ‹(*x*, *x′*) ∈ *Id* ⟹ (*x*, *x′*) ∈ ⟨*Id*⟩*option-rel*› **for** *x x′* **by** *auto*
have *dec*: ‹(*decide-l-or-skip*, *decide-or-skip*) ∈ *?R* →
⟨{((*brk*, *T*), (*brk′*, *T′*)). (*T*, *T′*) ∈ *twl-st-l None* ∧ *brk* = *brk′* ∧ *twl-list-invs T* ∧
clauses-to-update-l T = {#} ∧
(¬*brk* ⟶ *literals-to-update-l T* ≠ {#})∧
(*brk* ⟶ *literals-to-update-l T* = {#}) }⟩ *nres-rel*›
unfolding *decide-l-or-skip-def decide-or-skip-def*
apply (*refine-vcg find-unassigned-lit-l I*)
subgoal unfolding *decide-l-or-skip-pre-def* **by** (*auto simp*: *twl-st-l-def*)
subgoal by *auto*
subgoal for *S S′*
by (*cases S*)
(*auto simp*: *decide-lit-l-def propagate-dec-def twl-list-invs-def twl-st-l-def*)
done

have *KK*: ‹*SPEC* (λ(*brk*, *T*). *cdcl-twl-o*** *S′ T* ∧ *P brk T*) = ⇓ {(*S*, *S′*). *snd S* = *S′* ∧
P (*fst S*) (*snd S*)} (*SPEC* (*cdcl-twl-o*** *S′*))›
for *S′ P*
by (*auto simp*: *conc-fun-def*)
have *nf*: ‹*nofail* (*SPEC* (*cdcl-twl-o*** *S′*))› ‹*nofail* (*SPEC* (*cdcl-twl-o*** *S′*))› **for** *S S′*
by *auto*
have *set*: ‹{((*a*,*b*), (*a′*, *b′*)). *P a b a′ b′*} = {(*a*, *b*). *P* (*fst a*) (*snd a*) (*fst b*) (*snd b*)}› **for** *P*
by *auto*

show *?thesis*
proof (*intro frefI nres-relI*)
fix *S S′*
assume *SS′*: ‹(*S*, *S′*) ∈ *?R*›
have ‹*decide-l-or-skip S*
≤ ⇓ {((*brk*, *T*), *brk′*, *T′*).
(*T*, *T′*) ∈ *twl-st-l None* ∧
brk = *brk′* ∧
twl-list-invs T ∧
clauses-to-update-l T = {#} ∧
(¬ *brk* ⟶ *literals-to-update-l T* ≠ {#}) ∧ (*brk* ⟶ *literals-to-update-l T* = {#})}
(*decide-or-skip S′*)›
apply (*rule dec*[*to-*⇓, *of S S′*])
using *SS′* **by** *auto*
moreover have ‹ *decide-or-skip S′*
≤ ⇓ {(*S*, *S′a*).
snd S = *S′a* ∧
get-conflict (*snd S*) = *None* ∧
(∀ *S′*. ¬ *cdcl-twl-o* (*snd S*) *S′*) ∧
(*fst S* ⟶ (∀ *S′*. ¬ *cdcl-twl-stgy* (*snd S*) *S′*)) ∧
twl-struct-invs (*snd S*) ∧
twl-stgy-invs (*snd S*) ∧
clauses-to-update (*snd S*) = {#} ∧
(¬ *fst S* ⟶ *literals-to-update* (*snd S*) ≠ {#}) ∧
(¬ (∀ *S′a*. ¬ *cdcl-twl-o S′ S′a*) ⟶ *cdcl-twl-o*++ *S′* (*snd S*))}
(*SPEC* (*cdcl-twl-o*** *S′*))›
by (*rule decide-or-skip-spec*[*of S′*, *unfolded KK*]) (*use SS′* **in** *auto*)
ultimately show ‹*decide-l-or-skip S* ≤ ⇓ *?S* (*decide-or-skip S′*)›
apply −
apply *unify-Down-invs2+*
apply (*simp only*: *set nf*)
apply (*match-*⇓)
subgoal
apply (*rule*; *rule*)
apply (*clarsimp simp*: *twl-st-l-def*)
done
subgoal by *fast*
done
qed
qed

lemma *refinement-trans-eq*:
‹$A = A' \Longrightarrow B = B' \Longrightarrow R' = R \Longrightarrow A \le \Downarrow R\ B \Longrightarrow A' \le \Downarrow R'\ B'$›
by (*auto simp*: *pw-ref-iff*)

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

$(\exists S'. (S, S') \in$ *twl-st-l None* $\wedge$
twl-struct-invs $S' \wedge$
twl-stgy-invs $S' \wedge$
twl-list-invs S)›

definition *cdcl-twl-o-prog-l* :: ‹'v twl-st-l ⇒ (bool × 'v twl-st-l) nres› **where**
‹*cdcl-twl-o-prog-l* S =
do {
ASSERT(*cdcl-twl-o-prog-l-pre* S);
do {
if get-conflict-l S = *None*
then decide-l-or-skip S
else if count-decided (*get-trail-l* S) > *0*
then do {
$T \leftarrow$ *skip-and-resolve-loop-l* S;
ASSERT(*get-conflict-l* $T \neq$ *None* $\wedge$ *get-conflict-l* $T \neq$ *Some* {#});
$U \leftarrow$ *backtrack-l* T;
RETURN (*False*, U)
}
else RETURN (*True*, S)
}
}
›

lemma *twl-st-lE*:
‹($\bigwedge M\ N\ D\ NE\ UE\ WS\ Q.\ T = (M, N, D, NE, UE, WS, Q) \Longrightarrow P\ (M, N, D, NE, UE, WS, Q)$) $\Longrightarrow P\ T$›
for T :: ‹'a twl-st-l›
by (*cases* T) *auto*

lemma *weaken-*$\Downarrow$*'*: ‹$f \leq \Downarrow R'\ g \Longrightarrow R' \subseteq R \Longrightarrow f \leq \Downarrow R\ g$›
by (*meson pw-ref-iff subset-eq*)

lemma *cdcl-twl-o-prog-l-spec*:
‹(*cdcl-twl-o-prog-l*, *cdcl-twl-o-prog*) ∈
{(S, S'). (S, S') ∈ *twl-st-l None* ∧
clauses-to-update-l S = {#} ∧ *literals-to-update-l* S = {#} ∧ *no-step cdcl-twl-cp* S' ∧
twl-struct-invs S' ∧ *twl-stgy-invs* S' ∧ *twl-list-invs* S} $\rightarrow_f$
⟨{((*brk*, T), (*brk'*, T')). (T, T') ∈ *twl-st-l None* ∧ *brk* = *brk'* ∧ *twl-list-invs* T ∧
clauses-to-update-l T = {#} ∧
(*get-conflict-l* $T \neq$ *None* $\longrightarrow$ *count-decided* (*get-trail-l* T) = *0*)∧
twl-struct-invs T' ∧ *twl-stgy-invs* T'}⟩ *nres-rel*›
(**is** ‹ - ∈ *?R* $\rightarrow_f$ *?I*› **is** ‹ - ∈ *?R* $\rightarrow_f$ ⟨*?J*⟩*nres-rel*›)
proof −
have *twl-prog*:
‹(*cdcl-twl-o-prog-l*, *cdcl-twl-o-prog*) ∈ *?R* $\rightarrow_f$
⟨{((*brk*, S), (*brk'*, S')).
(*brk* = *brk'* ∧ (S, S') ∈ *twl-st-l None*) ∧ *twl-list-invs* S ∧
clauses-to-update-l S = {#}}⟩ *nres-rel*›
(**is** ‹- ∈ - $\rightarrow_f$ ⟨*?I'*⟩ *nres-rel*›)
supply [[*goals-limit=3*]]
unfolding *cdcl-twl-o-prog-l-def cdcl-twl-o-prog-def*
find-unassigned-lit-def fref-param1[*symmetric*]
apply (*refine-vcg*

decide-l-or-skip-spec[*THEN fref-to-Down*, *THEN weaken-*$\Downarrow$*'*]
skip-and-resolve-loop-l-spec[*THEN fref-to-Down*]
backtrack-l-spec[*THEN fref-to-Down*]; *remove-dummy-vars*)
subgoal for S S'
unfolding *cdcl-twl-o-prog-l-pre-def* **by** (*rule exI*[*of - S'*]) (*force simp*: *twl-st-l*)
subgoal by *auto*
subgoal by *simp*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
have *set*: ‹$\{((a,b), (a', b')).\ P\ a\ b\ a'\ b'\} = \{(a, b).\ P\ (fst\ a)\ (snd\ a)\ (fst\ b)\ (snd\ b)\}$› **for** P
by *auto*
have *SPEC-Id*: ‹$SPEC\ \Phi = \Downarrow \{(T, T').\ \Phi\ T\}\ (SPEC\ \Phi)$› **for** Φ
unfolding *conc-fun-RES*
by *auto*
show *bt'*: *?thesis*
proof (*intro frefI nres-relI*)
fix S S'
assume *SS'*: ‹$(S, S') \in$ *?R*›
have ‹*cdcl-twl-o-prog* $S' \le$ *SPEC* (*cdcl-twl-o-prog-spec* S')›
by (*rule cdcl-twl-o-prog-spec*[*of S'*]) (*use SS'* **in** *auto*)
moreover have ‹*cdcl-twl-o-prog-l* $S \le \Downarrow$ *?I'* (*cdcl-twl-o-prog* S')›
by (*rule twl-prog*[*unfolded fref-param1*[*symmetric*], *to-*$\Downarrow$])
(*use SS'* **in** *auto*)
ultimately show ‹*cdcl-twl-o-prog-l* $S \le \Downarrow$ *?J* (*cdcl-twl-o-prog* S')›
apply −
unfolding *set*
apply (*subst*(*asm*) *SPEC-Id*)
apply *unify-Down-invs2*+
apply (*match-*$\Downarrow$)
subgoal by (*clarsimp simp del*: *split-paired-All simp*: *twl-st-l-def*)
subgoal by *simp*
done
qed
qed

1.3.3 Full Strategy

definition *cdcl-twl-stgy-prog-l-inv* :: ‹*'v twl-st-l* ⇒ *bool* × *'v twl-st-l* ⇒ *bool*› **where**
‹*cdcl-twl-stgy-prog-l-inv* $S_0 \equiv \lambda(brk, T).\ \exists S_0'\ T'.\ (T, T') \in$ *twl-st-l None* ∧
$(S_0, S_0') \in$ *twl-st-l None* ∧
twl-struct-invs T' ∧
twl-stgy-invs T' ∧
(*brk* ⟶ *final-twl-state* T') ∧
cdcl-twl-stgy** S_0' T' ∧
clauses-to-update-l $T = \{\#\}$ ∧
(¬*brk* ⟶ *get-conflict-l* T = *None*)›

definition *cdcl-twl-stgy-prog-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*cdcl-twl-stgy-prog-l* S_0 =

do {
  do {
    (*brk*, *T*) ← $WHILE_T{}^{cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}inv\ S_0}$
      (λ(*brk*, -). ¬*brk*)
      (λ(*brk*, *S*).
      do {
        T ← *unit-propagation-outer-loop-l S*;
        cdcl-twl-o-prog-l T
      })
      (*False*, S_0);
    RETURN T
  }
}
›

lemma *cdcl-twl-stgy-prog-l-spec*:
‹(*cdcl-twl-stgy-prog-l*, *cdcl-twl-stgy-prog*) ∈
  {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
    clauses-to-update-l S = {#} ∧
    twl-struct-invs S' ∧ *twl-stgy-invs S'*} $\to_f$
  ⟨{(*T*, *T'*). (*T*, *T'*) ∈ {(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs T* ∧
    twl-struct-invs T' ∧ *twl-stgy-invs T'*} ∧ *True*}⟩ *nres-rel*›
(**is** ‹ - ∈ *?R* $\to_f$ *?I*› **is** ‹ - ∈ *?R* $\to_f$ ⟨*?J*⟩*nres-rel*›)
proof −
  have *R*: ‹(*a*, *b*) ∈ *?R* ⟹
    ((*False*, *a*), (*False*, *b*)) ∈ {((*brk*, *S*), (*brk'*, *S'*)). *brk* = *brk'* ∧ (*S*, *S'*) ∈ *?R*}›
    for *a b* **by** *auto*

  show *?thesis*
    unfolding *cdcl-twl-stgy-prog-l-def cdcl-twl-stgy-prog-def cdcl-twl-o-prog-l-spec*
      fref-param1[*symmetric*] *cdcl-twl-stgy-prog-l-inv-def*
    apply (*refine-rcg R cdcl-twl-o-prog-l-spec*[*THEN fref-to-Down*, *THEN weaken-⇓'*]
        unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]; *remove-dummy-vars*)
    subgoal for S_0 S_0' *T T'*
      apply (*rule exI*[*of* - S_0'])
      apply (*rule exI*[*of* - ‹*snd T*›])
      by (*auto simp add*: *case-prod-beta*)
    subgoal by *auto*
    subgoal by *fastforce*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    done
qed

lemma *refine-pair-to-SPEC*:
  fixes *f* :: ‹*'s* ⇒ *'s nres*› **and** *g* :: ‹*'b* ⇒ *'b nres*›
  assumes ‹(*f*, *g*) ∈ {(*S*, *S'*). (*S*, *S'*) ∈ *H* ∧ *R S S'*} $\to_f$ ⟨{(*S*, *S'*). (*S*, *S'*) ∈ *H'* ∧ *P' S*}⟩*nres-rel*›
    (**is** ‹- ∈ *?R* $\to_f$ *?I*›)
  assumes ‹*R S S'*› **and** [*simp*]: ‹(*S*, *S'*) ∈ *H*›
  shows ‹*f S* ≤ ⇓ {(*S*, *S'*). (*S*, *S'*) ∈ *H'* ∧ *P' S*} (*g S'*)›
proof −
  have ‹(*f S*, *g S'*) ∈ *?I*›
    using *assms* **unfolding** *fref-def nres-rel-def* **by** *auto*
  then show *?thesis*
    unfolding *nres-rel-def fun-rel-def pw-le-iff pw-conc-inres pw-conc-nofail*

by *auto*
qed

definition *cdcl-twl-stgy-prog-l-pre* **where**
‹*cdcl-twl-stgy-prog-l-pre S S'* ⟷
((*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'* ∧ *twl-stgy-invs S'* ∧
clauses-to-update-l S = {#} ∧ *get-conflict-l S* = *None* ∧ *twl-list-invs S*)›

lemma *cdcl-twl-stgy-prog-l-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-l-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-l S* ≤ ⇓ (*twl-st-l None*) (*conclusive-TWL-run S'*)›
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-l-spec*[*THEN refine-pair-to-SPEC*,
of S S']])
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal
apply (*rule ref-two-step*)
prefer *2*
apply (*rule cdcl-twl-stgy-prog-spec*)
using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** (*auto intro*: *conc-fun-R-mono*)
done

lemma *cdcl-twl-stgy-prog-l-spec-final'*:
assumes
‹*cdcl-twl-stgy-prog-l-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-l S* ≤ ⇓ {(*S*, *T*). (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
twl-struct-invs S' ∧ *twl-stgy-invs S'*} (*conclusive-TWL-run S'*)›
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-l-spec*[*THEN refine-pair-to-SPEC*,
of S S']])
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal
apply (*rule ref-two-step*)
prefer *2*
apply (*rule cdcl-twl-stgy-prog-spec*)
using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** (*auto intro*: *conc-fun-R-mono*)
done

definition *cdcl-twl-stgy-prog-break-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*cdcl-twl-stgy-prog-break-l* S_0 =
do {
b ← *SPEC*(λ-. *True*);
(*b*, *brk*, *T*) ← $WHILE_T{}^{\lambda(b,\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}inv\ S_0\ S}$
(λ(*b*, *brk*, -). *b* ∧ ¬*brk*)
(λ(-, *brk*, *S*). *do* {
T ← *unit-propagation-outer-loop-l S*;
T ← *cdcl-twl-o-prog-l T*;
b ← *SPEC*(λ-. *True*);
RETURN (*b*, *T*)
})
(*b*, *False*, S_0);
if brk then RETURN T
else cdcl-twl-stgy-prog-l T

}›

lemma *cdcl-twl-stgy-prog-break-l-spec*:
‹(*cdcl-twl-stgy-prog-break-l*, *cdcl-twl-stgy-prog-break*) ∈
{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#} ∧
twl-struct-invs S' ∧ *twl-stgy-invs S'*} $\rightarrow_f$
⟨{(*T*, *T'*). (*T*, *T'*) ∈ {(*T*, *T'*). (*T*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs T* ∧
twl-struct-invs T' ∧ *twl-stgy-invs T'*} ∧ *True*}⟩ *nres-rel*›
(**is** ‹ - ∈ *?R* $\rightarrow_f$ *?I*› **is** ‹ - ∈ *?R* $\rightarrow_f$ ⟨*?J*⟩*nres-rel*›)
proof −
have *R*: ‹(*a*, *b*) ∈ *?R* ⟹ (*bb*, *bb'*) ∈ *bool-rel* ⟹
((*bb*, *False*, *a*), (*bb'*, *False*, *b*)) ∈ {((*b*, *brk*, *S*), (*b'*, *brk'*, *S'*)). *b* = *b'* ∧ *brk* = *brk'* ∧
(*S*, *S'*) ∈ *?R*}›
for *a b bb bb'* **by** *auto*

show *?thesis*
supply [[*goals-limit=1*]]
unfolding *cdcl-twl-stgy-prog-break-l-def cdcl-twl-stgy-prog-break-def cdcl-twl-o-prog-l-spec*
fref-param1[*symmetric*] *cdcl-twl-stgy-prog-l-inv-def*
apply (*refine-rcg cdcl-twl-o-prog-l-spec*[*THEN fref-to-Down*]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-stgy-prog-l-spec[*THEN fref-to-Down*]; *remove-dummy-vars*)
apply (*rule R*)
subgoal by *auto*
subgoal by *auto*
subgoal for S_0 S_0' *b b' T T'*
apply (*rule exI*[*of* - S_0'])
apply (*rule exI*[*of* - ‹*snd* (*snd T*)›])
by (*auto simp add*: *case-prod-beta*)
subgoal
by *auto*
subgoal by *fastforce*
subgoal by (*auto simp*: *twl-st-l*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

lemma *cdcl-twl-stgy-prog-break-l-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-l-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-break-l S* ≤ ⇓ (*twl-st-l None*) (*conclusive-TWL-run S'*)›
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-break-l-spec*[*THEN refine-pair-to-SPEC*,
of S S']])
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def* **by** *auto*
subgoal
apply (*rule ref-two-step*)
prefer *2*
apply (*rule cdcl-twl-stgy-prog-break-spec*)
using *assms* **unfolding** *cdcl-twl-stgy-prog-l-pre-def*
by (*auto intro*: *conc-fun-R-mono*)

done

end
theory *Watched-Literals-List-Restart*
imports *Watched-Literals-List Watched-Literals-Algorithm-Restart*
begin

Unlike most other refinements steps we have done, we don't try to refine our specification to our code directly: We first introduce an intermediate transition system which is closer to what we want to implement. Then we refine it to code.

This invariant abstract over the restart operation on the trail. There can be a backtracking on the trail and there can be a renumbering of the indexes.

inductive *valid-trail-reduction* **for** $M\ M'$ :: ‹('v ,'c) ann-lits› **where**
backtrack-red:
‹valid-trail-reduction M M'›
if
‹(Decided K # M'', M2) ∈ set (get-all-ann-decomposition M)› **and**
‹map lit-of M'' = map lit-of M'› **and**
‹map is-decided M'' = map is-decided M'› |
keep-red:
‹valid-trail-reduction M M'›
if
‹map lit-of M = map lit-of M'› **and**
‹map is-decided M = map is-decided M'›

lemma *valid-trail-reduction-simps*: ‹valid-trail-reduction M M' ⟷
((∃ K M'' M2. (Decided K # M'', M2) ∈ set (get-all-ann-decomposition M) ∧
map lit-of M'' = map lit-of M' ∧ map is-decided M'' = map is-decided M' ∧
length M' = length M'') ∨
map lit-of M = map lit-of M' ∧ map is-decided M = map is-decided M' ∧ length M = length M')›
apply (*auto simp*: *valid-trail-reduction.simps dest*: *arg-cong*[*of* ‹map lit-of -› - *length*])
apply (*force dest*: *arg-cong*[*of* ‹map lit-of -› - *length*])+
done

lemma *trail-changes-same-decomp*:
assumes
M'-lit: ‹map lit-of M' = map lit-of ysa @ L # map lit-of zsa› **and**
M'-dec: ‹map is-decided M' = map is-decided ysa @ False # map is-decided zsa›
obtains *C' M2 M1* **where** ‹M' = M2 @ Propagated L C' # M1› **and**
‹map lit-of M2 = map lit-of ysa›**and**
‹map is-decided M2 = map is-decided ysa› **and**
‹map lit-of M1 = map lit-of zsa› **and**
‹map is-decided M1 = map is-decided zsa›
proof −
define *M1 M2 K* **where** ‹M1 ≡ drop (Suc (length ysa)) M'› **and** ‹M2 ≡ take (length ysa) M'› **and**
‹K ≡ hd (drop (length ysa) M')›
have
M': ‹M' = M2 @ K # M1›
using *arg-cong*[*OF M'-lit*, *of length*] **unfolding** *M1-def M2-def K-def*
by (*simp add*: *Cons-nth-drop-Suc hd-drop-conv-nth*)
have [*simp*]:
‹length M2 = length ysa›
‹length M1 = length zsa›
using *arg-cong*[*OF M'-lit*, *of length*] **unfolding** *M1-def M2-def K-def* **by** *auto*

obtain C' **where**
[*simp*]: ‹K = *Propagated* L C'›
using M'-*lit* M'-*dec* **unfolding** M'
by (*cases* K) *auto*

show *?thesis*
using *that*[*of* $M2$ C' $M1$] M'-*lit* M'-*dec* **unfolding** M'
by *auto*
qed

lemma
assumes
‹*map lit-of* M = *map lit-of* M'› **and**
‹*map is-decided* M = *map is-decided* M'›
shows
trail-renumber-count-dec:
‹*count-decided* M = *count-decided* M'› **and**
trail-renumber-get-level:
‹*get-level* M L = *get-level* M' L›
proof −
have [*dest*]: ‹*count-decided* M = *count-decided* M'›
if ‹*map is-decided* M = *map is-decided* M'› **for** M M'
using *that*
apply (*induction* M *arbitrary*: M' *rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for L M M'
by (*cases* M')
(*auto simp*: *get-level-cons-if*)
subgoal for L C M M'
by (*cases* M')
(*auto simp*: *get-level-cons-if*)
done
then show ‹*count-decided* M = *count-decided* M'› **using** *assms* **by** *blast*
show ‹*get-level* M L = *get-level* M' L›
using *assms*
apply (*induction* M *arbitrary*: M' *rule*: *ann-lit-list-induct*)
subgoal by *auto*
subgoal for L M M'
by (*cases* M'; *cases* ‹*hd* M'›)
(*auto simp*: *get-level-cons-if*)
subgoal for L C M M'
by (*cases* M')
(*auto simp*: *get-level-cons-if*)
done
qed

lemma *valid-trail-reduction-Propagated-inD*:
‹*valid-trail-reduction* M M' $\Longrightarrow$ *Propagated* L C $\in$ *set* M' $\Longrightarrow$ $\exists$ C'. *Propagated* L C' $\in$ *set* M›
by (*induction rule*: *valid-trail-reduction.induct*)
(*force dest*!: *get-all-ann-decomposition-exists-prepend*
dest!: *split-list*[*of* ‹*Propagated* L C›] *elim*!: *trail-changes-same-decomp*)+

lemma *valid-trail-reduction-Propagated-inD2*:
‹*valid-trail-reduction* M M' $\Longrightarrow$ *length* M = *length* M' $\Longrightarrow$ *Propagated* L C $\in$ *set* M $\Longrightarrow$
$\exists$ C'. *Propagated* L C' $\in$ *set* M'›

apply (*induction rule*: *valid-trail-reduction.induct*)
apply (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
dest!: *split-list*[*of* ‹*Propagated L C*›] *elim*!: *trail-changes-same-decomp*)+
apply (*metis add-Suc-right le-add2 length-Cons length-append length-map not-less-eq-eq*)
by (*metis* (*no-types*, *lifting*) *in-set-conv-decomp trail-changes-same-decomp*)

lemma *get-all-ann-decomposition-change-annotation-exists*:
assumes
‹(*Decided K* # *M′*, *M2′*) ∈ *set* (*get-all-ann-decomposition M2*)› **and**
‹*map lit-of M1* = *map lit-of M2*› **and**
‹*map is-decided M1* = *map is-decided M2*›
shows ‹∃ *M″ M2′*. (*Decided K* # *M″*, *M2′*) ∈ *set* (*get-all-ann-decomposition M1*) ∧
map lit-of M″ = *map lit-of M′* ∧ *map is-decided M″* = *map is-decided M′*›
using *assms*
proof (*induction M1 arbitrary*: *M2 M2′ rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Decided L xs M2*)
then show *?case*
by (*cases M2*; *cases* ‹*hd M2*›) *fastforce*+
next
case (*Propagated L m xs M2*) **note** *IH* = *this*(*1*) **and** *prems* = *this*(*2*−)
show *?case*
using *IH*[*of* - ‹*tl M2*›] *prems get-all-ann-decomposition-decomp*[*of xs*]
get-all-ann-decomposition-decomp[*of M2* ‹*Decided K* # *M′*›]
by (*cases M2*; *cases* ‹*hd M2*›; *cases* ‹(*get-all-ann-decomposition* (*tl M2*))›;
cases ‹*hd* (*get-all-ann-decomposition xs*)›; *cases* ‹*get-all-ann-decomposition xs*›)
fastforce+
qed

lemma *valid-trail-reduction-trans*:
assumes
M1-M2: ‹*valid-trail-reduction M1 M2*› **and**
M2-M3: ‹*valid-trail-reduction M2 M3*›
shows ‹*valid-trail-reduction M1 M3*›
proof −
consider
(*same*) ‹*map lit-of M2* = *map lit-of M3*› **and**
‹*map is-decided M2* = *map is-decided M3*› ‹*length M2* = *length M3*› |
(*decomp-M1*) *K M″ M2′* **where**
‹(*Decided K* # *M″*, *M2′*) ∈ *set* (*get-all-ann-decomposition M2*)› **and**
‹*map lit-of M″* = *map lit-of M3*› **and**
‹*map is-decided M″* = *map is-decided M3*› **and**
‹*length M3* = *length M″*›
using *M2-M3* **unfolding** *valid-trail-reduction-simps*
by *auto*
note *decomp-M2* = *this*
consider
(*same*) ‹*map lit-of M1* = *map lit-of M2*› **and**
‹*map is-decided M1* = *map is-decided M2*› ‹*length M1* = *length M2*› |
(*decomp-M1*) *K M″ M2′* **where**
‹(*Decided K* # *M″*, *M2′*) ∈ *set* (*get-all-ann-decomposition M1*)› **and**
‹*map lit-of M″* = *map lit-of M2*› **and**
‹*map is-decided M″* = *map is-decided M2*› **and**
‹*length M2* = *length M″*›

```
    using M1-M2 unfolding valid-trail-reduction-simps
    by auto
  then show ?thesis
  proof cases
    case same
    from decomp-M2
    show ?thesis
    proof cases
      case same': same
      then show ?thesis
        using same by (auto simp: valid-trail-reduction-simps)
    next
      case decomp-M1 note decomp = this(1) and eq = this(2,3) and [simp] = this(4)
      obtain M4 M5 where
          decomp45: ‹(Decided K # M4, M5) ∈ set (get-all-ann-decomposition M1)› and
          M4-lit: ‹map lit-of M4 = map lit-of M''› and
          M4-dec: ‹map is-decided M4 = map is-decided M''›
        using get-all-ann-decomposition-change-annotation-exists[OF decomp, of M1] eq same
        by (auto simp: valid-trail-reduction-simps)
      show ?thesis
        by (rule valid-trail-reduction.backtrack-red[OF decomp45])
          (use M4-lit M4-dec eq same in auto)
    qed
  next
    case decomp-M1 note decomp = this(1) and eq = this(2,3) and [simp] = this(4)
    from decomp-M2
    show ?thesis
    proof cases
      case same
      obtain M4 M5 where
          decomp45: ‹(Decided K # M4, M5) ∈ set (get-all-ann-decomposition M1)› and
          M4-lit: ‹map lit-of M4 = map lit-of M''› and
          M4-dec: ‹map is-decided M4 = map is-decided M''›
        using get-all-ann-decomposition-change-annotation-exists[OF decomp, of M1] eq same
        by (auto simp: valid-trail-reduction-simps)
      show ?thesis
        by (rule valid-trail-reduction.backtrack-red[OF decomp45])
          (use M4-lit M4-dec eq same in auto)
    next
     case (decomp-M1 K' M''' M2''') note decomp' = this(1) and eq' = this(2,3) and [simp] = this(4)
      obtain M4 M5 where
          decomp45: ‹(Decided K' # M4, M5) ∈ set (get-all-ann-decomposition M'')› and
          M4-lit: ‹map lit-of M4 = map lit-of M'''› and
          M4-dec: ‹map is-decided M4 = map is-decided M'''›
        using get-all-ann-decomposition-change-annotation-exists[OF decomp', of M''] eq
        by (auto simp: valid-trail-reduction-simps)
      obtain M6 where
        decomp45: ‹(Decided K' # M4, M6) ∈ set (get-all-ann-decomposition M1)›
        using get-all-ann-decomposition-exists-prepend[OF decomp45]
          get-all-ann-decomposition-exists-prepend[OF decomp]
          get-all-ann-decomposition-ex[of K' M4 ‹- @ M2' @ Decided K # - @ M5›]
        by (auto simp: valid-trail-reduction-simps)
      show ?thesis
        by (rule valid-trail-reduction.backtrack-red[OF decomp45])
          (use M4-lit M4-dec eq decomp-M1 in auto)
    qed
```

  qed
qed

lemma *valid-trail-reduction-length-leD*: ‹*valid-trail-reduction M M'* $\Longrightarrow$ *length M'* $\leq$ *length M*›
  by (*auto simp*: *valid-trail-reduction-simps*)

lemma *valid-trail-reduction-level0-iff*:
  assumes *valid*: ‹*valid-trail-reduction M M'*› **and** *n-d*: ‹*no-dup M*›
  shows ‹($L \in$ *lits-of-l M* $\wedge$ *get-level M L* = *0*) $\longleftrightarrow$ ($L \in$ *lits-of-l M'* $\wedge$ *get-level M' L* = *0*)›
proof −
  have *H*[*intro*]: ‹*map lit-of M* = *map lit-of M'* $\Longrightarrow$ $L \in$ *lits-of-l M* $\Longrightarrow$ $L \in$ *lits-of-l M'*› **for** *M M'*
    by (*metis lits-of-def set-map*)
  have [*dest*]: ‹*undefined-lit c L* $\Longrightarrow$ $L \in$ *lits-of-l c* $\Longrightarrow$ *False*› **for** *c*
    by (*auto dest*: *in-lits-of-l-defined-litD*)

  show *?thesis*
    using *valid*
  proof *cases*
    case *keep-red*
    then show *?thesis*
      by (*metis H trail-renumber-get-level*)
  next
    case (*backtrack-red K M'' M2*) **note** *decomp* = *this*(*1*) **and** *eq* = *this*(*2*,*3*)
    obtain *M3* **where** *M*: ‹*M* = *M3* @ *Decided K* # *M''*›
      using *decomp* **by** *auto*
    have ‹($L \in$ *lits-of-l M* $\wedge$ *get-level M L* = *0*) $\longleftrightarrow$
      ($L \in$ *lits-of-l M''* $\wedge$ *get-level M'' L* = *0*)›
      using *n-d* **unfolding** *M*
      by (*auto 4 4 simp*: *valid-trail-reduction-simps get-level-append-if get-level-cons-if*
          atm-of-eq-atm-of
        dest: *in-lits-of-l-defined-litD no-dup-append-in-atm-notin*
        split: *if-splits*)
    also have ‹... $\longleftrightarrow$ ($L \in$ *lits-of-l M'* $\wedge$ *get-level M' L* = *0*)›
      using *eq* **by** (*metis local.H trail-renumber-get-level*)
    finally show *?thesis*
      by *blast*
  qed
qed

lemma *map-lit-of-eq-defined-litD*: ‹*map lit-of M* = *map lit-of M'* $\Longrightarrow$ *defined-lit M* = *defined-lit M'*›
  apply (*induction M arbitrary*: *M'*)
  subgoal by *auto*
  subgoal for *L M M'*
    by (*cases M'*; *cases L*; *cases hd M'*)
      (*auto simp*: *defined-lit-cons*)
  done

lemma *map-lit-of-eq-no-dupD*: ‹*map lit-of M* = *map lit-of M'* $\Longrightarrow$ *no-dup M* = *no-dup M'*›
  apply (*induction M arbitrary*: *M'*)
  subgoal by *auto*
  subgoal for *L M M'*
    by (*cases M'*; *cases L*; *cases hd M'*)
      (*auto dest*: *map-lit-of-eq-defined-litD*)
  done

Remarks about the predicate:

- The cases $\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow E = (0::'b) \longrightarrow E' \neq (0::'c) \longrightarrow P$ are already covered by the invariants (where P means that there is clause which was already present before).

inductive *cdcl-twl-restart-l* :: ‹'v twl-st-l ⇒ 'v twl-st-l ⇒ bool› **where**
restart-trail:
  ‹*cdcl-twl-restart-l* (M, N, None, NE, UE, {#}, Q)
    (M', N', None, NE + mset '# NE', UE + mset '# UE', {#}, Q')›
  if
    ‹*valid-trail-reduction* M M'› **and**
    ‹*init-clss-lf* N = *init-clss-lf* N' + NE'› **and**
    ‹*learned-clss-lf* N' + UE' ⊆# *learned-clss-lf* N› **and**
    ‹∀ E∈# (NE'+UE'). ∃ L∈set E. L ∈ *lits-of-l* M ∧ *get-level* M L = 0› **and**
    ‹∀ L E E' . Propagated L E ∈ set M' ⟶ Propagated L E' ∈ set M ⟶ E > 0 ⟶ E' > 0 ⟶
      E ∈# *dom-m* N' ∧ N' ∝ E = N ∝ E'› **and**
    ‹∀ L E E'. Propagated L E ∈ set M' ⟶ Propagated L E' ∈ set M ⟶ E = 0 ⟶ E' ≠ 0 ⟶
      mset (N ∝ E') ∈# NE + mset '# NE' + UE + mset '# UE'› **and**
    ‹∀ L E E'. Propagated L E ∈ set M' ⟶ Propagated L E' ∈ set M ⟶ E' = 0 ⟶ E = 0› **and**
    ‹0 ∉# *dom-m* N'› **and**
    ‹if length M = length M' then Q = Q' else Q' = {#}›

lemma *cdcl-twl-restart-l-list-invs*:
  assumes
    ‹*cdcl-twl-restart-l* S T› **and**
    ‹*twl-list-invs* S›
  shows
    ‹*twl-list-invs* T›
  using *assms*
proof (*induction rule*: *cdcl-twl-restart-l.induct*)
  case (*restart-trail* M M' N N' NE' UE' NE UE Q Q') **note** *red* = *this*(1) **and** *init* = *this*(2) **and**
    learned = *this*(3) **and** *NUE* = *this*(4) **and** *tr-ge0* = *this*(5) **and** *tr-new0* = *this*(6) **and**
    tr-still0 = *this*(7) **and** *dom0* = *this*(8) **and** *QQ'* = *this*(9) **and** *list-invs* = *this*(10)
  let ?S = ‹(M, N, None, NE, UE, {#}, Q)›
  let ?T = ‹(M', N', None, NE + mset '# NE', UE + mset '# UE', {#}, Q')›
  show ?*case*
    unfolding *twl-list-invs-def*
  proof (*intro conjI impI allI ballI*)
    fix C
    assume ‹C ∈# *clauses-to-update-l* ?T›
    then show ‹C ∈# *dom-m* (*get-clauses-l* ?T)›
      by *simp*
  next
    show ‹0 ∉# *dom-m* (*get-clauses-l* ?T)›
      using *dom0* **by** *simp*
  next
    fix L C
    assume *LC*: ‹Propagated L C ∈ set (*get-trail-l* ?T)› **and** *C0*: ‹0 < C›
    then obtain C' **where** *LC'*: ‹Propagated L C' ∈ set (*get-trail-l* ?S)›
      using *red* **by** (*auto dest*!: *valid-trail-reduction-Propagated-inD*)
    moreover have *C'0*: ‹C' ≠ 0›
      apply (*rule ccontr*)
      using *C0 tr-still0 LC LC'*

by (*auto simp*: *twl-list-invs-def*
dest!: *valid-trail-reduction-Propagated-inD*)
ultimately have *C-dom*: ‹$C \in\# $ *dom-m* (*get-clauses-l ?T*)› **and** *NCC'*: ‹$N' \propto C = N \propto C'$›
using *tr-ge0 C0 LC* **by** *auto*
show ‹$C \in\# $ *dom-m* (*get-clauses-l ?T*)›
using *C-dom* **.**

have
L-watched: ‹$L \in$ *set* (*watched-l* (*get-clauses-l ?S* $\propto C'$))› **and**
L-C'0: ‹*length* (*get-clauses-l ?S* $\propto C'$) $> 2 \Longrightarrow L =$ *get-clauses-l ?S* $\propto C'$! *0*›
using *list-invs C'0 LC'* **unfolding** *twl-list-invs-def*
by *auto*
show ‹$L \in$ *set* (*watched-l* (*get-clauses-l ?T* $\propto C$))›
using *L-watched NCC'* **by** *simp*

show ‹*length* (*get-clauses-l ?T* $\propto C$) $> 2 \Longrightarrow L =$ *get-clauses-l ?T* $\propto C$! *0*›
using *L-C'0 NCC'* **by** *simp*
next
show ‹*distinct-mset* (*clauses-to-update-l ?T*)›
by *auto*
qed
qed

lemma *rtranclp-cdcl-twl-restart-l-list-invs*:
assumes
‹*cdcl-twl-restart-l*** *S T*› **and**
‹*twl-list-invs S*›
shows
‹*twl-list-invs T*›
using *assms* **by** *induction* (*auto intro*: *cdcl-twl-restart-l-list-invs*)

lemma *cdcl-twl-restart-l-cdcl-twl-restart*:
assumes *ST*: ‹(*S*, *T*) $\in$ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*›
shows ‹*SPEC* (*cdcl-twl-restart-l S*) $\le \Downarrow$ {(*S*, *S'*). (*S*, *S'*) $\in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$
clauses-to-update-l S = {#}}
(*SPEC* (*cdcl-twl-restart T*))›
proof –
have [*simp*]: ‹*set* (*watched-l x*) $\cup$ *set* (*unwatched-l x*) = *set x*› **for** *x*
by (*metis append-take-drop-id set-append*)
have ‹$\exists$ *T'*. *cdcl-twl-restart T T'* $\wedge$ (*S'*, *T'*) $\in$ *twl-st-l None*›
if ‹*cdcl-twl-restart-l S S'*› **for** *S'*
using *that ST struct-invs*
proof (*induction rule*: *cdcl-twl-restart-l.induct*)
case (*restart-trail M M' N N' NE' UE' NE UE Q Q'*) **note** *red* = *this*(*1*) **and** *init* = *this*(*2*) **and**
learned = *this*(*3*) **and** *NUE* = *this*(*4*) **and** *tr-ge0* = *this*(*5*) **and** *tr-new0* = *this*(*6*) **and**
tr-still0 = *this*(*7*) **and** *dom0* = *this*(*8*) **and** *QQ'* = *this*(*9*) **and** *ST* = *this*(*10*) **and**
struct-invs = *this*(*11*)
let *?T'* = ‹(*drop* (*length M* – *length M'*) (*get-trail T*), *twl-clause-of* ‘# *init-clss-lf N'*,
twl-clause-of ‘# *learned-clss-lf N'*, *None*, *NE*+*mset* ‘# *NE'*, *UE*+*mset* ‘# *UE'*, {#}, *Q'*)›
have [*intro*]: ‹$Q \ne Q' \Longrightarrow Q' = $ {#}›
using *QQ'* **by** (*auto split*: *if-splits*)
obtain *TM* **where**
T: ‹*T* = (*TM*, *twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*, *None*,

NE, *UE*, {#}, *Q*)⟩ **and**
M-TM: ⟨(*M*, *TM*) ∈ *convert-lits-l N* (*NE*+*UE*)⟩
using *ST*
by (*cases T*) (*auto simp*: *twl-st-l-def*)
have ⟨*no-dup TM*⟩
using *struct-invs* **unfolding** *T twl-struct-invs-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by (*simp add*: *trail.simps*)
then have *n-d*: ⟨*no-dup M*⟩
using *M-TM* **by** *auto*
have ⟨*cdcl-twl-restart T ?T'*⟩
using *red*
proof (*induction*)
case *keep-red*
from *arg-cong*[*OF this*(*1*), *of length*] **have** [*simp*]: ⟨*length M* = *length M'*⟩ **by** *simp*
have [*simp*]: ⟨*Q* = *Q'*⟩
using *QQ'* **by** *simp*
have *annot-in-clauses*: ⟨∀ *L E*. *Propagated L E* ∈ *set TM* ⟶
E ∈# *clauses*
(*twl-clause-of* ‘# *init-clss-lf N* +
twl-clause-of ‘# *learned-clss-lf N'*) +
NE +
UE +
clauses (*twl-clause-of* ‘# *UE'*)⟩
proof (*intro allI impI conjI*)
fix *L E*
assume ⟨*Propagated L E* ∈ *set TM*⟩
then obtain *E'* **where** *LE'-M*: ⟨*Propagated L E'* ∈ *set M*⟩ **and**
E-E': ⟨*convert-lit N* (*NE*+*UE*) (*Propagated L E'*) (*Propagated L E*)⟩
using *in-convert-lits-lD*[*OF - M-TM*, *of* ⟨*Propagated L E*⟩]
by (*auto simp*: *convert-lit.simps*)
then obtain *E''* **where** *LE''-M*: ⟨*Propagated L E''* ∈ *set M'*⟩
using *valid-trail-reduction-Propagated-inD2*[*OF red*, *of L E'*] **by** *auto*

consider
⟨*E'* = *0*⟩ **and** ⟨*E''* = *0*⟩ |
⟨*E'* > *0*⟩ **and** ⟨*E''* = *0*⟩ **and** ⟨*mset* (*N* ∝ *E'*) ∈# *NE* + *mset* ‘# *NE'* + *UE* + *mset* ‘# *UE'*⟩ |
⟨*E'* > *0*⟩ **and** ⟨*E''* > *0*⟩ **and** ⟨*E''* ∈# *dom-m N'*⟩ **and** ⟨*N* ∝ *E'* = *N'* ∝ *E''*⟩
using *tr-ge0 tr-new0 tr-still0 LE'-M LE''-M E-E'*
by (*cases* ⟨*E''*>*0*⟩; *cases* ⟨*E'* > *0*⟩) *auto*
then show ⟨*E* ∈# *clauses*
(*twl-clause-of* ‘# *init-clss-lf N* +
twl-clause-of ‘# *learned-clss-lf N'*) +
NE +
UE +
clauses (*twl-clause-of* ‘# *UE'*)⟩
apply *cases*
subgoal
using *E-E'*
by (*auto simp*: *mset-take-mset-drop-mset' convert-lit.simps*)
subgoal
using *E-E' init*
by (*auto simp*: *mset-take-mset-drop-mset' convert-lit.simps*)
subgoal
using *E-E' init*

by (*auto simp*: *mset-take-mset-drop-mset′ convert-lit.simps*)
done
qed
have ‹*cdcl-twl-restart*
(*TM*, *twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*, *None*,
NE, *UE*, {#}, *Q*)
(*TM*, *twl-clause-of* ‘# *init-clss-lf N′*, *twl-clause-of* ‘# *learned-clss-lf N′*, *None*,
NE + *clauses* (*twl-clause-of* ‘# *NE′*), *UE* + *clauses* (*twl-clause-of* ‘# *UE′*), {#},
Q)› (**is** ‹*cdcl-twl-restart ?A ?B*›)
apply (*rule cdcl-twl-restart.restart-clauses*)
subgoal
using *learned* **by** (*auto dest*: *image-mset-subseteq-mono*)
subgoal unfolding *init image-mset-union* **by** *auto*
subgoal using *NUE M-TM* **by** *auto*
subgoal by (*rule annot-in-clauses*)
done
moreover have ‹*?A* = *T*›
unfolding *T* **by** *simp*
moreover have ‹*?B* = *?T′*›
by (*auto simp*: *T mset-take-mset-drop-mset′*)
ultimately show *?case*
by *argo*
next
case (*backtrack-red K M2 M′′*) **note** *decomp* = *this*(*1*)
have [*simp*]: ‹*length M2* = *length M′*›
using *arg-cong*[*OF backtrack-red*(*2*), *of length*] **by** *simp*
have *M-TM*: ‹(*drop* (*length M* − *length M′*) *M*, *drop* (*length M* − *length M′*) *TM*) ∈
convert-lits-l N (*NE*+*UE*)›
using *M-TM* **unfolding** *convert-lits-l-def list-rel-def* **by** *auto*
have *red*: ‹*valid-trail-reduction* (*drop* (*length M* − *length M′*) *M*) *M′*›
using *red backtrack-red* **by** (*auto simp*: *valid-trail-reduction.simps*)
have *annot-in-clauses*: ‹∀ *L E*. *Propagated L E* ∈ *set* (*drop* (*length M* − *length M′*) *TM*) ⟶
E ∈# *clauses*
(*twl-clause-of* ‘# *init-clss-lf N* +
twl-clause-of ‘# *learned-clss-lf N′*) +
NE +
UE +
clauses (*twl-clause-of* ‘# *UE′*)›
proof (*intro allI impI conjI*)
fix *L E*
assume ‹*Propagated L E* ∈ *set* (*drop* (*length M* − *length M′*) *TM*)›
then obtain *E′* **where** *LE′-M*: ‹*Propagated L E′* ∈ *set* (*drop* (*length M* − *length M′*) *M*)› **and**
E-E′: ‹*convert-lit N* (*NE*+*UE*) (*Propagated L E′*) (*Propagated L E*)›
using *in-convert-lits-lD*[*OF - M-TM*, *of* ‹*Propagated L E*›]
by (*auto simp*: *convert-lit.simps*)
then have ‹*Propagated L E′* ∈ *set M2*›
using *decomp* **by** (*auto dest*!: *get-all-ann-decomposition-exists-prepend*)
then obtain *E′′* **where** *LE′′-M*: ‹*Propagated L E′′* ∈ *set M′*›
using *valid-trail-reduction-Propagated-inD2*[*OF red*, *of L E′*] *decomp*
by (*auto dest*!: *get-all-ann-decomposition-exists-prepend*)
consider
‹*E′* = *0*› **and** ‹*E′′* = *0*› |
‹*E′* > *0*› **and** ‹*E′′* = *0*› **and** ‹*mset* (*N* ∝ *E′*) ∈# *NE* + *mset* ‘# *NE′* + *UE* + *mset* ‘# *UE′*› |
‹*E′* > *0*› **and** ‹*E′′* > *0*› **and** ‹*E′′* ∈# *dom-m N′*› **and** ‹*N* ∝ *E′* = *N′* ∝ *E′′*›
using *tr-ge0 tr-new0 tr-still0 LE′-M LE′′-M E-E′ decomp*
by (*cases* ‹*E′′*>*0*›; *cases* ‹*E′* > *0*›)

(*auto 5 5 dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *convert-lit.simps*)
then show ‹$E \in\#$ *clauses*
(*twl-clause-of* ‘# *init-clss-lf* N +
twl-clause-of ‘# *learned-clss-lf* N') +
NE +
UE +
clauses (*twl-clause-of* ‘# UE')›
apply *cases*
subgoal
using *E-E*$'$
by (*auto simp*: *mset-take-mset-drop-mset*$'$ *convert-lit.simps*)
subgoal
using *E-E*$'$ *init*
by (*auto simp*: *mset-take-mset-drop-mset*$'$ *convert-lit.simps*)
subgoal
using *E-E*$'$ *init*
by (*auto simp*: *mset-take-mset-drop-mset*$'$ *convert-lit.simps*)
done
qed
have *lits-of-M2-M*$'$: ‹*lits-of-l M2* = *lits-of-l* M'›
using *arg-cong*[*OF backtrack-red*(*2*), *of set*] **by** (*auto simp*: *lits-of-def*)
have *lev-M2-M*$'$: ‹*get-level M2 L* = *get-level* M' L› **for** L
using *trail-renumber-get-level*[*OF backtrack-red*(*2*−*3*)] **by** (*auto simp*:)
have *drop-M-M2*: ‹*drop* (*length M* − *length* M') M = *M2*›
using *backtrack-red*(*1*) **by** *auto*
have H: ‹$L \in$ *lits-of-l* (*drop* (*length M* − *length* M') *TM*) ∧
get-level (*drop* (*length M* − *length* M') *TM*) L = *0*›
if ‹$L \in$ *lits-of-l M* ∧ *get-level M L* = *0*› **for** L
proof −
have ‹$L \in$ *lits-of-l M2* ∧ *get-level M2 L* = *0*›
using *decomp that n-d*
by (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
dest: *in-lits-of-l-defined-litD*
simp: *get-level-append-if get-level-cons-if split*: *if-splits*)
then show *?thesis*
using *M-TM*
by (*auto dest*!: *simp*: *drop-M-M2*)
qed

have
‹∃ *M2*. (*Decided K* # *drop* (*length M* − *length* M') *TM*, *M2*) ∈ *set* (*get-all-ann-decomposition*
TM)›
using *convert-lits-l-decomp-ex*[*OF decomp* ‹(*M*, *TM*) ∈ *convert-lits-l N* (*NE* + *UE*)›]
‹(*M*, *TM*) ∈ *convert-lits-l N* (*NE* + *UE*)›
by (*simp add*: *convert-lits-l-imp-same-length*)
then obtain *TM2* **where** *decomp-TM*:
‹(*Decided K* # *drop* (*length M* − *length* M') *TM*, *TM2*) ∈ *set* (*get-all-ann-decomposition TM*)›
by *blast*
have ‹*cdcl-twl-restart*
(*TM*, *twl-clause-of* ‘# *init-clss-lf N*, *twl-clause-of* ‘# *learned-clss-lf N*, *None*,
NE, *UE*, {#}, *Q*)
(*drop* (*length M* − *length* M') *TM*, *twl-clause-of* ‘# *init-clss-lf* N',
twl-clause-of ‘# *learned-clss-lf* N', *None*,
NE + *clauses* (*twl-clause-of* ‘# NE'), *UE* + *clauses* (*twl-clause-of* ‘# UE'), {#},
{#})› (**is** ‹*cdcl-twl-restart ?A ?B*›)

apply (*rule cdcl-twl-restart.restart-trail*)
apply (*rule decomp-TM*)
subgoal
using *learned* **by** (*auto dest*: *image-mset-subseteq-mono*)
subgoal unfolding *init image-mset-union* **by** *auto*
subgoal using *NUE M-TM H* **by** *fastforce*
subgoal by (*rule annot-in-clauses*)
done
moreover have ‹*?A* = *T*›
unfolding *T* **by** *auto*
moreover have ‹*?B* = *?T′*›
using *QQ′ decomp* **unfolding** *T* **by** (*auto simp*: *mset-take-mset-drop-mset′*)
ultimately show *?case*
by *argo*
qed
moreover {
have ‹(*M′*, *drop* (*length M* − *length M′*) *TM*) ∈ *convert-lits-l N′* (*NE* + *mset* ‘# *NE′* + (*UE* + *mset* ‘# *UE′*))›
proof (*rule convert-lits-lI*)
show ‹*length M′* = *length* (*drop* (*length M* − *length M′*) *TM*)›
using *M-TM red* **by** (*auto simp*: *valid-trail-reduction.simps T*
dest: *convert-lits-l-imp-same-length*
dest!: *arg-cong*[*of* ‹*map lit-of* -› - *length*] *get-all-ann-decomposition-exists-prepend*)
fix *i*
assume *i-M′*: ‹*i* < *length M′*›
then have *MM′-IM*: ‹*length M* − *length M′* + *i* < *length M*› ‹*length M* − *length M′* + *i* < *length TM*›
using *M-TM red* **by** (*auto simp*: *valid-trail-reduction.simps T*
dest: *convert-lits-l-imp-same-length*
dest!: *arg-cong*[*of* ‹*map lit-of* -› - *length*] *get-all-ann-decomposition-exists-prepend*)
then have ‹*convert-lit N* (*NE* + *UE*) (*drop* (*length M* − *length M′*) *M* ! *i*)
(*drop* (*length M* − *length M′*) *TM* ! *i*)›
using *M-TM list-all2-nthD*[*of* ‹*convert-lit N* (*NE* + *UE*)› *M TM* ‹*length M* − *length M′* + *i*›] *i-M′*
unfolding *convert-lits-l-def list-rel-def p2rel-def*
by *auto*
moreover
have ‹*lit-of* (*drop* (*length M* − *length M′*) *M*!*i*) = *lit-of* (*M′*!*i*)› **and**
‹*is-decided* (*drop* (*length M* − *length M′*) *M*!*i*) = *is-decided* (*M′*!*i*)›
using *red i-M′ MM′-IM*
by (*auto 5 5 simp*:*valid-trail-reduction-simps nth-append*
dest: *map-eq-nth-eq*[*of* - - - *i*]
dest!: *get-all-ann-decomposition-exists-prepend*)
moreover have ‹*M′*!*i* ∈ *set M′*›
using *i-M′* **by** *auto*
moreover have ‹*drop* (*length M* − *length M′*) *M*!*i* ∈ *set M*›
using *MM′-IM* **by** *auto*
ultimately show ‹*convert-lit N′* (*NE* + *mset* ‘# *NE′* + (*UE* + *mset* ‘# *UE′*)) (*M′* ! *i*)
(*drop* (*length M* − *length M′*) *TM* ! *i*)›
using *tr-new0 tr-still0 tr-ge0*
by (*cases* ‹*M′*!*i*›) (*fastforce simp*: *convert-lit.simps*)+
qed
then have ‹((*M′*, *N′*, *None*, *NE* + *mset* ‘# *NE′*, *UE* + *mset* ‘# *UE′*, {#}, *Q′*), *?T′*)
∈ *twl-st-l None*›
using *M-TM* **by** (*auto simp*: *twl-st-l-def T*)
}

ultimately show *?case*
by *fast*
qed
moreover have ‹*cdcl-twl-restart-l S S′* ⟹ *twl-list-invs S′*› **for** *S′*
by (*rule cdcl-twl-restart-l-list-invs*) (*use list-invs* **in** *fast*)+
moreover have ‹*cdcl-twl-restart-l S S′* ⟹ *clauses-to-update-l S′* = {#}› **for** *S′*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
ultimately show *?thesis*
by (*blast intro*!: *RES-refine*)
qed

definition (**in** −) *restart-abs-l-pre* :: ‹′*v twl-st-l* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-abs-l-pre S brk* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *twl-st-l None* ∧ *restart-prog-pre S′ brk*
∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#})›

context *twl-restart-ops*
begin

definition *restart-required-l* :: ′*v twl-st-l* ⇒ *nat* ⇒ *bool nres* **where**
‹*restart-required-l S n* = *SPEC* (λ*b*. *b* ⟶ *size* (*get-learned-clss-l S*) > *f n*)›

definition *restart-abs-l*
:: ′*v twl-st-l* ⇒ *nat* ⇒ *bool* ⇒ (′*v twl-st-l* × *nat*) *nres*
where
‹*restart-abs-l S n brk* = *do* {
  ASSERT(*restart-abs-l-pre S brk*);
  b ← *restart-required-l S n*;
  b2 ← *SPEC* (λ(- ::*bool*). *True*);
  if b ∧ *b2* ∧ ¬*brk then do* {
    T ← *SPEC*(λ*T*. *cdcl-twl-restart-l S T*);
    RETURN (*T*, *n* + *1*)
  }
  else
  if b ∧ ¬*brk then do* {
    T ← *SPEC*(λ*T*. *cdcl-twl-restart-l S T*);
    RETURN (*T*, *n* + *1*)
  }
  else
    RETURN (*S*, *n*)
}›

lemma (**in** −)[*twl-st-l*]:
‹(*S*, *S′*) ∈ *twl-st-l None* ⟹ *get-learned-clss S′* = *twl-clause-of* ‘# (*get-learned-clss-l S*)›
by (*auto simp*: *get-learned-clss-l-def twl-st-l-def*)

lemma *restart-required-l-restart-required*:
‹(*uncurry restart-required-l*, *uncurry restart-required*) ∈
{(*S*, *S′*). (*S*, *S′*) ∈ *twl-st-l None* ∧ *twl-list-invs S*} $\times_f$ *nat-rel* $\to_f$
⟨*bool-rel*⟩ *nres-rel*›
unfolding *restart-required-l-def restart-required-def uncurry-def*
by (*intro frefI nres-relI*) (*auto simp*: *twl-st-l-def get-learned-clss-l-def*)

lemma *restart-abs-l-restart-prog*:

‹(*uncurry2 restart-abs-l*, *uncurry2 restart-prog*) ∈
  {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}}
    $\times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$
  ⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}}
    $\times_f$ *nat-rel*⟩ *nres-rel*›
  unfolding *restart-abs-l-def restart-prog-def uncurry-def*
  apply (*intro frefI nres-relI*)
  apply (*refine-rcg*
    restart-required-l-restart-required[*THEN fref-to-Down-curry*]
    cdcl-twl-restart-l-cdcl-twl-restart)
  subgoal for *Snb Snb'*
   unfolding *restart-abs-l-pre-def*
   by (*rule exI*[*of* - ‹*fst* (*fst* (*Snb'*))›]) *simp*
  subgoal by *simp*
  subgoal by *auto* — If condition
  subgoal by *simp*
  subgoal by *simp*
  subgoal unfolding *restart-prog-pre-def* **by** *meson*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal unfolding *restart-prog-pre-def* **by** *meson*
  subgoal by *auto*
  subgoal by *auto*
  done

definition *cdcl-twl-stgy-restart-abs-l-inv* **where**
  ‹*cdcl-twl-stgy-restart-abs-l-inv* S_0 *brk T n* ≡
    (∃ S_0' *T'*.
      (S_0, S_0') ∈ *twl-st-l None* ∧
      (*T*, *T'*) ∈ *twl-st-l None* ∧
      cdcl-twl-stgy-restart-prog-inv S_0' *brk T' n* ∧
      clauses-to-update-l T = {#} ∧
      twl-list-invs T)›

definition *cdcl-twl-stgy-restart-abs-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**
  ‹*cdcl-twl-stgy-restart-abs-l* S_0 =
  do {
    (*brk*, *T*, -) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n}$
      (λ(*brk*, -). ¬*brk*)
      (λ(*brk*, *S*, *n*).
      do {
        T ← *unit-propagation-outer-loop-l S*;
        (*brk*, *T*) ← *cdcl-twl-o-prog-l T*;
        (*T*, *n*) ← *restart-abs-l T n brk*;
        RETURN (*brk*, *T*, *n*)
      })
      (*False*, S_0, *0*);
    RETURN T
  }›

lemma *cdcl-twl-stgy-restart-abs-l-cdcl-twl-stgy-restart-abs-l*:
  ‹(*cdcl-twl-stgy-restart-abs-l*, *cdcl-twl-stgy-restart-prog*) ∈
    {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧

clauses-to-update-l S $= \{\#\}\} \rightarrow_f$
⟨$\{(S, S')$. $(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S*$\}$⟩ *nres-rel*⟩
unfolding *cdcl-twl-stgy-restart-abs-l-def cdcl-twl-stgy-restart-prog-def uncurry-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** R = ⟨$\{((brk :: bool, S, n :: nat), (brk', S', n'))$.
$(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$ $brk = brk' \wedge n = n' \wedge$
clauses-to-update-l S $= \{\#\}\}$⟩]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry2*])
subgoal by *simp*
subgoal for *x y xa x' x1 x2 x1a x2a*
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def*
apply (*rule-tac x=y* **in** *exI*)
apply (*rule-tac x=*⟨*fst* (*snd x'*)⟩ **in** *exI*)
by *auto*
subgoal by *fast*
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
cdcl-twl-stgy-restart-abs-l-inv-def
apply (*simp only*: *prod.case*)
apply (*normalize-goal*)+
by (*simp add*: *twl-st-l twl-st*)
subgoal by (*auto simp*: *twl-st-l twl-st*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

end

We here start the refinement towards an executable version of the restarts. The idea of the restart is the following:

1. We backtrack to level 0. This simplifies further steps.
2. We first move all clause annotating a literal to *NE* or *UE*.
3. Then, we move remaining clauses that are watching the some literal at level 0.
4. Now we can safely deleting any remaining learned clauses.
5. Once all that is done, we have to recalculate the watch lists (and can on the way GC the set of clauses).

Handle true clauses from the trail

lemma *in-set-mset-eq-in*:
⟨$i \in$ *set A* $\Longrightarrow$ *mset A* $= B \Longrightarrow i \in\# B$⟩
by *fastforce*

Our transformation will be chains of a weaker version of restarts, that don't update the watch lists and only keep partial correctness of it.

lemma *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*:

assumes
ST: ‹*cdcl-twl-restart-l S T*› **and** TU: ‹*cdcl-twl-restart-l T U*› **and**
n-d: ‹*no-dup* (*get-trail-l S*)›
shows ‹*cdcl-twl-restart-l S U*›
using *assms*
proof −
obtain $M\ M'\ N\ N'\ NE'\ UE'\ NE\ UE\ Q\ Q'\ W'\ W$ **where**
S: ‹$S = (M, N, None, NE, UE, W, Q)$› **and**
T: ‹$T = (M', N', None, NE + mset\ `\#\ NE', UE + mset\ `\#\ UE', W', Q')$› **and**
tr-red: ‹*valid-trail-reduction* $M\ M'$› **and**
init: ‹*init-clss-lf* N = *init-clss-lf* $N' + NE'$› **and**
learned: ‹*learned-clss-lf* $N' + UE' \subseteq\#$ *learned-clss-lf* N› **and**
NUE: ‹$\forall E \in\# NE' + UE'.\ \exists L \in set\ E.\ L \in lits\text{-}of\text{-}l\ M \wedge get\text{-}level\ M\ L = 0$› **and**
ge0: ‹$\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow 0 < E \longrightarrow 0 < E' \longrightarrow$
$E \in\#\ dom\text{-}m\ N' \wedge N' \propto E = N \propto E'$› **and**
new0: ‹$\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow E = 0 \longrightarrow$
$E' \neq 0 \longrightarrow mset\ (N \propto E') \in\# NE + mset\ `\#\ NE' + UE + mset\ `\#\ UE'$› **and**
still0: ‹$\forall L\ E\ E'.\ Propagated\ L\ E \in set\ M' \longrightarrow Propagated\ L\ E' \in set\ M \longrightarrow$
$E' = 0 \longrightarrow E = 0$› **and**
dom0: ‹$0 \notin\#\ dom\text{-}m\ N'$› **and**
QQ': ‹*if length* M = *length* M' *then* $Q = Q'$ *else* $Q' = \{\#\}$› **and**
W: ‹$W = \{\#\}$›
using ST **unfolding** *cdcl-twl-restart-l.simps*
apply −
apply *normalize-goal*+
by *blast*
obtain $M''\ N''\ NE''\ UE''\ Q''\ W''$ **where**
U: ‹$U = (M'', N'', None, NE + mset\ `\#\ NE' + mset\ `\#\ NE'', UE + mset\ `\#\ UE' + mset\ `\#$
UE'', W'',
$Q'')$› **and**
tr-red′: ‹*valid-trail-reduction* $M'\ M''$› **and**
init′: ‹*init-clss-lf* N' = *init-clss-lf* $N'' + NE''$› **and**
learned′: ‹*learned-clss-lf* $N'' + UE'' \subseteq\#$ *learned-clss-lf* N'› **and**
NUE': ‹$\forall E \in\# NE'' + UE''$.
$\exists L \in set\ E$.
$L \in lits\text{-}of\text{-}l\ M' \wedge$
$get\text{-}level\ M'\ L = 0$› **and**
ge0′: ‹$\forall L\ E\ E'$.
$Propagated\ L\ E \in set\ M'' \longrightarrow$
$Propagated\ L\ E' \in set\ M' \longrightarrow$
$0 < E \longrightarrow$
$0 < E' \longrightarrow$
$E \in\#\ dom\text{-}m\ N'' \wedge N'' \propto E = N' \propto E'$› **and**
new0′: ‹$\forall L\ E\ E'$.
$Propagated\ L\ E \in set\ M'' \longrightarrow$
$Propagated\ L\ E' \in set\ M' \longrightarrow$
$E = 0 \longrightarrow$
$E' \neq 0 \longrightarrow$
$mset\ (N' \propto E')$
$\in\# NE + mset\ `\#\ NE' + mset\ `\#\ NE'' +$
$(UE + mset\ `\#\ UE') +$
$mset\ `\#\ UE''$› **and**
still0′: ‹$\forall L\ E\ E'$.
$Propagated\ L\ E \in set\ M'' \longrightarrow$
$Propagated\ L\ E' \in set\ M' \longrightarrow$
$E' = 0 \longrightarrow E = 0$› **and**

dom0': ‹*0* $\notin$# *dom-m N''*› **and**
Q'Q'': ‹*if length M' = length M'' then Q' = Q'' else Q'' =* {#}› **and**
W': ‹*W'* = {#}› **and**
W'': ‹*W''* = {#}›
using *TU* **unfolding** *cdcl-twl-restart-l.simps T* **apply** −
apply *normalize-goal*+
by *blast*
have *U'*: ‹*U* = (*M''*, *N''*, *None*, *NE* + *mset* ‘# (*NE'* + *NE''*), *UE* + *mset* ‘# (*UE'* + *UE''*), *W''*, *Q''*)›
unfolding *U* **by** *simp*
show *?thesis*
unfolding *S U' W W' W''*
apply (*rule cdcl-twl-restart-l.restart-trail*)
subgoal using *valid-trail-reduction-trans*[*OF tr-red tr-red'*] **.**
subgoal using *init init'* **by** *auto*
subgoal using *learned learned' subset-mset.dual-order.trans* **by** *fastforce*
subgoal using *NUE NUE' valid-trail-reduction-level0-iff*[*OF tr-red*] *n-d* **unfolding** *S* **by** *auto*
subgoal using *ge0 ge0' tr-red' init learned NUE ge0 still0'*
apply (*auto dest*: *valid-trail-reduction-Propagated-inD*)
apply (*blast dest*: *valid-trail-reduction-Propagated-inD*)+
apply (*metis neq0-conv still0' valid-trail-reduction-Propagated-inD*)+
done
subgoal using *new0 new0' tr-red' init learned NUE ge0*
apply (*auto dest*: *valid-trail-reduction-Propagated-inD*)
by (*smt neq0-conv valid-trail-reduction-Propagated-inD*)
subgoal using *still0 still0' tr-red'* **by** (*fastforce dest*: *valid-trail-reduction-Propagated-inD*)
subgoal using *dom0'* **.**
subgoal using *QQ' Q'Q'' valid-trail-reduction-length-leD*[*OF tr-red*] *valid-trail-reduction-length-leD*[*OF tr-red'*]
by (*auto split*: *if-splits*)
done
qed

lemma *rtranclp-cdcl-twl-restart-l-no-dup*:
assumes
ST: ‹*cdcl-twl-restart-l*** *S T*› **and**
n-d: ‹*no-dup* (*get-trail-l S*)›
shows ‹*no-dup* (*get-trail-l T*)›
using *assms*
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal
by (*auto simp*: *cdcl-twl-restart-l.simps valid-trail-reduction-simps dest*: *map-lit-of-eq-no-dupD dest*!: *no-dup-appendD get-all-ann-decomposition-exists-prepend*)
done

lemma *tranclp-cdcl-twl-restart-l-cdcl-is-cdcl-twl-restart-l*:
assumes
ST: ‹*cdcl-twl-restart-l*$^{++}$ *S T*› **and**
n-d: ‹*no-dup* (*get-trail-l S*)›
shows ‹*cdcl-twl-restart-l S T*›
using *assms*
apply (*induction rule*: *tranclp-induct*)
subgoal by *auto*
subgoal
using *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*

rtranclp-cdcl-twl-restart-l-no-dup **by** *blast*
done

lemma *valid-trail-reduction-refl*: ‹*valid-trail-reduction a a*›
by (*auto simp*: *valid-trail-reduction.simps*)

Auxilary definition This definition states that the domain of the clauses is reduced, but the remaining clauses are not changed.

definition *reduce-dom-clauses* **where**
‹*reduce-dom-clauses N N′* $\longleftrightarrow$
($\forall$ *C*. *C* $\in\#$ *dom-m N′* $\longrightarrow$ *C* $\in\#$ *dom-m N* $\wedge$ *fmlookup N C* = *fmlookup N′ C*)›

lemma *reduce-dom-clauses-fdrop*[*simp*]: ‹*reduce-dom-clauses N* (*fmdrop C N*)›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *reduce-dom-clauses-def dest*: *in-diffD multi-member-split*
distinct-mem-diff-mset)

lemma *reduce-dom-clauses-refl*[*simp*]: ‹*reduce-dom-clauses N N*›
by (*auto simp*: *reduce-dom-clauses-def*)

lemma *reduce-dom-clauses-trans*:
‹*reduce-dom-clauses N N′* $\Longrightarrow$ *reduce-dom-clauses N′ N′a* $\Longrightarrow$ *reduce-dom-clauses N N′a*›
by (*auto simp*: *reduce-dom-clauses-def*)

definition *valid-trail-reduction-eq* **where**
‹*valid-trail-reduction-eq M M′* $\longleftrightarrow$ *valid-trail-reduction M M′* $\wedge$ *length M* = *length M′*›

lemma *valid-trail-reduction-eq-alt-def*:
‹*valid-trail-reduction-eq M M′* $\longleftrightarrow$ *map lit-of M* = *map lit-of M′* $\wedge$
map is-decided M = *map is-decided M′*›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*
dest!: *get-all-ann-decomposition-exists-prepend*
dest: *map-eq-imp-length-eq trail-renumber-get-level*)

lemma *valid-trail-reduction-change-annot*:
‹*valid-trail-reduction* (*M* @ *Propagated L C* # *M′*)
(*M* @ *Propagated L 0* # *M′*)›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*)

lemma *valid-trail-reduction-eq-change-annot*:
‹*valid-trail-reduction-eq* (*M* @ *Propagated L C* # *M′*)
(*M* @ *Propagated L 0* # *M′*)›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*)

lemma *valid-trail-reduction-eq-refl*: ‹*valid-trail-reduction-eq M M*›
by (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction-refl*)

lemma *valid-trail-reduction-eq-get-level*:
‹*valid-trail-reduction-eq M M′* $\Longrightarrow$ *get-level M* = *get-level M′*›
by (*intro ext*)
(*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*
dest!: *get-all-ann-decomposition-exists-prepend*
dest: *map-eq-imp-length-eq trail-renumber-get-level*)

lemma *valid-trail-reduction-eq-lits-of-l*:

‹*valid-trail-reduction-eq M M′ ⟹ lits-of-l M = lits-of-l M′*›
  apply (*auto simp*: *valid-trail-reduction-eq-def valid-trail-reduction.simps*
    dest!: *get-all-ann-decomposition-exists-prepend*
    dest: *map-eq-imp-length-eq trail-renumber-get-level*)
  apply (*metis image-set lits-of-def*)+
  done

lemma *valid-trail-reduction-eq-trans*:
  ‹*valid-trail-reduction-eq M M′ ⟹ valid-trail-reduction-eq M′ M″ ⟹*
   valid-trail-reduction-eq M M″›
  unfolding *valid-trail-reduction-eq-alt-def*
  by *auto*

definition *no-dup-reasons-invs-wl* **where**
  ‹*no-dup-reasons-invs-wl S ⟷*
   (*distinct-mset* (*mark-of* ‘# *filter-mset* (λ*C*. *is-proped C* ∧ *mark-of C* > *0*) (*mset* (*get-trail-l S*))))›

inductive *different-annot-all-killed* **where**
propa-changed:
  ‹*different-annot-all-killed N NUE* (*Propagated L C*) (*Propagated L C′*)›
   if ‹*C* ≠ *C′*› **and** ‹*C′* = *0*› **and**
    ‹*C* ∈# *dom-m N* ⟹ *mset* (*N*∝*C*) ∈# *NUE*› |
propa-not-changed:
  ‹*different-annot-all-killed N NUE* (*Propagated L C*) (*Propagated L C*)› |
decided-not-changed:
  ‹*different-annot-all-killed N NUE* (*Decided L*) (*Decided L*)›

lemma *different-annot-all-killed-refl*[*iff*]:
  ‹*different-annot-all-killed N NUE z z ⟷ is-proped z* ∨ *is-decided z*›
  by (*cases z*) (*auto simp*: *different-annot-all-killed.simps*)

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

lemma *different-annots-all-killed-refl*:
  ‹*different-annots-all-killed N NUE M M*›
  by (*auto intro*!: *list.rel-refl-strong simp*: *count-decided-0-iff is-decided-no-proped-iff*)

Refinement towards code Once of the first thing we do, is removing clauses we know to be true. We do in two ways:

- along the trail (at level 0); this makes sure that annotations are kept;
- then along the watch list.

This is (obviously) not complete but is faster by avoiding iterating over all clauses. Here are the rules we want to apply for our very limited inprocessing:

inductive *remove-one-annot-true-clause* :: ‹′*v twl-st-l* ⇒ ′*v twl-st-l* ⇒ *bool*› **where**
remove-irred-trail:
  ‹*remove-one-annot-true-clause* (*M* @ *Propagated L C* # *M′*, *N*, *D*, *NE*, *UE*, *W*, *Q*)
    (*M* @ *Propagated L 0* # *M′*, *fmdrop C N*, *D*, *add-mset* (*mset* (*N*∝*C*)) *NE*, *UE*, *W*, *Q*)›
if
  ‹*get-level* (*M* @ *Propagated L C* # *M′*) *L* = *0*› **and**
  ‹*C* > *0*› **and**

⟨$C \in\#$ *dom-m N*⟩ **and**
⟨*irred N C*⟩ |
remove-red-trail:
⟨*remove-one-annot-true-clause* (M @ *Propagated L C* # M', N, D, NE, UE, W, Q)
(M @ *Propagated L 0* # M', *fmdrop C N*, D, NE, *add-mset* (*mset* ($N \propto C$)) UE, W, Q)⟩
if
⟨*get-level* (M @ *Propagated L C* # M') $L = 0$⟩ **and**
⟨$C > 0$⟩ **and**
⟨$C \in\#$ *dom-m N*⟩ **and**
⟨$\neg$*irred N C*⟩ |
remove-irred:
⟨*remove-one-annot-true-clause* (M, N, D, NE, UE, W, Q)
(M, *fmdrop C N*, D, *add-mset* (*mset* ($N \propto C$))NE, UE, W, Q)⟩
if
⟨$L \in$ *lits-of-l M*⟩ **and**
⟨*get-level M L* $= 0$⟩ **and**
⟨$C \in\#$ *dom-m N*⟩ **and**
⟨$L \in$ *set* ($N \propto C$)⟩ **and**
⟨*irred N C*⟩ **and**
⟨$\forall L$. *Propagated L C* $\notin$ *set M*⟩ |
delete:
⟨*remove-one-annot-true-clause* (M, N, D, NE, UE, W, Q)
(M, *fmdrop C N*, D, NE, UE, W, Q)⟩
if
⟨$C \in\#$ *dom-m N*⟩ **and**
⟨$\neg$*irred N C*⟩ **and**
⟨$\forall L$. *Propagated L C* $\notin$ *set M*⟩

Remarks:

1. $\forall L$. *Propagated L C* $\notin$ *set M* is overkill. However, I am currently unsure how I want to handle it (either as *Propagated* ($N \propto C$! 0) $C \notin$ *set M* or as "the trail contains only zero anyway").

lemma *Ex-ex-eq-Ex*: ⟨($\exists NE'$. ($\exists b$. $NE' = \{\#b\#\} \wedge P\ b\ NE'$) $\wedge Q\ NE'$) $\longleftrightarrow$
($\exists b$. $P\ b\ \{\#b\#\} \wedge Q\ \{\#b\#\}$)⟩
by *auto*

lemma *in-set-definedD*:
⟨*Propagated* L' $C \in$ *set* $M' \Longrightarrow$ *defined-lit* M' L'⟩
⟨*Decided* $L' \in$ *set* $M' \Longrightarrow$ *defined-lit* M' L'⟩
by (*auto simp*: *defined-lit-def*)

lemma (**in** *conflict-driven-clause-learning*$_W$) *trail-no-annotation-reuse*:
assumes
struct-invs: ⟨*cdcl*$_W$*-all-struct-inv S*⟩ **and**
LC: ⟨*Propagated L C* $\in$ *set* (*trail S*)⟩ **and**
LC': ⟨*Propagated* L' $C \in$ *set* (*trail S*)⟩
shows $L = L'$
proof –
have
confl: ⟨*cdcl*$_W$*-conflicting S*⟩ **and**
n-d: ⟨*no-dup* (*trail S*)⟩
using *struct-invs* **unfolding** *cdcl*$_W$*-all-struct-inv-def* *cdcl*$_W$*-M-level-inv-def*
by *fast+*

find-theorems - @ -#- = - @ - # -
have *H*: ⟨*L* = *L'*⟩ **if** ⟨*trail S* = *ysa* @ *Propagated L' C* # *c21* @ *Propagated L C* # *zs*⟩
for *ysa xsa c21 zs L L'*
proof −
have *1*: ⟨*c21* @ *Propagated L C* # *zs* ⊨*as CNot* (*remove1-mset L' C*) ∧ *L'* ∈# *C*⟩
using *confl* **unfolding** *cdcl*$_W$*-conflicting-def that*
by (*auto*)
have *that'*: ⟨*trail S* = (*ysa* @ *Propagated L' C* # *c21*) @ *Propagated L C* # *zs*⟩
unfolding *that* **by** *auto*
have *2*: ⟨*zs* ⊨*as CNot* (*remove1-mset L C*) ∧ *L* ∈# *C*⟩
using *confl* **unfolding** *cdcl*$_W$*-conflicting-def that'*
by *blast*
show ⟨*L* = *L'*⟩
using *1 2 n-d* **unfolding** *that*
by (*auto dest*!: *multi-member-split*
simp: *true-annots-true-cls-def-iff-negation-in-model add-mset-eq-add-mset*
Decided-Propagated-in-iff-in-lits-of-l)
qed
show *?thesis*
using *H*[*of* - *L* - *L'*] *H*[*of* - *L'* - *L*]
using *split-list*[*OF LC*] *split-list*[*OF LC'*]
by (*force elim*!: *list-match-lel-lel*)
qed

lemma *remove-one-annot-true-clause-cdcl-twl-restart-l*:
assumes
rem: ⟨*remove-one-annot-true-clause S T*⟩ **and**
lst-invs: ⟨*twl-list-invs S*⟩ **and**
SS': ⟨(*S*, *S'*) ∈ *twl-st-l None*⟩ **and**
struct-invs: ⟨*twl-struct-invs S'*⟩ **and**
confl: ⟨*get-conflict-l S* = *None*⟩ **and**
upd: ⟨*clauses-to-update-l S* = {#}⟩ **and**
n-d: ⟨*no-dup* (*get-trail-l S*)⟩
shows ⟨*cdcl-twl-restart-l S T*⟩
using *assms*
proof −
have *dist-N*: ⟨*distinct-mset* (*dom-m* (*get-clauses-l S*))⟩
by (*rule distinct-mset-dom*)
then have *C-notin-rem*: ⟨*C* ∉# *remove1-mset C* (*dom-m* (*get-clauses-l S*))⟩ **for** *C*
by (*simp add*: *distinct-mset-remove1-All*)
have
⟨∀ *C*∈#*clauses-to-update-l S*. *C* ∈# *dom-m* (*get-clauses-l S*)⟩ **and**
dom0: ⟨*0* ∉# *dom-m* (*get-clauses-l S*)⟩ **and**
annot: ⟨⋀*L C*. *Propagated L C* ∈ *set* (*get-trail-l S*) ⟹
0 < *C* ⟹
(*C* ∈# *dom-m* (*get-clauses-l S*) ∧
L ∈ *set* (*watched-l* (*get-clauses-l S* ∝ *C*)) ∧
(*length* (*get-clauses-l S* ∝ *C*) > *2* ⟶ *L* = *get-clauses-l S* ∝ *C* ! *0*))⟩ **and**
⟨*distinct-mset* (*clauses-to-update-l S*)⟩
using *lst-invs* **unfolding** *twl-list-invs-def* **apply** −
by *fast*+

have *struct-S'*: ⟨*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*state*$_W$*-of S'*)⟩
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
show *?thesis*
using *rem*

proof (*cases rule*: *remove-one-annot-true-clause.cases*)
case (*remove-irred-trail M L C M' N D NE UE W Q*) **note** *S* = *this*(*1*) **and** *T* = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *C0* = *this*(*4*) **and** *C-dom* = *this*(*5*) **and** *irred* = *this*(*6*)
have *D*: ‹*D = None*› **and** *W*: ‹*W* = {#}›
using *confl upd* **unfolding** *S* **by** *auto*
have *NE*: ‹*add-mset* (*mset* (*N* $\propto$ *C*)) *NE* = *NE* + *mset* '# {#*N*$\propto$*C*#}›
by *simp*
have *UE*: ‹*UE* = *UE* + *mset* '# {#}›
by *simp*
have *new-NUE*: ‹$\forall$ *E*$\in$#{#*N* $\propto$ *C*#} + {#}.
$\exists$ *La*$\in$*set E*.
La $\in$ *lits-of-l* (*M* @ *Propagated L C* # *M'*) $\wedge$
get-level (*M* @ *Propagated L C* # *M'*) *La* = *0*›
apply (*intro ballI impI*)
apply (*rule-tac x*=*L* **in** *bexI*)
using *lev-L annot*[*of L -*] *C0* **by** (*auto simp*: *S dest*: *in-set-takeD*[*of - 2*])
have [*simp*]: ‹*Propagated L E'* $\notin$ *set M'*› ‹*Propagated L E'* $\notin$ *set M*› **for** *E'*
using *n-d lst-invs*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E'*› *M*]
split-list[*of* ‹*Propagated L E'*› *M'*])
have [*simp*]: ‹*Propagated L' C* $\notin$ *set M'*› ‹*Propagated L' C* $\notin$ *set M*› **for** *L'*
using *SS' n-d C0 struct-S'*
$cdcl_W$*-restart-mset.trail-no-annotation-reuse*[*of* ‹$state_W$*-of S'*› *L* ‹(*mset* (*N* $\propto$ *C*))› *L'*]
apply (*auto simp*: *S twl-st-l-def convert-lits-l-imp-same-length trail.simps*
)
apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff convert-lits-l-def*
p2rel-def)
apply (*case-tac y*)
apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff defined-lit-convert-lits-l*
simp flip: *p2rel-def convert-lits-l-def dest*: *in-set-definedD*(*1*)[*of - - M'*])
apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff convert-lits-l-def*
p2rel-def convert-lit.simps
dest!: *split-list*[*of* ‹*Propagated L' C*› *M'*]
split-list[*of* ‹*Propagated L' C*› *M*])
done
have *propa-MM*: ‹*Propagated L E* $\in$ *set M* $\Longrightarrow$ *Propagated L E'* $\in$ *set M* $\Longrightarrow$ *E*=*E'*› **for** *L E E'*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E'*› *M*]
elim!: *list-match-lel-lel*)
have *propa-M'M'*: ‹*Propagated L E* $\in$ *set M'* $\Longrightarrow$ *Propagated L E'* $\in$ *set M'* $\Longrightarrow$ *E*=*E'*› **for** *L E E'*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M'*]
split-list[*of* ‹*Propagated L E'*› *M'*]
elim!: *list-match-lel-lel*)
have *propa-MM'*: ‹*Propagated L E* $\in$ *set M* $\Longrightarrow$ *Propagated L E'* $\in$ *set M'* $\Longrightarrow$ *False*› **for** *L E E'*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E'*› *M'*]
elim!: *list-match-lel-lel*)
have *propa-M'-nC-dom*: ‹*Propagated La E* $\in$ *set M'* $\Longrightarrow$ *E* $\neq$ *C* $\wedge$ (*E* > *0* $\longrightarrow$ *E* $\in$# *dom-m N*)›
for *La E*

using *annot*[*of La E*] **unfolding** *S* **by** *auto*
have *propa-M-nC-dom*: ‹*Propagated La E* ∈ *set M* ⟹ *E* ≠ *C* ∧ (*E* > *0* ⟶ *E* ∈# *dom-m N*)›
for *La E*
using *annot*[*of La E*] **unfolding** *S* **by** *auto*
show *?thesis*
unfolding *S T D W NE*
apply (*subst* (*2*) *UE*)
apply (*rule cdcl-twl-restart-l.intros*)
subgoal by (*auto simp*: *valid-trail-reduction-change-annot*)
subgoal using *C-dom irred* **by** *auto*
subgoal using *irred* **by** *auto*
subgoal using *new-NUE* **.**
subgoal
apply (*intro conjI allI impI*)
subgoal for *La E E′*
using *C-notin-rem propa-MM*[*of La E E′*] *propa-MM′*[*of La E E′*] *propa-M′-nC-dom*[*of La E*] *propa-M-nC-dom*[*of La E*]
unfolding *S* **by** *auto*
subgoal for *La E E′*
using *C-notin-rem propa-MM*[*of La E E′*] *propa-MM′*[*of La E E′*] *propa-M′-nC-dom*[*of La E*] *propa-M-nC-dom*[*of La E*] *propa-MM′*[*of La E′ E*] *propa-M′M′*[*of La E′ E*]
unfolding *S* **by** *auto*
done
subgoal
apply (*intro allI impI*)
subgoal for *La E E′*
using *C-notin-rem propa-MM*[*of La E E′*] *propa-MM′*[*of La E E′*] *propa-M′-nC-dom*[*of La E*] *propa-M-nC-dom*[*of La E*] *propa-MM′*[*of La E′ E*] *propa-M′M′*[*of La E′ E*]
by *auto*
done
subgoal
apply (*intro allI impI*)
subgoal for *La E E′*
using *C-notin-rem propa-MM*[*of La E E′*] *propa-MM′*[*of La E E′*] *propa-M′-nC-dom*[*of La E*] *propa-M-nC-dom*[*of La E*] *propa-MM′*[*of La E′ E*] *propa-M′M′*[*of La E′ E*]
by *auto*
done
subgoal using *dom0* **unfolding** *S* **by** (*auto dest*: *in-diffD*)
subgoal by *auto*
done
next
case (*remove-red-trail M L C M′ N D NE UE W Q*) **note** *S* =*this*(*1*) **and** *T* = *this*(*2*) **and** *lev-L* = *this*(*3*) **and** *C0* = *this*(*4*) **and** *C-dom* = *this*(*5*) **and** *irred* = *this*(*6*)
have *D*: ‹*D* = *None*› **and** *W*: ‹*W* = {#}›
using *confl upd* **unfolding** *S* **by** *auto*
have *UE*: ‹*add-mset* (*mset* (*N* ∝ *C*)) *UE* = *UE* + *mset* ‘# {#*N*∝*C*#}›
by *simp*
have *NE*: ‹*NE* = *NE* + *mset* ‘# {#}›
by *simp*
have *new-NUE*: ‹∀ *E*∈#{#} + {#*N* ∝ *C*#}.
∃ *La*∈*set E*.
La ∈ *lits-of-l* (*M* @ *Propagated L C* # *M′*) ∧
get-level (*M* @ *Propagated L C* # *M′*) *La* = *0*›
apply (*intro ballI impI*)
apply (*rule-tac x*=*L* **in** *bexI*)
using *lev-L annot*[*of L -*] *C0* **by** (*auto simp*: *S dest*: *in-set-takeD*[*of - 2*])

have [*simp*]: ‹*Propagated L E′ ∉ set M′*› ‹*Propagated L E′ ∉ set M*› **for** *E′*
  using *n-d lst-invs*
  by (*auto simp*: *S twl-list-invs-def*
    dest!: *split-list*[*of* ‹*Propagated L E′*› *M*]
      split-list[*of* ‹*Propagated L E′*› *M′*])
have [*simp*]: ‹*Propagated L′ C ∉ set M′*› ‹*Propagated L′ C ∉ set M*› **for** *L′*
  using *SS′ n-d C0 struct-S′*
  $cdcl_W$*-restart-mset.trail-no-annotation-reuse*[*of* ‹$state_W$*-of S′*› *L* ‹(*mset* (*N* ∝ *C*))› *L′*]
  apply (*auto simp*: *S twl-st-l-def convert-lits-l-imp-same-length trail.simps*
)
  apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff convert-lits-l-def*
    p2rel-def)
  apply (*case-tac y*)
  apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff defined-lit-convert-lits-l*
    simp flip: *p2rel-def convert-lits-l-def dest*: *in-set-definedD*(*1*)[*of - - M′*])
  apply (*auto simp*: *list-rel-append1 list-rel-split-right-iff convert-lits-l-def*
    p2rel-def convert-lit.simps
    dest!: *split-list*[*of* ‹*Propagated L′ C*› *M′*]
      split-list[*of* ‹*Propagated L′ C*› *M*])
  done
have *propa-MM*: ‹*Propagated L E ∈ set M ⟹ Propagated L E′ ∈ set M ⟹ E=E′*› **for** *L E E′*
  using *n-d*
  by (*auto simp*: *S twl-list-invs-def*
    dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
      split-list[*of* ‹*Propagated L E′*› *M*]
      elim!: *list-match-lel-lel*)
have *propa-M′M′*: ‹*Propagated L E ∈ set M′ ⟹ Propagated L E′ ∈ set M′ ⟹ E=E′*› **for** *L E E′*
  using *n-d*
  by (*auto simp*: *S twl-list-invs-def*
    dest!: *split-list*[*of* ‹*Propagated L E*› *M′*]
      split-list[*of* ‹*Propagated L E′*› *M′*]
      elim!: *list-match-lel-lel*)
have *propa-MM′*: ‹*Propagated L E ∈ set M ⟹ Propagated L E′ ∈ set M′ ⟹ False*› **for** *L E E′*
  using *n-d*
  by (*auto simp*: *S twl-list-invs-def*
    dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
      split-list[*of* ‹*Propagated L E′*› *M′*]
      elim!: *list-match-lel-lel*)
have *propa-M′-nC-dom*: ‹*Propagated La E ∈ set M′ ⟹ E ≠ C ∧ (E > 0 ⟶ E ∈# dom-m N)*›
for *La E*
  using *annot*[*of La E*] **unfolding** *S* **by** *auto*
have *propa-M-nC-dom*: ‹*Propagated La E ∈ set M ⟹ E ≠ C ∧ (E > 0 ⟶ E ∈# dom-m N)*›
for *La E*
  using *annot*[*of La E*] **unfolding** *S* **by** *auto*
show *?thesis*
  unfolding *S T D W UE*
  apply (*subst* (*2*) *NE*)
  apply (*rule cdcl-twl-restart-l.intros*)
  subgoal by (*auto simp*: *valid-trail-reduction-change-annot*)
  subgoal using *C-dom irred* **by** *auto*
  subgoal using *C-dom irred* **by** *auto*
  subgoal using *new-NUE* **.**
  subgoal
    apply (*intro conjI allI impI*)
    subgoal for *La E E′*
      using *C-notin-rem propa-MM*[*of La E E′*] *propa-MM′*[*of La E E′*] *propa-M′-nC-dom*[*of La E*]

propa-M-nC-dom[*of La E*]
unfolding *S* **by** *auto*
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*] *propa-MM'*[*of La E' E*] *propa-M'M'*[*of La E' E*]
unfolding *S* **by** *auto*
done
subgoal
apply (*intro allI impI*)
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*] *propa-MM'*[*of La E' E*] *propa-M'M'*[*of La E' E*]
by *auto*
done
subgoal
apply (*intro allI impI*)
subgoal for *La E E'*
using *C-notin-rem propa-MM*[*of La E E'*] *propa-MM'*[*of La E E'*] *propa-M'-nC-dom*[*of La E*]
propa-M-nC-dom[*of La E*] *propa-MM'*[*of La E' E*] *propa-M'M'*[*of La E' E*]
by *auto*
done
subgoal using *dom0* **unfolding** *S* **by** (*auto dest*: *in-diffD*)
subgoal by *auto*
done
next
case (*remove-irred L M C N D NE UE W Q*) **note** *S* =*this*(*1*) **and** *T* = *this*(*2*) **and**
L-M = *this*(*3*) **and** *lev-L* = *this*(*4*) **and** *C-dom* = *this*(*5*) **and** *watched-L* = *this*(*6*) **and**
irred = *this*(*7*) **and** *L-notin-M* = *this*(*8*)
have *NE*: ‹*add-mset* (*mset* (*N* ∝ *C*)) *NE* = *NE* + *mset* ‘# {#*N*∝*C*#}›
by *simp*
have *UE*: ‹*UE* = *UE* + *mset* ‘# {#}›
by *simp*
have *D*: ‹*D* = *None*› **and** *W*: ‹*W* = {#}›
using *confl upd* **unfolding** *S* **by** *auto*
have *new-NUE*: ‹∀ *E*∈#{#*N* ∝ *C*#} + {#}.
∃ *La*∈*set E*.
La ∈ *lits-of-l M* ∧
get-level M La = *0*›
apply (*intro ballI impI*)
apply (*rule-tac x*=*L* **in** *bexI*)
using *lev-L annot*[*of L -*] *L-M watched-L* **by** (*auto simp*: *S dest*: *in-set-takeD*[*of - 2*])
have *C0*: ‹*C* > *0*›
using *dom0 C-dom* **unfolding** *S* **by** (*auto dest*!: *multi-member-split*)
have [*simp*]: ‹*Propagated La C* ∉ *set M*› **for** *La*
using *annot*[*of La C*] *dom0 n-d L-notin-M C0* **unfolding** *S*
by *auto*
have *propa-MM*: ‹*Propagated L E* ∈ *set M* ⟹ *Propagated L E'* ∈ *set M* ⟹ *E*=*E'*› **for** *L E E'*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E'*› *M*]
elim!: *list-match-lel-lel*)
show *?thesis*
unfolding *S T D W NE*
apply (*subst* (*2*) *UE*)
apply (*rule cdcl-twl-restart-l.intros*)

subgoal by (*auto simp*: *valid-trail-reduction-refl*)
subgoal using *C-dom irred* **by** *auto*
subgoal using *C-dom irred* **by** *auto*
subgoal using *new-NUE* **.**
subgoal
using *n-d L-notin-M C-notin-rem annot propa-MM* **unfolding** *S* **by** *force*
subgoal
using *propa-MM* **by** *auto*
subgoal
using *propa-MM* **by** *auto*
subgoal using *dom0 C-dom* **unfolding** *S* **by** (*auto dest*: *in-diffD*)
subgoal by *auto*
done
next
case (*delete C N M D NE UE W Q*) **note** *S* = *this*(*1*) **and** *T* = *this*(*2*) **and** *C-dom* = *this*(*3*) **and** *irred* = *this*(*4*) **and** *L-notin-M* = *this*(*5*)
have *D*: ‹*D* = *None*› **and** *W*: ‹*W* = {#}›
using *confl upd* **unfolding** *S* **by** *auto*
have *UE*: ‹*UE* = *UE* + *mset* '# {#}›
by *simp*
have *NE*: ‹*NE* = *NE* + *mset* '# {#}›
by *simp*
have *propa-MM*: ‹*Propagated L E* ∈ *set M* ⟹ *Propagated L E'* ∈ *set M* ⟹ *E*=*E'*› **for** *L E E'*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E'*› *M*]
elim!: *list-match-lel-lel*)
show *?thesis*
unfolding *S T D W*
apply (*subst* (*2*) *NE*)
apply (*subst* (*2*) *UE*)
apply (*rule cdcl-twl-restart-l.intros*)
subgoal by (*auto simp*: *valid-trail-reduction-refl*)
subgoal using *C-dom irred* **by** *auto*
subgoal using *C-dom irred* **by** *auto*
subgoal by *simp*
subgoal
apply (*intro conjI impI allI*)
subgoal for *L E E'*
using *n-d L-notin-M C-notin-rem annot propa-MM*[*of L E E'*] **unfolding** *S*
by (*metis dom-m-fmdrop get-clauses-l.simps get-trail-l.simps in-remove1-msetI*)
subgoal for *L E E'*
using *n-d L-notin-M C-notin-rem annot propa-MM*[*of L E E'*] **unfolding** *S*
by *auto*
done
subgoal
using *propa-MM* **by** *auto*
subgoal
using *propa-MM* **by** *auto*
subgoal using *dom0 C-dom* **unfolding** *S* **by** (*auto dest*: *in-diffD*)
subgoal by *auto*
done
qed
qed

lemma *is-annot-iff-annotates-first*:
  assumes
    ST: ‹$(S,\ T) \in$ *twl-st-l None*› **and**
    list-invs: ‹*twl-list-invs S*› **and**
    struct-invs: ‹*twl-struct-invs T*› **and**
    C0: ‹$C > 0$›
  shows
    ‹$(\exists L.$ *Propagated L C* $\in$ *set* (*get-trail-l S*)) $\longleftrightarrow$
      ((*length* (*get-clauses-l S* $\propto$ *C*) $> 2 \longrightarrow$
        Propagated (*get-clauses-l S* $\propto$ *C* ! *0*) *C* $\in$ *set* (*get-trail-l S*)) $\wedge$
       ((*length* (*get-clauses-l S* $\propto$ *C*) $\leq 2 \longrightarrow$
    Propagated (*get-clauses-l S* $\propto$ *C* ! *0*) *C* $\in$ *set* (*get-trail-l S*) $\vee$
    Propagated (*get-clauses-l S* $\propto$ *C* ! *1*) *C* $\in$ *set* (*get-trail-l S*))))›
    (**is** ‹*?A* $\longleftrightarrow$ *?B*›)
proof (*rule iffI*)
  assume *?B*
  then show *?A* **by** *auto*
next
  assume *?A*
  then obtain *L* **where**
    LC: ‹*Propagated L C* $\in$ *set* (*get-trail-l S*)›
    by *blast*
  then have
    C: ‹$C \in\#$ *dom-m* (*get-clauses-l S*)› **and**
    L-w: ‹$L \in$ *set* (*watched-l* (*get-clauses-l S* $\propto$ *C*))› **and**
    L: ‹*length* (*get-clauses-l S* $\propto$ *C*) $> 2 \Longrightarrow L =$ *get-clauses-l S* $\propto$ *C* ! *0*›
    using *list-invs C0* **unfolding** *twl-list-invs-def* **by** *blast+*
  have ‹*twl-st-inv T*›
    using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
  then have *le2*: ‹*length* (*get-clauses-l S* $\propto$ *C*) ≥ 2›
    using *C ST multi-member-split*[*OF C*] **unfolding** *twl-struct-invs-def*
    by (*cases S*; *cases T*)
      (*auto simp*: *twl-st-inv.simps twl-st-l-def*
        image-Un[*symmetric*])
  show *?B*
  proof (*cases* ‹*length* (*get-clauses-l S* $\propto$ *C*) > 2›)
    case *True*
    show *?thesis*
      using *L True LC* **by** *auto*
  next
    case *False*
    then show *?thesis*
      using *LC le2 L-w*
      by (*cases* ‹*get-clauses-l S* $\propto$ *C*›;
          cases ‹*tl* (*get-clauses-l S* $\propto$ *C*)›)
        auto
  qed
qed

lemma *trail-length-ge2*:
  assumes
    ST: ‹$(S,\ T) \in$ *twl-st-l None*› **and**
    list-invs: ‹*twl-list-invs S*› **and**
    struct-invs: ‹*twl-struct-invs T*› **and**
    LaC: ‹*Propagated L C* $\in$ *set* (*get-trail-l S*)› **and**

C0: ‹*C* > *0*›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
proof −
have *conv*:
‹(*get-trail-l S*, *get-trail T*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-unit-clauses-l S*)›
using *ST* **unfolding** *twl-st-l-def* **by** *auto*

have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting* ($state_W$*-of T*)› **and**
lev-inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* ($state_W$*-of T*)›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *fast+*

have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *ST lev-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st-l twl-st*)
have
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
using *list-invs C0 LaC* **by** (*auto simp*: *twl-list-invs-def all-conj-distrib*)
have ‹*twl-st-inv T*›
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
then show *le2*: ‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
using *C ST multi-member-split*[*OF C*] **unfolding** *twl-struct-invs-def*
by (*cases S*; *cases T*)
(*auto simp*: *twl-st-inv.simps twl-st-l-def*
image-Un[*symmetric*])
qed

lemma *is-annot-no-other-true-lit*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
C0: ‹*C* > *0*› **and**
LaC: ‹*Propagated La C* ∈ *set* (*get-trail-l S*)› **and**
LC: ‹*L* ∈ *set* (*get-clauses-l S* ∝ *C*)› **and**
L: ‹*L* ∈ *lits-of-l* (*get-trail-l S*)›
shows
‹*La* = *L*› **and**
‹*length* (*get-clauses-l S* ∝ *C*) > *2* ⟹ *L* = *get-clauses-l S* ∝ *C* ! *0*›
proof −
have *conv*:
‹(*get-trail-l S*, *get-trail T*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-unit-clauses-l S*)›
using *ST* **unfolding** *twl-st-l-def* **by** *auto*

obtain *M2 M1* **where**
tr-S: ‹*get-trail-l S* = *M2* @ *Propagated La C* # *M1*›
using *split-list*[*OF LaC*] **by** *blast*
then obtain *M2' M1'* **where**
tr-T: ‹*get-trail T* = *M2'* @ *Propagated La* (*mset* (*get-clauses-l S* ∝ *C*)) # *M1'*› **and**
M2: ‹(*M2*, *M2'*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-unit-clauses-l S*)› **and**
M1: ‹(*M1*, *M1'*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-unit-clauses-l S*)›
using *conv C0* **by** (*auto simp*: *list-all2-append1 list-all2-append2 list-all2-Cons1 list-all2-Cons2*
convert-lits-l-def list-rel-def convert-lit.simps dest!: *p2relD*)
have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting* ($state_W$*-of T*)› **and**
lev-inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* ($state_W$*-of T*)›

using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *fast+*
then have ‹*La* ∈# *mset* (*get-clauses-l S* ∝ *C*)› **and**
‹*M1′* ⊨*as CNot* (*remove1-mset La* (*mset* (*get-clauses-l S* ∝ *C*)))›
using *tr-T*
unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
by (*auto 5 5 simp*: *twl-st twl-st-l*)
then have
‹*M1* ⊨*as CNot* (*remove1-mset La* (*mset* (*get-clauses-l S* ∝ *C*)))›
using *M1 convert-lits-l-true-clss-clss* **by** *blast*
then have *all-false*: ‹−*K* ∈ *lits-of-l* (*get-trail-l S*)›
if ‹*K* ∈# *remove1-mset La* (*mset* (*get-clauses-l S* ∝ *C*))›
for *K*
using *that tr-S* **unfolding** *true-annots-true-cls-def-iff-negation-in-model*
by (*auto dest*!: *multi-member-split*)
have *La0*: ‹*length* (*get-clauses-l S* ∝ *C*) > *2* ⟹ *La* = *get-clauses-l S* ∝ *C* ! *0*› **and**
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)› **and**
‹*La* ∈ *set* (*watched-l* (*get-clauses-l S* ∝ *C*))›
using *list-invs tr-S C0* **by** (*auto simp*: *twl-list-invs-def all-conj-distrib*)
have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *ST lev-inv* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st-l twl-st*)
show ‹*La* = *L*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then have ‹*L* ∈# *remove1-mset La* (*mset* (*get-clauses-l S* ∝ *C*))›
using *LC* **by** *auto*
from *all-false*[*OF this*] **show** *False*
using *L n-d* **by** (*auto dest*: *no-dup-consistentD*)
qed
then show ‹*length* (*get-clauses-l S* ∝ *C*) > *2* ⟹ *L* = *get-clauses-l S* ∝ *C* ! *0*›
using *La0* **by** *simp*
qed

lemma *remove-one-annot-true-clause-cdcl-twl-restart-l2*:
assumes
rem: ‹*remove-one-annot-true-clause S T*› **and**
lst-invs: ‹*twl-list-invs S*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
n-d: ‹(*S*, *T′*) ∈ *twl-st-l None*› ‹*twl-struct-invs T′*›
shows ‹*cdcl-twl-restart-l S T*›
proof −
have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *n-d* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st twl-st-l*)

show *?thesis*
apply (*rule remove-one-annot-true-clause-cdcl-twl-restart-l*[*OF* - - ‹(*S*, *T′*) ∈ *twl-st-l None*›])
subgoal using *rem* **.**
subgoal using *lst-invs* **.**
subgoal using ‹*twl-struct-invs T′*› **.**
subgoal using *confl* **.**
subgoal using *upd* **.**
subgoal using *n-d* **.**

done
qed

lemma *remove-one-annot-true-clause-get-conflict-l*:
⟨*remove-one-annot-true-clause S T* $\Longrightarrow$ *get-conflict-l T* = *get-conflict-l S*⟩
by (*auto simp*: *remove-one-annot-true-clause.simps*)

lemma *rtranclp-remove-one-annot-true-clause-get-conflict-l*:
⟨*remove-one-annot-true-clause*** *S T* $\Longrightarrow$ *get-conflict-l T* = *get-conflict-l S*⟩
by (*induction rule*: *rtranclp-induct*) (*auto simp*: *remove-one-annot-true-clause-get-conflict-l*)

lemma *remove-one-annot-true-clause-clauses-to-update-l*:
⟨*remove-one-annot-true-clause S T* $\Longrightarrow$ *clauses-to-update-l T* = *clauses-to-update-l S*⟩
by (*auto simp*: *remove-one-annot-true-clause.simps*)

lemma *rtranclp-remove-one-annot-true-clause-clauses-to-update-l*:
⟨*remove-one-annot-true-clause*** *S T* $\Longrightarrow$ *clauses-to-update-l T* = *clauses-to-update-l S*⟩
by (*induction rule*: *rtranclp-induct*) (*auto simp*: *remove-one-annot-true-clause-clauses-to-update-l*)

lemma *cdcl-twl-restart-l-invs*:
assumes *ST*: ⟨(*S*, *T*) ∈ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs S*⟩ **and**
struct-invs: ⟨*twl-struct-invs T*⟩ **and** ⟨*cdcl-twl-restart-l S S'*⟩
shows ⟨∃ *T'*. (*S'*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs S'* ∧
clauses-to-update-l S' = {#} ∧ *cdcl-twl-restart T T'* ∧ *twl-struct-invs T'*⟩
using *cdcl-twl-restart-l-cdcl-twl-restart*[*OF ST list-invs struct-invs*]
cdcl-twl-restart-twl-struct-invs[*OF - struct-invs*]
by (*smt RETURN-ref-SPECD RETURN-rule assms*(*4*) *in-pair-collect-simp order-trans*)

lemma *rtranclp-cdcl-twl-restart-l-invs*:
assumes
⟨*cdcl-twl-restart-l*** *S S'*⟩ **and**
ST: ⟨(*S*, *T*) ∈ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs S*⟩ **and**
struct-invs: ⟨*twl-struct-invs T*⟩ **and**
⟨*clauses-to-update-l S* = {#}⟩
shows ⟨∃ *T'*. (*S'*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs S'* ∧
clauses-to-update-l S' = {#} ∧ *cdcl-twl-restart*** *T T'* ∧ *twl-struct-invs T'*⟩
using *assms*(*1*)
apply (*induction rule*: *rtranclp-induct*)
subgoal
using *assms*(*2*−) **apply** − **by** (*rule exI*[*of - T*]) *auto*
subgoal for *T U*
using *cdcl-twl-restart-l-invs*[*of T - U*] *assms*
by (*meson rtranclp.rtrancl-into-rtrancl*)
done

lemma *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*:
assumes
rem: ⟨*remove-one-annot-true-clause*** *S T*⟩ **and**
lst-invs: ⟨*twl-list-invs S*⟩ **and**
confl: ⟨*get-conflict-l S* = *None*⟩ **and**
upd: ⟨*clauses-to-update-l S* = {#}⟩ **and**
n-d: ⟨(*S*, *S'*) ∈ *twl-st-l None*⟩ ⟨*twl-struct-invs S'*⟩

shows ‹∃ *T'*. *cdcl-twl-restart-l*** *S T* ∧ (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-restart*** *S' T'* ∧
twl-struct-invs T'›
using *rem*
proof (*induction*)
case *base*
then show *?case*
using *assms* **apply** − **by** (*rule-tac x=S'* **in** *exI*) *auto*
next
case (*step U V*) **note** *st* = *this*(*1*) **and** *step* = *this*(*2*) **and** *IH* = *this*(*3*)
obtain *U'* **where**
IH: ‹*cdcl-twl-restart-l*** *S U*› **and**
UT': ‹(*U*, *U'*) ∈ *twl-st-l None*› **and**
S'U': ‹*cdcl-twl-restart*** *S' U'*›
using *IH* **by** *blast*
have ‹*twl-list-invs U*›
using *rtranclp-cdcl-twl-restart-l-list-invs*[*OF IH lst-invs*] **.**
have ‹*get-conflict-l U* = *None*›
using *rtranclp-remove-one-annot-true-clause-get-conflict-l*[*OF st*] *confl*
by *auto*
have ‹*clauses-to-update-l U* = {#}›
using *rtranclp-remove-one-annot-true-clause-clauses-to-update-l*[*OF st*] *upd*
by *auto*
have ‹*twl-struct-invs U'*›
by (*metis* (*no-types*, *hide-lams*) ‹*cdcl-twl-restart*** *S' U'*›
cdcl-twl-restart-twl-struct-invs n-d(*2*) *rtranclp-induct*)
have ‹*cdcl-twl-restart-l U V*›
apply (*rule remove-one-annot-true-clause-cdcl-twl-restart-l2*[*of - - U'*])
subgoal using *step* **.**
subgoal using ‹*twl-list-invs U*› **.**
subgoal using ‹*get-conflict-l U* = *None*› **.**
subgoal using ‹*clauses-to-update-l U* = {#}› **.**
subgoal using *UT'* **.**
subgoal using ‹*twl-struct-invs U'*› **.**
done
moreover obtain *V'* **where**
UT': ‹(*V*, *V'*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart U' V'*› **and**
‹*twl-struct-invs V'*›
using *cdcl-twl-restart-l-invs*[*OF UT'* - - ‹*cdcl-twl-restart-l U V*›] ‹*twl-list-invs U*›
‹*twl-struct-invs U'*›
by *blast*
ultimately show *?case*
using *S'U' IH* **by** *fastforce*
qed

definition *drop-clause-add-move-init* **where**
‹*drop-clause-add-move-init* = (λ(*M*, *N0*, *D*, *NE0*, *UE*, *Q*, *W*) *C*.
(*M*, *fmdrop C N0*, *D*, *add-mset* (*mset* (*N0* ∝ *C*)) *NE0*, *UE*, *Q*, *W*))›

lemma [*simp*]:
‹*get-trail-l* (*drop-clause-add-move-init V C*) = *get-trail-l V*›
by (*cases V*) (*auto simp*: *drop-clause-add-move-init-def*)

definition *drop-clause* **where**
‹*drop-clause* = (λ(*M*, *N0*, *D*, *NE0*, *UE*, *Q*, *W*) *C*.
(*M*, *fmdrop C N0*, *D*, *NE0*, *UE*, *Q*, *W*))›

lemma [*simp*]:
  ‹*get-trail-l* (*drop-clause* V C) = *get-trail-l* V›
  by (*cases* V) (*auto simp*: *drop-clause-def*)

definition *remove-all-annot-true-clause-one-imp*
where
‹*remove-all-annot-true-clause-one-imp* = (λ(C, S). *do* {
      if C $\in\#$ *dom-m* (*get-clauses-l* S) *then*
        if irred (*get-clauses-l* S) C
        then RETURN (*drop-clause-add-move-init* S C)
        else RETURN (*drop-clause* S C)
      else do {
        RETURN S
      }
  })›

definition *remove-one-annot-true-clause-imp-inv* **where**
  ‹*remove-one-annot-true-clause-imp-inv* S =
    (λ(i, T). *remove-one-annot-true-clause*** S T $\wedge$ *twl-list-invs* S $\wedge$ $i \le$ *length* (*get-trail-l* S) $\wedge$
      twl-list-invs S $\wedge$
      clauses-to-update-l S = *clauses-to-update-l* T $\wedge$
      literals-to-update-l S = *literals-to-update-l* T $\wedge$
      get-conflict-l T = *None* $\wedge$
      ($\exists$ S'. (S, S') $\in$ *twl-st-l None* $\wedge$ *twl-struct-invs* S') $\wedge$
      get-conflict-l S = *None* $\wedge$ *clauses-to-update-l* S = {#} $\wedge$
      length (*get-trail-l* S) = *length* (*get-trail-l* T) $\wedge$
      ($\forall$ j<i. *is-proped* (*rev* (*get-trail-l* T) ! j) $\wedge$ *mark-of* (*rev* (*get-trail-l* T) ! j) = *0*))›

definition *remove-all-annot-true-clause-imp-inv* **where**
  ‹*remove-all-annot-true-clause-imp-inv* S *xs* =
    (λ(i, T). *remove-one-annot-true-clause*** S T $\wedge$ *twl-list-invs* S $\wedge$ $i \le$ *length xs* $\wedge$
          twl-list-invs S $\wedge$ *get-trail-l* S = *get-trail-l* T $\wedge$
          ($\exists$ S'. (S, S') $\in$ *twl-st-l None* $\wedge$ *twl-struct-invs* S') $\wedge$
          get-conflict-l S = *None* $\wedge$ *clauses-to-update-l* S = {#})›

definition *remove-all-annot-true-clause-imp-pre* **where**
  ‹*remove-all-annot-true-clause-imp-pre* L S $\longleftrightarrow$
    (*twl-list-invs* S $\wedge$ *twl-list-invs* S $\wedge$
    ($\exists$ S'. (S, S') $\in$ *twl-st-l None* $\wedge$ *twl-struct-invs* S') $\wedge$
    get-conflict-l S = *None* $\wedge$ *clauses-to-update-l* S = {#} $\wedge$ L $\in$ *lits-of-l* (*get-trail-l* S))›

definition *remove-all-annot-true-clause-imp*
  :: ‹$'v$ *literal* $\Rightarrow$ $'v$ *twl-st-l* $\Rightarrow$ ($'v$ *twl-st-l*) *nres*›
where
‹*remove-all-annot-true-clause-imp* = (λL S. *do* {
    ASSERT(*remove-all-annot-true-clause-imp-pre* L S);
    xs $\leftarrow$ *SPEC*(λ*xs*.
       ($\forall$ $x$$\in$*set xs*. x $\in\#$ *dom-m* (*get-clauses-l* S) $\longrightarrow$ L $\in$ *set* ((*get-clauses-l* S)$\propto$$x$)));
    (-, T) $\leftarrow$ $WHILE_T{}^{\lambda(i,\ T).\ remove\text{-}all\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}inv\ S\ xs\ (i,\ T)}$
      (λ(i, T). i < *length xs*)
      (λ(i, T). *do* {
          ASSERT(i < *length xs*);
          if xs!i $\in\#$ *dom-m* (*get-clauses-l* T) $\wedge$ *length* ((*get-clauses-l* T) $\propto$ (*xs*!i)) $\neq$ *2*
          then do {

T ← remove-all-annot-true-clause-one-imp (xs!i, T);
ASSERT(remove-all-annot-true-clause-imp-inv S xs (i, T));
RETURN (i+1, T)
}
else
RETURN (i+1, T)
})
(0, S);
RETURN T
})⟩

definition *remove-one-annot-true-clause-one-imp-pre* **where**
⟨remove-one-annot-true-clause-one-imp-pre i T ⟷
(twl-list-invs T ∧ i < length (get-trail-l T) ∧
twl-list-invs T ∧
(∃ S′. (T, S′) ∈ twl-st-l None ∧ twl-struct-invs S′) ∧
get-conflict-l T = None ∧ clauses-to-update-l T = {#})⟩

definition *replace-annot-l* **where**
⟨replace-annot-l L C =
(λ(M, N, D, NE, UE, Q, W).
RES {(M′, N, D, NE, UE, Q, W)| M′.
(∃ M2 M1 C. M = M2 @ Propagated L C # M1 ∧ M′ = M2 @ Propagated L 0 # M1)})⟩

definition *remove-and-add-cls-l* **where**
⟨remove-and-add-cls-l C =
(λ(M, N, D, NE, UE, Q, W).
RETURN (M, fmdrop C N, D,
(if irred N C then add-mset (mset (N∝C)) else id) NE,
(if ¬irred N C then add-mset (mset (N∝C)) else id) UE, Q, W))⟩

The following progrom removes all clauses that are annotations. However, this is not compatible with binary clauses, since we want to make sure that they should not been deleted.

term *remove-all-annot-true-clause-imp*
definition *remove-one-annot-true-clause-one-imp*
where
⟨remove-one-annot-true-clause-one-imp = (λi S. do {
ASSERT(remove-one-annot-true-clause-one-imp-pre i S);
ASSERT(is-proped ((rev (get-trail-l S))!i));
(L, C) ← SPEC(λ(L, C). (rev (get-trail-l S))!i = Propagated L C);
ASSERT(Propagated L C ∈ set (get-trail-l S));
if C = 0 then RETURN (i+1, S)
else do {
ASSERT(C ∈# dom-m (get-clauses-l S));
S ← replace-annot-l L C S;
S ← remove-and-add-cls-l C S;
~~S ← remove-all-annot-true-clause-imp L S;~~
RETURN (i+1, S)
}
})⟩

definition *remove-one-annot-true-clause-imp* :: ⟨′v twl-st-l ⇒ (′v twl-st-l) nres⟩
where
⟨remove-one-annot-true-clause-imp = (λS. do {
k ← SPEC(λk. (∃ M1 M2 K. (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (get-trail-l S)) ∧

count-decided M1 = *0* ∧ *k* = *length M1*)
∨ (*count-decided* (*get-trail-l S*) = *0* ∧ *k* = *length* (*get-trail-l S*)));
(-, *S*) ← $WHILE_T{}^{remove\text{-}one\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}inv\ S}$
(λ(*i*, *S*). *i* < *k*)
(λ(*i*, *S*). *remove-one-annot-true-clause-one-imp i S*)
(*0*, *S*);
RETURN S
})›

lemma *remove-one-annot-true-clause-imp-same-length*:
‹*remove-one-annot-true-clause S T* ⟹ *length* (*get-trail-l S*) = *length* (*get-trail-l T*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*) (*auto simp*:)

lemma *rtranclp-remove-one-annot-true-clause-imp-same-length*:
‹*remove-one-annot-true-clause*** *S T* ⟹ *length* (*get-trail-l S*) = *length* (*get-trail-l T*)›
by (*induction rule*: *rtranclp-induct*) (*auto simp*: *remove-one-annot-true-clause-imp-same-length*)

lemma *remove-one-annot-true-clause-map-is-decided-trail*:
‹*remove-one-annot-true-clause S U* ⟹
map is-decided (*get-trail-l S*) = *map is-decided* (*get-trail-l U*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
auto

lemma *remove-one-annot-true-clause-map-mark-of-same-or-0*:
‹*remove-one-annot-true-clause S U* ⟹
mark-of (*get-trail-l S* ! *i*) = *mark-of* (*get-trail-l U* ! *i*) ∨ *mark-of* (*get-trail-l U* ! *i*) = *0*›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
(*auto simp*: *nth-append nth-Cons split*: *nat.split*)

lemma *remove-one-annot-true-clause-imp-inv-trans*:
‹*remove-one-annot-true-clause-imp-inv S* (*i*, *T*) ⟹ *remove-one-annot-true-clause-imp-inv T U* ⟹
remove-one-annot-true-clause-imp-inv S U›
using *rtranclp-remove-one-annot-true-clause-imp-same-length*[*of S T*]
by (*auto simp*: *remove-one-annot-true-clause-imp-inv-def*)

lemma *rtranclp-remove-one-annot-true-clause-map-is-decided-trail*:
‹*remove-one-annot-true-clause*** *S U* ⟹
map is-decided (*get-trail-l S*) = *map is-decided* (*get-trail-l U*)›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *remove-one-annot-true-clause-map-is-decided-trail*)

lemma *rtranclp-remove-one-annot-true-clause-map-mark-of-same-or-0*:
‹*remove-one-annot-true-clause*** *S U* ⟹
mark-of (*get-trail-l S* ! *i*) = *mark-of* (*get-trail-l U* ! *i*) ∨ *mark-of* (*get-trail-l U* ! *i*) = *0*›
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *remove-one-annot-true-clause-map-mark-of-same-or-0*)

lemma *remove-one-annot-true-clause-map-lit-of-trail*:
‹*remove-one-annot-true-clause S U* ⟹
map lit-of (*get-trail-l S*) = *map lit-of* (*get-trail-l U*)›
by (*induction rule*: *remove-one-annot-true-clause.induct*)
auto

lemma *rtranclp-remove-one-annot-true-clause-map-lit-of-trail*:
‹*remove-one-annot-true-clause*** *S U* ⟹

map lit-of (*get-trail-l S*) = *map lit-of* (*get-trail-l U*)⟩
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *remove-one-annot-true-clause-map-lit-of-trail*)

lemma *remove-one-annot-true-clause-reduce-dom-clauses*:
⟨*remove-one-annot-true-clause S U* ⟹
reduce-dom-clauses (*get-clauses-l S*) (*get-clauses-l U*)⟩
by (*induction rule*: *remove-one-annot-true-clause.induct*)
auto

lemma *rtranclp-remove-one-annot-true-clause-reduce-dom-clauses*:
⟨*remove-one-annot-true-clause*** *S U* ⟹
reduce-dom-clauses (*get-clauses-l S*) (*get-clauses-l U*)⟩
by (*induction rule*: *rtranclp-induct*)
(*auto dest*!: *remove-one-annot-true-clause-reduce-dom-clauses intro*: *reduce-dom-clauses-trans*)

lemma *decomp-nth-eq-lit-eq*:
assumes
⟨*M* = *M2* @ *Propagated L C'* # *M1*⟩ **and**
⟨*rev M* ! *i* = *Propagated L C*⟩ **and**
⟨*no-dup M*⟩ **and**
⟨*i* < *length M*⟩
shows ⟨*length M1* = *i*⟩ **and** ⟨*C* = *C'*⟩
proof −
have [*simp*]: ⟨*defined-lit M1* (*lit-of* (*M1* ! *i*))⟩ **if** ⟨*i* < *length M1*⟩ **for** *i*
using *that* **by** (*simp add*: *in-lits-of-l-defined-litD lits-of-def*)
have[*simp*]: ⟨*undefined-lit M2 L* ⟹
k < *length M2* ⟹
M2 ! *k* ≠ *Propagated L C*⟩ **for** *k*
using *defined-lit-def nth-mem* **by** *fastforce*
have[*simp*]: ⟨*undefined-lit M1 L* ⟹
k < *length M1* ⟹
M1 ! *k* ≠ *Propagated L C*⟩ **for** *k*
using *defined-lit-def nth-mem* **by** *fastforce*
have ⟨*M* ! (*length M* − *Suc i*) ∈ *set M*⟩
apply (*rule nth-mem*)
using *assms* **by** *auto*
from *split-list*[*OF this*] **show** ⟨*length M1* = *i*⟩ **and** ⟨*C* = *C'*⟩
using *assms*
by (*auto simp*: *nth-append nth-Cons nth-rev split*: *if-splits nat.splits*
elim!: *list-match-lel-lel*)
qed

lemma
assumes ⟨*no-dup M*⟩
shows
no-dup-same-annotD:
⟨*Propagated L C* ∈ *set M* ⟹ *Propagated L C'* ∈ *set M* ⟹ *C* = *C'*⟩ **and**
no-dup-no-propa-and-dec:
⟨*Propagated L C* ∈ *set M* ⟹ *Decided L* ∈ *set M* ⟹ *False*⟩
using *assms*
by (*auto dest*!: *split-list elim*: *list-match-lel-lel*)

lemma *remove-one-annot-true-clause-imp-inv-spec*:
assumes
annot: ⟨*remove-one-annot-true-clause-imp-inv S* (*i*+*1*, *U*)⟩ **and**

i-le: ‹i < *length* (*get-trail-l S*)› **and**
L: ‹*L* ∈ *lits-of-l* (*get-trail-l S*)› **and**
lev0: ‹*get-level* (*get-trail-l S*) *L* = *0*› **and**
LC: ‹*Propagated L 0* ∈ *set* (*get-trail-l U*)›
shows ‹*remove-all-annot-true-clause-imp L U*
≤ *SPEC* (λ*Sa*. *RETURN* (*i* + *1*, *Sa*)
≤ *SPEC* (λ*s*′. *remove-one-annot-true-clause-imp-inv S s*′ ∧
(*s*′, (*i*, *T*))
∈ *measure*
(λ(*i*, -). *length* (*get-trail-l S*) − *i*)))›
proof −

obtain *M N D NE UE WS Q* **where**
U: ‹*U* = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
by (*cases U*)
obtain *x* **where**
SU: ‹*remove-one-annot-true-clause*** *S* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)› **and**
‹*twl-list-invs S*› **and**
‹*i* + *1* ≤ *length* (*get-trail-l S*)› **and**
‹*twl-list-invs S*› **and**
‹*get-conflict-l S* = *None*› **and**
‹(*S*, *x*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs x*› **and**
‹*clauses-to-update-l S* = {#}› **and**
level0: ‹∀ *j*<*i* + *1*. *is-proped* (*rev* (*get-trail-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)) ! *j*)›**and**
mark0: ‹∀ *j*<*i* + *1*. *mark-of* (*rev* (*get-trail-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)) ! *j*) = *0*›
using *annot* **unfolding** *U prod.case remove-one-annot-true-clause-imp-inv-def*
by *blast*
obtain *U*′ **where**
‹*cdcl-twl-restart-l*** *S U*› **and**
U′*V*′: ‹(*U*, *U*′) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *x U*′› **and**
struvt-invs-V′: ‹*twl-struct-invs U*′›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF SU* ‹*twl-list-invs S*›
‹*get-conflict-l S* = *None*› ‹*clauses-to-update-l S* = {#}› ‹(*S*, *x*) ∈ *twl-st-l None*›
‹*twl-struct-invs x*›] **unfolding** *U*
by *auto*
moreover have ‹*twl-list-invs U*›
using ‹*twl-list-invs S*› *calculation*(*1*) *rtranclp-cdcl-twl-restart-l-list-invs* **by** *blast*
ultimately have *rem-U-U*: ‹*remove-one-annot-true-clause-imp-inv U* (*i* + *1*, *U*)›
using *level0 rtranclp-remove-one-annot-true-clause-clauses-to-update-l*[*OF SU*]
rtranclp-remove-one-annot-true-clause-get-conflict-l[*OF SU*] *mark0*
‹*clauses-to-update-l S* = {#}› ‹*get-conflict-l S* = *None*› *i-le*
arg-cong[*OF rtranclp-remove-one-annot-true-clause-map-lit-of-trail*[*OF SU*], *of length*]
unfolding *remove-one-annot-true-clause-imp-inv-def* **unfolding** *U*
by (*cases U*′) *fastforce*
then have *rem-true-U-U*: ‹*remove-all-annot-true-clause-imp-inv U xs* (*0*, *U*)› **for** *xs*
using *level0 rtranclp-remove-one-annot-true-clause-clauses-to-update-l*[*OF SU*]
rtranclp-remove-one-annot-true-clause-get-conflict-l[*OF SU*] ‹*twl-list-invs U*›
‹*clauses-to-update-l S* = {#}› ‹*get-conflict-l S* = *None*› *i-le*
arg-cong[*OF rtranclp-remove-one-annot-true-clause-map-lit-of-trail*[*OF SU*], *of length*]
unfolding *U remove-all-annot-true-clause-imp-inv-def remove-one-annot-true-clause-imp-inv-def*
by (*cases U*′) *blast*
moreover have *L-M*: ‹*L* ∈ *lits-of-l M*›
using *L arg-cong*[*OF rtranclp-remove-one-annot-true-clause-map-lit-of-trail*[*OF SU*], *of set*]
by (*simp add*: *lits-of-def*)

ultimately have *pre*: ‹*remove-all-annot-true-clause-imp-pre L U*›
unfolding *remove-all-annot-true-clause-imp-pre-def remove-all-annot-true-clause-imp-inv-def prod.case U* **by** *force*

have *remove-all-annot-true-clause-one-imp*:
‹*remove-all-annot-true-clause-one-imp* (*xs* ! *k*, *V*)
≤ *SPEC*
(λ*T*. *do* {
- ← *ASSERT* (*remove-all-annot-true-clause-imp-inv U xs* (*k*, *T*));
RETURN (*k* + *1*, *T*)
} ≤ *SPEC*
(λ*s*′. (*case s*′ *of*
(*i*, *T*) ⇒
remove-all-annot-true-clause-imp-inv U xs (*i*, *T*)) ∧
(*case s*′ *of*
(*uu-*, *W*) ⇒
remove-one-annot-true-clause-imp-inv U (*i* + *1*, *W*)) ∧
(*s*′, *s*) ∈ *measure* (λ(*i*, -). *length xs* − *i*)))›
if
xs: ‹*xs* ∈ {*xs*.
∀ *x*∈*set xs*.
x ∈# *dom-m* (*get-clauses-l U*) ⟶ *L* ∈ *set* (*get-clauses-l U* ∝ *x*)}› **and**
I′: ‹*case s of* (*i*, *T*) ⇒ *remove-all-annot-true-clause-imp-inv U xs* (*i*, *T*)› **and**
I: ‹*case s of* (*uu-*, *W*) ⇒ *remove-one-annot-true-clause-imp-inv U* (*i* + *1*, *W*)› **and**
cond: ‹*case s of* (*i*, *T*) ⇒ *i* < *length xs*› **and**
s: ‹*s* = (*k*, *V*)› **and**
k-le: ‹*k* < *length xs*› **and**
dom: ‹*xs* ! *k* ∈# *dom-m* (*get-clauses-l V*) ∧
length (*get-clauses-l V* ∝ (*xs* ! *k*)) ≠ *2*›
for *s k V xs*
proof −
obtain *x* **where**
UU′: ‹*remove-one-annot-true-clause*** *U V*› **and**
i-le: ‹*i* + *1* ≤ *length* (*get-trail-l U*)› **and**
list-invs: ‹*twl-list-invs U*› **and**
confl: ‹*get-conflict-l U* = *None*› **and**
Ux: ‹(*U*, *x*) ∈ *twl-st-l None*› **and**
struct-x: ‹*twl-struct-invs x*› **and**
upd: ‹*clauses-to-update-l U* = {#}› **and**
all-level0: ‹∀ *j*<*i* + *1*. *is-proped* (*rev* (*get-trail-l V*) ! *j*)› **and**
all-mark0: ‹∀ *j*<*i* + *1*. *mark-of* (*rev* (*get-trail-l V*) ! *j*) = *0*› **and**
lits-upd: ‹*literals-to-update-l U* = *literals-to-update-l V*› **and**
clss-upd: ‹*clauses-to-update-l U* = *clauses-to-update-l V*› **and**
confl-V: ‹*get-conflict-l V* = *None*› **and**
tr: ‹*get-trail-l U* = *get-trail-l V*›
using *I*′ *I* **unfolding** *s prod.case remove-one-annot-true-clause-imp-inv-def remove-all-annot-true-clause-imp-inv-def*
by *blast*
have *n-d*: ‹*no-dup* (*get-trail-l U*)›
using *Ux struct-x* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st twl-st-l*)
have *SU*′: ‹*remove-one-annot-true-clause*** *S V*›
using *SU UU*′ **unfolding** *U* **by** *simp*
have ‹*get-level M L* = *0*›
using *lev0 rtranclp-remove-one-annot-true-clause-map-is-decided-trail*[*OF SU*]

rtranclp-remove-one-annot-true-clause-map-lit-of-trail[*OF SU*]
U trail-renumber-get-level[*of* ‹*get-trail-l S*› ‹*get-trail-l U*› *L*]
by *force*
have *red*: ‹*reduce-dom-clauses* (*get-clauses-l U*)
(*get-clauses-l V*)›
using *rtranclp-remove-one-annot-true-clause-reduce-dom-clauses*[*OF UU′*] **unfolding** *U*
by *simp*
then have [*simp*]: ‹$N \propto (xs\ !\ k)$ = *get-clauses-l V* $\propto (xs\ !\ k)$› **and**
dom-VU: ‹$\bigwedge C.\ C \in\#$ *dom-m* (*get-clauses-l V*) $\longrightarrow C \in\#$ *dom-m* (*get-clauses-l U*)›
using *dom* **unfolding** *reduce-dom-clauses-def U* **by** *simp-all*
obtain V' **where**
‹*cdcl-twl-restart-l*** *U V*› **and**
$U'V'$: ‹$(V,\ V') \in$ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *x* V'› **and**
struvt-invs-V': ‹*twl-struct-invs* V'›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF UU′* ‹*twl-list-invs U*›
‹*get-conflict-l U = None*› ‹*clauses-to-update-l U* = {#}› ‹$(U,\ x) \in$ *twl-st-l None*›
‹*twl-struct-invs x*›]
by *auto*
have *list-invs-*U': ‹*twl-list-invs V*›
using ‹*cdcl-twl-restart-l*** *U V*› ‹*twl-list-invs U*›
rtranclp-cdcl-twl-restart-l-list-invs **by** *blast*

have *dom-N*: ‹*xs* ! *k* $\in\#$ *dom-m* (*get-clauses-l V*)›
using *dom red* **unfolding** *s*
by (*auto simp del*: *nth-mem simp*: *reduce-dom-clauses-def*)

have *xs-k-0*: ‹$0 <$ *xs* ! *k*›
apply (*rule ccontr*)
using *dom list-invs-*U' **by** (*auto simp*: *twl-list-invs-def*)
have *L-set*: ‹$L \in$ *set* (*get-clauses-l V* $\propto$ (*xs*!*k*))›
using *xs cond nth-mem*[*of k xs*] *dom-N dom-VU*[*of* ‹*xs*!*k*›] **unfolding** *s U*
by (*auto simp del*: *nth-mem*)
have ‹*no-dup M*›
using *n-d* **unfolding** *U* **by** *simp*
then have *no-already-annot*: ‹*Propagated Laa* (*xs* ! *k*) $\in$ *set* (*get-trail-l V*) $\Longrightarrow$ *False*› **for** *Laa*
using *is-annot-iff-annotates-first*[*OF* $U'V'$ *list-invs-*U' *struvt-invs-*V' *xs-k-0*] *LC*
is-annot-no-other-true-lit[*OF* $U'V'$ *list-invs-*U' *struvt-invs-*V' *xs-k-0*, *of Laa L*]
L-set L-M xs-k-0 tr **unfolding** *U*
by (*auto dest*: *no-dup-same-annotD*)
let *?U′* = ‹(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
have *V*: ‹*V* = (*M*, *get-clauses-l V*, *D*, *get-unit-init-clauses-l V*,
get-unit-learned-clauses-l V, *WS*, *Q*)›
using *confl upd lits-upd tr clss-upd confl-V* **unfolding** *U*
by (*cases V*) *auto*
let *?V* = ‹(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
let *?Vt* = ‹*drop-clause-add-move-init V* (*xs*!*k*)›
let *?Vf* = ‹*drop-clause V* (*xs*!*k*)›
have ‹*remove-one-annot-true-clause V ?Vt*›
if ‹*irred* (*get-clauses-l V*) (*xs* ! *k*)›
apply (*subst* (*2*) *V*)
apply (*subst V*)
unfolding *drop-clause-add-move-init-def prod.case*
apply (*rule remove-one-annot-true-clause.remove-irred*[*of L*])
subgoal using ‹$L \in$ *lits-of-l M*› **.**
subgoal using ‹*get-level M L* = *0*› **.**

subgoal using *dom* **by** *simp*
subgoal using *L-set* **by** *auto*
subgoal using *that* **.**
subgoal using *no-already-annot tr* **unfolding** *U* **by** *auto*
done
then have *UV-irred*: ‹*remove-one-annot-true-clause*** *U ?Vt*›
if ‹*irred* (*get-clauses-l V*) (*xs* ! *k*)›
using *UU′ that* **by** *simp*
have ‹*remove-one-annot-true-clause V ?Vf*›
if ‹¬*irred* (*get-clauses-l V*) (*xs* ! *k*)›
apply (*subst* (*2*) *V*)
apply (*subst V*)
unfolding *drop-clause-def prod.case*
apply (*rule remove-one-annot-true-clause.delete*)
subgoal using *dom* **by** *simp*
subgoal using *that* **.**
subgoal using *no-already-annot tr* **unfolding** *U* **by** *auto*
done
then have *UV-red*: ‹*remove-one-annot-true-clause*** *U ?Vf*›
if ‹¬*irred* (*get-clauses-l V*) (*xs* ! *k*)›
using *UU′ that* **by** *simp*
have *i-le*: ‹*Suc i* ≤ *length M*›
using *annot assms*(*2*) **unfolding** *U remove-one-annot-true-clause-imp-inv-def*
by *auto*
have *1*: ‹*remove-one-annot-true-clause-imp-inv U* (*Suc i*, *?Vt*)›
if ‹*irred* (*get-clauses-l V*) (*xs* ! *k*)›
using *UV-irred that* ‹*twl-list-invs U*› *i-le all-level0 all-mark0*
‹*get-conflict-l U* = *None*› ‹*clauses-to-update-l U* = {#}› ‹(*U*, *x*) ∈ *twl-st-l None*›
‹*twl-struct-invs x*› **unfolding** *U*
unfolding *remove-one-annot-true-clause-imp-inv-def prod.case*
apply (*intro conjI*)
subgoal by *auto*
subgoal by *auto*
subgoal using *i-le* **by** *auto*
subgoal using *tr* **by** (*cases V*) (*auto simp*: *drop-clause-add-move-init-def U*)
subgoal using *clss-upd* **by** (*cases V*) (*auto simp*: *drop-clause-add-move-init-def U*)
subgoal using *lits-upd* **by** (*cases V*) (*auto simp*: *drop-clause-add-move-init-def U*)
subgoal using *confl-V* **by** (*cases V*) (*auto simp*: *drop-clause-add-move-init-def U*)
subgoal by *blast*
subgoal by *auto*
subgoal by *auto*
subgoal using *tr* **by** (*cases V*) (*auto simp*: *drop-clause-add-move-init-def U*)
subgoal using *tr* **by** (*cases V*) (*auto simp*: *drop-clause-add-move-init-def U*)
done
have *2*: ‹*remove-one-annot-true-clause-imp-inv U* (*Suc i*, *?Vf*)›
if ‹¬*irred* (*get-clauses-l V*) (*xs* ! *k*)›
using *UV-red that* ‹*twl-list-invs U*› *i-le all-level0 all-mark0*
‹*get-conflict-l U* = *None*› ‹*clauses-to-update-l U* = {#}› ‹(*U*, *x*) ∈ *twl-st-l None*›
‹*twl-struct-invs x*› **unfolding** *U*
unfolding *remove-one-annot-true-clause-imp-inv-def prod.case*
apply (*intro conjI*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *tr* **by** (*cases V*) (*auto simp*: *drop-clause-def U*)
subgoal using *clss-upd* **by** (*cases V*) (*auto simp*: *drop-clause-def U*)

subgoal using *lits-upd* **by** (*cases V*) (*auto simp*: *drop-clause-def U*)
subgoal using *confl-V* **by** (*cases V*) (*auto simp*: *drop-clause-def U*)
subgoal by *blast*
subgoal by *auto*
subgoal by *auto*
subgoal using *tr* **by** (*cases V*) (*auto simp*: *drop-clause-def U*)
subgoal using *tr* **by** (*cases V*) (*auto simp*: *drop-clause-def U*)
done
have ‹*remove-all-annot-true-clause-imp-inv U xs*
(*k*, *?Vt*)›
if ‹*irred* (*get-clauses-l V*) (*xs* ! *k*)›
proof −
have ∃ *p*. (*U*, *p*) ∈ *twl-st-l None* ∧ *twl-struct-invs p*
using *Ux struct-x*
by *meson*
then show *?thesis*
using *that Ux struct-x list-invs i-le confl upd UV-irred cond tr*
unfolding *remove-all-annot-true-clause-imp-inv-def prod.case s*
by (*simp add*: *less-imp-le-nat*)
qed
moreover have ‹*remove-all-annot-true-clause-imp-inv U xs*
(*k*, *?Vf*)›
if ‹¬*irred* (*get-clauses-l V*) (*xs* ! *k*)›
proof −
have ∃ *p*. (*U*, *p*) ∈ *twl-st-l None* ∧ *twl-struct-invs p*
using *Ux struct-x*
by *meson*
then show *?thesis*
using *that Ux struct-x list-invs i-le confl upd UV-red cond tr*
unfolding *remove-all-annot-true-clause-imp-inv-def prod.case*
by (*simp add*: *less-imp-le-nat s*)
qed
ultimately show *?thesis*
using *dom 1 2 cond*
unfolding *remove-all-annot-true-clause-one-imp-def s*
by (*auto simp*:
Suc-le-eq remove-all-annot-true-clause-imp-inv-def)
qed
have *remove-all-annot-true-clause-imp-inv-Suc*:
‹*remove-all-annot-true-clause-imp-inv S xs* (*Suc i*, *T*)›
if ‹*remove-all-annot-true-clause-imp-inv S xs* (*i*, *T*)› **and**
‹*i* < *length xs*›
for *xs*
using *that*
by (*auto simp*: *remove-all-annot-true-clause-imp-inv-def*)
have *one-all*: ‹*remove-one-annot-true-clause-imp-inv S* (*Suc i*, *T*) ⟹
remove-all-annot-true-clause-imp-inv S xs (*a*, *T*) ⟹
Suc a ≤ *length xs* ⟹
remove-all-annot-true-clause-imp-inv S xs (*Suc a*, *T*)› **for** *S T a xs*
unfolding *remove-one-annot-true-clause-imp-inv-def remove-all-annot-true-clause-imp-inv-def*
by *auto*

show *?thesis*
unfolding *remove-all-annot-true-clause-imp-def prod.case assert-bind-spec-conv*
apply (*subst intro-spec-refine-iff*[*of - - Id*, *simplified*])
apply (*intro ballI conjI*)

    subgoal using *pre* **unfolding** *U* **.**
    subgoal for *xs*
      apply (*refine-vcg*
        WHILEIT-rule-stronger-inv[**where**
          $R = \langle$*measure* (λ(*i*, -). *length xs* − *i*)$\rangle$ **and**
          $I' = \langle\lambda$(-, *W*). *remove-one-annot-true-clause-imp-inv U* (*i*+*1*, *W*)$\rangle$])
      subgoal by *auto*
      subgoal using *rem-true-U-U* **unfolding** *U* **by** *auto*
      subgoal using *rem-U-U* **unfolding** *U* **by** *auto*
      subgoal by *simp*
      apply (*rule remove-all-annot-true-clause-one-imp*; *assumption*)
      subgoal by (*auto simp*: *remove-all-annot-true-clause-imp-inv-Suc U one-all*)
      subgoal by (*auto simp*: *remove-all-annot-true-clause-imp-inv-Suc U one-all*)
      subgoal by (*auto simp*: *remove-all-annot-true-clause-imp-inv-Suc U one-all*)
      subgoal
        apply (*rule remove-one-annot-true-clause-imp-inv-trans*[*OF annot*])
        apply *auto*
        done
      subgoal using *i-le* **by** *auto*
      done
    done
qed

lemma *RETURN-le-RES-no-return*:
  $\langle f \leq SPEC\ (\lambda S.\ g\ S \in \Phi) \Longrightarrow do\ \{S \leftarrow f;\ RETURN\ (g\ S)\} \leq RES\ \Phi\rangle$
  by (*cases f*) (*auto simp*: *RES-RETURN-RES*)

lemma *remove-one-annot-true-clause-one-imp-spec*:
  assumes
    I: ⟨*remove-one-annot-true-clause-imp-inv S iT*⟩ **and**
    cond: ⟨*case iT of* (*i*, *S*) ⇒ *i* < *length* (*get-trail-l S*)⟩ **and**
    iT: ⟨*iT* = (*i*, *T*)⟩ **and**
    proped: ⟨*is-proped* (*rev* (*get-trail-l S*) ! *i*)⟩
  shows ⟨*remove-one-annot-true-clause-one-imp i T*
        ≤ *SPEC* ($\lambda s'$. *remove-one-annot-true-clause-imp-inv S s'* ∧
              (s', *iT*) ∈ *measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*))⟩
proof −
  obtain *M N D NE UE WS Q* **where** *T*: ⟨*T* = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
    by (*cases T*)

  obtain *x* **where**
    ST: ⟨*remove-one-annot-true-clause*** *S T*⟩ **and**
    ⟨*twl-list-invs S*⟩ **and**
    ⟨*i* ≤ *length* (*get-trail-l S*)⟩ **and**
    ⟨*twl-list-invs S*⟩ **and**
    ⟨(*S*, *x*) ∈ *twl-st-l None*⟩ **and**
    ⟨*twl-struct-invs x*⟩ **and**
    confl: ⟨*get-conflict-l S* = *None*⟩ **and**
    upd: ⟨*clauses-to-update-l S* = {#}⟩ **and**
    level0: ⟨∀*j*<*i*. *is-proped* (*rev* (*get-trail-l T*) ! *j*)⟩ **and**
    mark0: ⟨∀*j*<*i*. *mark-of* (*rev* (*get-trail-l T*) ! *j*) = *0*⟩ **and**
    le: ⟨*length* (*get-trail-l S*) = *length* (*get-trail-l T*)⟩ **and**
    clss-upd: ⟨*clauses-to-update-l S* = *clauses-to-update-l T*⟩ **and**
    lits-upd: ⟨*literals-to-update-l S* = *literals-to-update-l T*⟩
    using *I* **unfolding** *remove-one-annot-true-clause-imp-inv-def iT prod.case* **by** *blast*
  then have *list-invs-T*: ⟨*twl-list-invs T*⟩

by (*meson rtranclp-cdcl-twl-restart-l-list-invs*
rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2)
obtain x' **where**
Tx': ‹(*T*, x') ∈ *twl-st-l None*› **and**
struct-invs-T: ‹*twl-struct-invs* x'›
using ‹(*S*, *x*) ∈ *twl-st-l None*› ‹*twl-list-invs S*› ‹*twl-struct-invs x*› *confl*
rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2 ST upd **by** *blast*
then have *n-d*: ‹*no-dup* (*get-trail-l T*)›
unfolding *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st twl-st-l*)

have *D*: ‹*D* = *None*› **and** *WS*: ‹*WS* = {#}›
using *confl upd rtranclp-remove-one-annot-true-clause-clauses-to-update-l*[*OF ST*]
using *rtranclp-remove-one-annot-true-clause-get-conflict-l*[*OF ST*] **unfolding** *T* **by** *auto*
have *lits-of-ST*: ‹*lits-of-l* (*get-trail-l S*) = *lits-of-l* (*get-trail-l T*)›
using *arg-cong*[*OF rtranclp-remove-one-annot-true-clause-map-lit-of-trail*[*OF ST*], *of set*]
by (*simp add*: *lits-of-def*)

have *rem-one-annot-i-T*: ‹*remove-one-annot-true-clause-one-imp-pre i T*›
using *Tx' struct-invs-T level0 cond list-invs-T D WS*
unfolding *remove-one-annot-true-clause-one-imp-pre-def iT T prod.case*
by *fastforce*
have
annot-in-dom: ‹*C* ∈# *dom-m* (*get-clauses-l T*)› (**is** *?annot*)
if
‹*case LC of* (*L*, *C*) ⇒ *rev* (*get-trail-l T*) ! *i* = *Propagated L C*› **and**
‹*LC* = (*L*, *C*)› **and**
‹¬(*C* = *0*)›
for *LC L C*
proof −
have ‹*rev* (*get-trail-l T*)!*i* ∈ *set* (*get-trail-l T*)›
using *list-invs-T assms le* **unfolding** *T*
by (*auto simp*: *twl-list-invs-def rev-nth*)
then show *?annot*
using *list-invs-T that le* **unfolding** *T*
by (*auto simp*: *twl-list-invs-def simp del*: *nth-mem*)
qed
have *replace-annot-l*:
‹*replace-annot-l L C T*
≤ *SPEC*
(λ*Sa. do* {
S ← *remove-and-add-cls-l C Sa*;
RETURN (*i* + *1*, *S*)
} ≤ *SPEC*
(λ*s'. remove-one-annot-true-clause-imp-inv S s'* ∧
(*s'*, *iT*)
∈ *measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*)))›
if
rem-one: ‹*remove-one-annot-true-clause-one-imp-pre i T*› **and**
‹*is-proped* (*rev* (*get-trail-l T*) ! *i*)› **and**
LC-d: ‹*case LC of* (*L*, *C*) ⇒ *rev* (*get-trail-l T*) ! *i* = *Propagated L C*› **and**
LC: ‹*LC* = (*L*, *C*)› **and**
LC-T: ‹*Propagated L C* ∈ *set* (*get-trail-l T*)› **and**
‹*C* ≠ *0*› **and**
dom: ‹*C* ∈# *dom-m* (*get-clauses-l T*)›

```
    for LC C L
  proof -
    have ‹i < length M›
      using rem-one unfolding remove-one-annot-true-clause-one-imp-pre-def T by auto

    {
      fix M2 Ca M1
      assume M: ‹M = M2 @ Propagated L Ca # M1› and ‹irred N Ca›
      have n-d: ‹no-dup M›
        using n-d unfolding T by auto
      then have [simp]: ‹Ca = C›
        using LC-T
        by (auto simp: T M dest!: in-set-definedD)
      have ‹Ca > 0›
        using that(6) by auto
      let ?U = ‹(M2 @ Propagated L 0 # M1, fmdrop Ca N, D, add-mset (mset (N ∝ Ca)) NE, UE,
WS, Q)›

      have lev: ‹get-level (M2 @ Propagated L C # M1) L = 0› and
        M1: ‹length M1 = i›
        using n-d level0 LC-d decomp-nth-eq-lit-eq(1)[OF M
    LC-d[unfolded T get-trail-l.simps LC prod.case]
    n-d ‹i < length M›]
  unfolding M T
      apply (auto simp: count-decided-0-iff nth-append nth-Cons is-decided-no-proped-iff
        in-set-conv-nth rev-nth
       split: if-splits)
       by (metis diff-less gr-implies-not0 linorder-neqE-nat nth-rev-alt rev-nth zero-less-Suc)

      have TU: ‹remove-one-annot-true-clause T ?U›
        unfolding T M
  apply (rule remove-one-annot-true-clause.remove-irred-trail)
  using ‹irred N Ca› ‹Ca > 0› dom lev
  by (auto simp: T M)
      moreover {
  have ‹length (get-trail-l ?U) = length (get-trail-l T)›
    using TU by (auto simp: remove-one-annot-true-clause.simps T M)
  then have ‹j<i ⟹ is-proped (rev (get-trail-l ?U) ! j)› for j
    using arg-cong[OF remove-one-annot-true-clause-map-is-decided-trail[OF TU],
     of ‹λxs. xs ! (length (get-trail-l ?U) - Suc j)›] level0 ‹i < length M›
    by (auto simp: rev-nth T is-decided-no-proped-iff M
      nth-append nth-Cons split: nat.splits)
      }
      moreover {
  have ‹length (get-trail-l ?U) = length (get-trail-l T)›
    using TU by (auto simp: remove-one-annot-true-clause.simps T M)
  then have ‹j<i ⟹ mark-of (rev (get-trail-l ?U) ! j) = 0› for j
    using remove-one-annot-true-clause-map-mark-of-same-or-0[OF TU,
      of ‹(length (get-trail-l ?U) - Suc j)›] mark0 ‹i < length M›
    by (auto simp: rev-nth T is-decided-no-proped-iff M
      nth-append nth-Cons split: nat.splits)
      }
      moreover have ‹length (get-trail-l S) = length (get-trail-l ?U)›
  using le TU by (auto simp: T M split: if-splits)
      moreover have ‹∃ S'. (S, S') ∈ twl-st-l None ∧ twl-struct-invs S'›
        by (rule exI[of - x])
```

(*use* ‹(*S*, *x*) ∈ *twl-st-l None*› ‹*twl-struct-invs x*› **in** *blast*)
ultimately have *1*: ‹*remove-one-annot-true-clause-imp-inv S* (*Suc i*, *?U*)›
using ‹*twl-list-invs S*› ‹*i* ≤ *length* (*get-trail-l S*)›
‹(*S*, *x*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs x*› **and**
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}› **and**
‹∀ *j*<*i*. *is-proped* (*rev* (*get-trail-l T*) ! *j*)› **and**
‹∀ *j*<*i*. *mark-of* (*rev* (*get-trail-l T*) ! *j*) = *0*› **and**
le T clss-upd lits-upd ST TU D M1 ‹*i* < *length M*›
unfolding *remove-one-annot-true-clause-imp-inv-def prod.case*
by (*auto simp*: *less-Suc-eq nth-append*)
have *2*: ‹*length* (*get-trail-l S*) − *Suc i* < *length* (*get-trail-l S*) − *i*›
by (*simp add*: *T* ‹*i* < *length M*› *diff-less-mono2 le*)
note *1 2*
}
moreover {
fix *M2 Ca M1*
assume *M*: ‹*M* = *M2* @ *Propagated L Ca* # *M1*› **and** ‹¬*irred N Ca*›
have *n-d*: ‹*no-dup M*›
using *n-d* **unfolding** *T* **by** *auto*
then have [*simp*]: ‹*Ca* = *C*›
using *LC-T*
by (*auto simp*: *T M dest*!: *in-set-definedD*)
have ‹*Ca* > *0*›
using *that*(*6*) **by** *auto*
let *?U* = ‹(*M2* @ *Propagated L 0* # *M1*, *fmdrop Ca N*, *D*, *NE*,
add-mset (*mset* (*N* ∝ *Ca*)) *UE*, *WS*, *Q*)›

have *lev*: ‹*get-level* (*M2* @ *Propagated L C* # *M1*) *L* = *0*› **and**
M1: ‹*length M1* = *i*›
using *n-d level0 LC-d decomp-nth-eq-lit-eq*(*1*)[*OF M*
LC-d[*unfolded T get-trail-l.simps LC prod.case*]
n-d ‹*i* < *length M*›]
unfolding *M T*
apply (*auto simp*: *count-decided-0-iff nth-append nth-Cons is-decided-no-proped-iff*
in-set-conv-nth rev-nth
split: *if-splits*)
by (*metis diff-less gr-implies-not0 linorder-neqE-nat nth-rev-alt rev-nth zero-less-Suc*)

have *TU*: ‹*remove-one-annot-true-clause T ?U*›
unfolding *T M*
apply (*rule remove-one-annot-true-clause.remove-red-trail*)
using ‹¬*irred N Ca*› ‹*Ca* > *0*› *dom lev*
by (*auto simp*: *T M*)
moreover {
have ‹*length* (*get-trail-l ?U*) = *length* (*get-trail-l T*)›
using *TU* **by** (*auto simp*: *remove-one-annot-true-clause.simps T M*)
then have ‹*j*<*i* ⟹ *is-proped* (*rev* (*get-trail-l ?U*) ! *j*)› **for** *j*
using *arg-cong*[*OF remove-one-annot-true-clause-map-is-decided-trail*[*OF TU*],
of ‹λ*xs*. *xs* ! (*length* (*get-trail-l ?U*) − *Suc j*)›] *level0* ‹*i* < *length M*›
by (*auto simp*: *rev-nth T is-decided-no-proped-iff M*
nth-append nth-Cons split: *nat.splits*)
}
moreover {
have ‹*length* (*get-trail-l ?U*) = *length* (*get-trail-l T*)›

using *TU* **by** (*auto simp*: *remove-one-annot-true-clause.simps T M*)
then have ‹$j<i \Longrightarrow$ *mark-of* (*rev* (*get-trail-l ?U*) ! *j*) = *0*› **for** *j*
using *remove-one-annot-true-clause-map-mark-of-same-or-0*[*OF TU*,
of ‹(*length* (*get-trail-l ?U*) − *Suc j*)›] *mark0* ‹*i* < *length M*›
by (*auto simp*: *rev-nth T is-decided-no-proped-iff M*
nth-append nth-Cons split: *nat.splits*)
}
moreover have ‹*length* (*get-trail-l S*) = *length* (*get-trail-l ?U*)›
using *le TU* **by** (*auto simp*: *T M split*: *if-splits*)
moreover have ‹$\exists S'$. (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-struct-invs S'*›
by (*rule exI*[*of - x*])
(*use* ‹(*S*, *x*) ∈ *twl-st-l None*› ‹*twl-struct-invs x*› **in** *blast*)
ultimately have *1*: ‹*remove-one-annot-true-clause-imp-inv S* (*Suc i*, *?U*)›
using ‹*twl-list-invs S*› ‹$i \leq$ *length* (*get-trail-l S*)›
‹(*S*, *x*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs x*› **and**
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}› **and**
‹$\forall j<i$. *is-proped* (*rev* (*get-trail-l T*) ! *j*)› **and**
‹$\forall j<i$. *mark-of* (*rev* (*get-trail-l T*) ! *j*) = *0*› **and**
le T clss-upd lits-upd ST TU D cond ‹*i* < *length M*› *M1*
unfolding *remove-one-annot-true-clause-imp-inv-def prod.case*
by (*auto simp*: *less-Suc-eq nth-append*)
have *2*: ‹*length* (*get-trail-l S*) − *Suc i* < *length* (*get-trail-l S*) − *i*›
by (*simp add*: *T* ‹*i* < *length M*› *diff-less-mono2 le*)
note *1 2*
}
moreover have ‹*C* = *Ca*› **if** ‹*M* = *M2* @ *Propagated L Ca* # *M1*› **for** *M1 M2 Ca*
using *LC-T n-d*
by (*auto simp*: *T that dest*!: *in-set-definedD*)
ultimately show *?thesis*
using *dom cond*
by (*auto simp*: *remove-and-add-cls-l-def*
replace-annot-l-def T iT
intro!: *RETURN-le-RES-no-return*)
qed

have *rev-set*: ‹*rev* (*get-trail-l T*) ! *i* ∈ *set* (*get-trail-l T*)›
using *assms*
by (*metis length-rev nth-mem rem-one-annot-i-T*
remove-one-annot-true-clause-one-imp-pre-def set-rev)
show *?thesis*
unfolding *remove-one-annot-true-clause-one-imp-def*
apply *refine-vcg*
subgoal using *rem-one-annot-i-T* **unfolding** *iT T* **by** *simp*
subgoal using *proped I le*
rtranclp-remove-one-annot-true-clause-map-is-decided-trail[*of S T*,
THEN arg-cong, *of* ‹λxs. (*rev xs*) ! *i*›]
unfolding *iT T remove-one-annot-true-clause-imp-inv-def*
remove-one-annot-true-clause-one-imp-pre-def
by (*auto simp add*: *All-less-Suc rev-map is-decided-no-proped-iff*)
subgoal
using *rev-set* **unfolding** *T*
by *auto*
subgoal using *I le* **unfolding** *iT T remove-one-annot-true-clause-imp-inv-def*
remove-one-annot-true-clause-one-imp-pre-def

by (*auto simp add*: *All-less-Suc*)
subgoal using *cond le* **unfolding** *iT T remove-one-annot-true-clause-one-imp-pre-def* **by** *auto*
subgoal by (*rule annot-in-dom*)
subgoal for *LC L C*
by (*rule replace-annot-l*)
done

qed

lemma *remove-one-annot-true-clause-count-dec*: ‹*remove-one-annot-true-clause S b* $\Longrightarrow$
count-decided (*get-trail-l S*) = *count-decided* (*get-trail-l b*)›
by (*auto simp*: *remove-one-annot-true-clause.simps*)

lemma *rtranclp-remove-one-annot-true-clause-count-dec*:
‹*remove-one-annot-true-clause*** *S b* $\Longrightarrow$
count-decided (*get-trail-l S*) = *count-decided* (*get-trail-l b*)›
by (*induction rule*: *rtranclp-induct*)
(*auto simp*: *remove-one-annot-true-clause-count-dec*)

lemma *remove-one-annot-true-clause-imp-spec*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}›
shows ‹*remove-one-annot-true-clause-imp S* ≤ *SPEC*(λT. *remove-one-annot-true-clause*** *S T*)›
unfolding *remove-one-annot-true-clause-imp-def*
apply (*refine-vcg WHILEIT-rule*[**where** *R*=‹*measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*)› **and**
I=‹*remove-one-annot-true-clause-imp-inv S*›]
remove-one-annot-true-clause-imp-inv-spec)
subgoal by *auto*
subgoal using *assms* **unfolding** *remove-one-annot-true-clause-imp-inv-def* **by** *blast*
apply (*rule remove-one-annot-true-clause-one-imp-spec*[*of* - -])
subgoal unfolding *remove-one-annot-true-clause-imp-inv-def* **by** *auto*
subgoal unfolding *remove-one-annot-true-clause-imp-inv-def* **by** *auto*
subgoal
by (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *count-decided-0-iff rev-nth is-decided-no-proped-iff*)
subgoal
by (*auto dest*!: *get-all-ann-decomposition-exists-prepend*
simp: *count-decided-0-iff rev-nth is-decided-no-proped-iff*)
subgoal unfolding *remove-one-annot-true-clause-imp-inv-def* **by** *auto*
done

lemma *remove-one-annot-true-clause-imp-spec-lev0*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
‹*get-conflict-l S* = *None*› **and**
‹*clauses-to-update-l S* = {#}› **and**
‹*count-decided* (*get-trail-l S*) = *0*›
shows ‹*remove-one-annot-true-clause-imp S* ≤ *SPEC*(λT. *remove-one-annot-true-clause*** *S T* ∧

count-decided (*get-trail-l* T) = *0* ∧ (∀ L ∈ *set* (*get-trail-l* T). *mark-of* L = *0*) ∧
length (*get-trail-l* S) = *length* (*get-trail-l* T)) ›

proof −
have H: ‹∀ *j*<*a*. *is-proped* (*rev* (*get-trail-l* *b*) ! *j*) ∧
mark-of (*rev* (*get-trail-l* *b*) ! *j*) = *0* ⟹ ¬ *a* < *length* (*get-trail-l* *b*) ⟹
∀ *x* ∈ *set* (*get-trail-l* *b*). *is-proped* *x* ∧ *mark-of* *x* = *0*› **for** *a b*
apply (*rule ballI*)
apply (*subst* (*asm*) *set-rev*[*symmetric*])
apply (*subst* (*asm*) *in-set-conv-nth*)
apply *auto*
done
have K: ‹*a* < *length* (*get-trail-l* *b*) ⟹ *is-decided* (*get-trail-l* *b* ! *a*) ⟹
count-decided (*get-trail-l* *b*) ≠ *0*› **for** *a b*
using *count-decided-0-iff nth-mem* **by** *blast*
show *?thesis*
unfolding *remove-one-annot-true-clause-imp-def*
apply (*refine-vcg WHILEIT-rule*[**where**
R=‹*measure* (λ(*i*, -::′*a twl-st-l*). *length* (*get-trail-l* S) − *i*)› **and**
I=‹*remove-one-annot-true-clause-imp-inv* S›]
remove-one-annot-true-clause-one-imp-spec)
subgoal by *auto*
subgoal using *assms* **unfolding** *remove-one-annot-true-clause-imp-inv-def* **by** *blast*
subgoal using *assms* **unfolding** *remove-one-annot-true-clause-imp-inv-def* **by** *auto*
subgoal using *assms* **by** (*auto simp*: *count-decided-0-iff is-decided-no-proped-iff*
rev-nth)
subgoal
using *assms*(*6*) **unfolding** *remove-one-annot-true-clause-imp-inv-def*
by (*auto dest*: *H K*)
subgoal using *assms* **unfolding** *remove-one-annot-true-clause-imp-inv-def*
by (*auto simp*: *rtranclp-remove-one-annot-true-clause-count-dec*)
subgoal
using *assms*(*6*) **unfolding** *remove-one-annot-true-clause-imp-inv-def*
by (*auto dest*: *H K*)
subgoal
using *assms*(*6*) **unfolding** *remove-one-annot-true-clause-imp-inv-def*
by (*auto dest*: *H K*)
done
qed

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

definition *mark-to-delete-clauses-l-inv*
:: ‹′*v twl-st-l* ⇒ *nat list* ⇒ *nat* × ′*v twl-st-l* × *nat list* ⇒ *bool*›
where
‹*mark-to-delete-clauses-l-inv* = (λS *xs0* (*i*, T, *xs*).
remove-one-annot-true-clause** S T ∧
get-trail-l S = *get-trail-l* T ∧
(∃ S'. (S, S') ∈ *twl-st-l None* ∧ *twl-struct-invs* S') ∧
twl-list-invs S ∧
get-conflict-l S = *None* ∧
clauses-to-update-l S = {#})›

definition *mark-to-delete-clauses-l-pre*
:: ‹′*v twl-st-l* ⇒ *bool*›

where
⟨*mark-to-delete-clauses-l-pre S* $\longleftrightarrow$
($\exists$ *T*. (*S*, *T*) $\in$ *twl-st-l None* $\wedge$ *twl-struct-invs T* $\wedge$ *twl-list-invs S*)⟩

definition *mark-garbage-l*:: ⟨*nat* $\Rightarrow$ *'v twl-st-l* $\Rightarrow$ *'v twl-st-l*⟩ **where**
⟨*mark-garbage-l* = (λ*C* (*M*, *N0*, *D*, *NE*, *UE*, *WS*, *Q*). (*M*, *fmdrop C N0*, *D*, *NE*, *UE*, *WS*, *Q*))⟩

definition *can-delete* **where**
⟨*can-delete S C b* = (*b* $\longrightarrow$
(*length* (*get-clauses-l S* $\propto$ *C*) = *2* $\longrightarrow$
(*Propagated* (*get-clauses-l S* $\propto$ *C* ! *0*) *C* $\notin$ *set* (*get-trail-l S*)) $\wedge$
(*Propagated* (*get-clauses-l S* $\propto$ *C* ! *1*) *C* $\notin$ *set* (*get-trail-l S*))) $\wedge$
(*length* (*get-clauses-l S* $\propto$ *C*) > *2* $\longrightarrow$
(*Propagated* (*get-clauses-l S* $\propto$ *C* ! *0*) *C* $\notin$ *set* (*get-trail-l S*))) $\wedge$
$\neg$*irred* (*get-clauses-l S*) *C*)⟩

definition *mark-to-delete-clauses-l* :: ⟨*'v twl-st-l* $\Rightarrow$ *'v twl-st-l nres*⟩ **where**
⟨*mark-to-delete-clauses-l* = (λ*S*. *do* {
ASSERT(*mark-to-delete-clauses-l-pre S*);
xs $\leftarrow$ *collect-valid-indices S*;
to-keep $\leftarrow$ *SPEC*(λ-::*nat*. *True*); — the minimum number of clauses that should be kept.
(-, *S*, -) $\leftarrow$ $WHILE_T{}^{mark\text{-}to\text{-}delete\text{-}clauses\text{-}l\text{-}inv\ S\ xs}$
(λ(*i*, *S*, *xs*). *i* < *length xs*)
(λ(*i*, *S*, *xs*). *do* {
if(*xs*!*i* $\notin$# *dom-m* (*get-clauses-l S*)) *then RETURN* (*i*, *S*, *delete-index-and-swap xs i*)
else do {
ASSERT(*0* < *length* (*get-clauses-l S*$\propto$(*xs*!*i*)));
can-del $\leftarrow$ *SPEC* (*can-delete S* (*xs*!*i*));
ASSERT(*i* < *length xs*);
if can-del
then
RETURN (*i*, *mark-garbage-l* (*xs*!*i*) *S*, *delete-index-and-swap xs i*)
else
RETURN (*i*+*1*, *S*, *xs*)
}
})
(*to-keep*, *S*, *xs*);
RETURN S
})⟩

definition *mark-to-delete-clauses-l-post* **where**
⟨*mark-to-delete-clauses-l-post S T* $\longleftrightarrow$
($\exists$ *S'*. (*S*, *S'*) $\in$ *twl-st-l None* $\wedge$ *remove-one-annot-true-clause*** *S T* $\wedge$
twl-list-invs S $\wedge$ *twl-struct-invs S'* $\wedge$ *get-conflict-l S* = *None* $\wedge$
clauses-to-update-l S = {#})⟩

lemma *mark-to-delete-clauses-l-spec*:
assumes
ST: ⟨(*S*, *S'*) $\in$ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs S*⟩ **and**
struct-invs: ⟨*twl-struct-invs S'*⟩ **and**
confl: ⟨*get-conflict-l S* = *None*⟩ **and**
upd: ⟨*clauses-to-update-l S* = {#}⟩
shows ⟨*mark-to-delete-clauses-l S* $\le$ $\Downarrow$ *Id* (*SPEC*(λ*T*. *remove-one-annot-true-clause*** *S T* $\wedge$
get-trail-l S = *get-trail-l T*))⟩

proof −

define *I* **where**
⟨*I* (*xs* :: *nat list*) ≡ (λ(*i* :: *nat*, *T*, *xs* :: *nat list*). *remove-one-annot-true-clause*** *S T*)⟩ **for** *xs*

have *mark0*: ⟨*mark-to-delete-clauses-l-pre S*⟩
using *ST list-invs struct-invs* **unfolding** *mark-to-delete-clauses-l-pre-def*
by *blast*
have *I0*: ⟨*I xs* (*l*, *S*, *xs'*)⟩
for *xs xs'* :: ⟨*nat list*⟩ **and** *l* :: *nat*
proof −
show *?thesis*
unfolding *I-def*
by *auto*
qed
have *mark-to-delete-clauses-l-inv-notin*:
⟨*mark-to-delete-clauses-l-inv S xs0* (*a*, *aa*, *delete-index-and-swap xs' a*)⟩
if
⟨*mark-to-delete-clauses-l-pre S*⟩ **and**
⟨*xs0* ∈ {*N*. *True*}⟩ **and**
⟨*mark-to-delete-clauses-l-inv S xs0 s*⟩ **and**
⟨*I xs0 s*⟩ **and**
⟨*case s of* (*i*, *S*, *xs*) ⇒ *i* < *length xs*⟩ **and**
⟨*aa'* = (*aa*, *xs'*)⟩ **and**
⟨*s* = (*a*, *aa'*)⟩ **and**
⟨*ba* ! *a* ∉# *dom-m* (*get-clauses-l aa*)⟩
for *s a aa ba xs0 aa' xs'*
proof −
show *?thesis*
using *that*
unfolding *mark-to-delete-clauses-l-inv-def*
by *auto*
qed
have *I-notin*: ⟨*I xs0* (*a*, *aa*, *delete-index-and-swap xs' a*)⟩
if
⟨*mark-to-delete-clauses-l-pre S*⟩ **and**
⟨*xs0* ∈ {*N*. *True*}⟩ **and**
⟨*mark-to-delete-clauses-l-inv S xs0 s*⟩ **and**
⟨*I xs0 s*⟩ **and**
⟨*case s of* (*i*, *S*, *xs*) ⇒ *i* < *length xs*⟩ **and**
⟨*aa'* = (*aa*, *xs'*)⟩ **and**
⟨*s* = (*a*, *aa'*)⟩ **and**
⟨*ba* ! *a* ∉# *dom-m* (*get-clauses-l aa*)⟩
for *s a aa ba xs0 aa' xs'*
proof −
show *?thesis*
using *that*
unfolding *I-def*
by *auto*
qed

have *length-ge0*: ⟨*0* < *length* (*get-clauses-l aa* ∝ (*xs* ! *a*))⟩
if
inv: ⟨*mark-to-delete-clauses-l-inv S xs0 s*⟩ **and**
I: ⟨*I xs0 s*⟩ **and**
cond: ⟨*case s of* (*i*, *S*, *xs0*) ⇒ *i* < *length xs0*⟩ **and**

st:
  ‹$aa' = (aa, xs)$›
  ‹$s = (a, aa')$› **and**
xs: ‹¬ *xs* ! *a* ∉# *dom-m* (*get-clauses-l aa*)›
for *s a b aa xs0 aa' xs*

proof −
  have
    rem: ‹*remove-one-annot-true-clause*** *S aa*›
    using *I* **unfolding** *I-def st prod.case* **by** *blast*+
  then obtain *T'* **where**
    T': ‹(*aa*, *T'*) ∈ *twl-st-l None*› **and**
    ‹*twl-struct-invs T'*›
    using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF rem list-invs confl upd ST struct-invs*] **by** *blast*
  then have ‹*Multiset.Ball* (*get-clauses T'*) *struct-wf-twl-cls*›
    unfolding *twl-struct-invs-def twl-st-inv-alt-def*
    by *fast*
  then have ‹∀ *x*∈#*ran-m* (*get-clauses-l aa*). *2* ≤ *length* (*fst x*)›
    using *xs T'* **by** (*auto simp*: *twl-st-l*)
  then show *?thesis*
    using *xs* **by** (*auto simp*: *ran-m-def*)
qed

have *mark-to-delete-clauses-l-inv-del*:
  ‹*mark-to-delete-clauses-l-inv S xs0* (*i*, *mark-garbage-l* (*xs* ! *i*) *T*, *delete-index-and-swap xs i*)› **and**
  I-del: ‹*I xs0* (*i*, *mark-garbage-l* (*xs* ! *i*) *T*, *delete-index-and-swap xs i*)›
  if
    ‹*mark-to-delete-clauses-l-pre S*› **and**
    ‹*xs0* ∈ {*N*. *True*}› **and**
    inv: ‹*mark-to-delete-clauses-l-inv S xs0 s*› **and**
    I: ‹*I xs0 s*› **and**
    i-le: ‹*case s of* (*i*, *S*, *xs*) ⇒ *i* < *length xs*› **and**
    st: ‹*sT* = (*T*, *xs*)›
      ‹*s* = (*i*, *sT*)› **and**
    in-dom: ‹¬ *xs* ! *i* ∉# *dom-m* (*get-clauses-l T*)› **and**
    ‹*0* < *length* (*get-clauses-l T* ∝ (*xs* ! *i*))› **and**
    can-del: ‹*can-delete T* (*xs* ! *i*) *b*› **and**
    ‹*i* < *length xs*› **and**
    [*simp*]: ‹*b*›
  for *x s i T b xs0 sT xs*

proof −
  obtain *M N D NE UE WS Q* **where** *S*: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
    by (*cases S*)
  obtain *M' N' D' NE' UE' WS' Q'* **where** *T*: ‹*T* = (*M'*, *N'*, *D'*, *NE'*, *UE'*, *WS'*, *Q'*)›
    by (*cases T*)
  have
    rem: ‹*remove-one-annot-true-clause*** *S T*›
    using *I* **unfolding** *I-def st prod.case* **by** *blast*+

  obtain *V* **where**
    SU: ‹*cdcl-twl-restart-l*** *S T*› **and**
    UV: ‹(*T*, *V*) ∈ *twl-st-l None*› **and**
    TV: ‹*cdcl-twl-restart*** *S' V*› **and**
    struct-invs-V: ‹*twl-struct-invs V*›
    using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF rem list-invs confl upd ST struct-invs*]

by *auto*
have *list-invs-U′*: ‹*twl-list-invs T*›
using *SU list-invs rtranclp-cdcl-twl-restart-l-list-invs* **by** *blast*

have ‹*xs* ! *i* > *0*›
apply (*rule ccontr*)
using *in-dom list-invs-U′* **unfolding** *twl-list-invs-def* **by** (*auto dest*: *multi-member-split*)
have ‹*N′* ∝ (*xs* ! *i*) ! *0* ∈ *lits-of-l M′*›
if ‹*Propagated* (*N′* ∝ (*xs* ! *i*) ! *0*) (*xs0* ! *i*) ∈ *set M′*›
using *that* **by** (*auto dest*!: *split-list*)
then have *not-annot*: ‹*Propagated Laa* (*xs* ! *i*) ∈ *set M′* ⟹ *False*› **for** *Laa*
using *is-annot-iff-annotates-first*[*OF UV list-invs-U′ struct-invs-V* ‹*xs* ! *i* > *0*›]
is-annot-no-other-true-lit[*OF UV list-invs-U′ struct-invs-V* ‹*xs* ! *i* > *0*›, *of Laa* ‹
N′ ∝ (*xs* !*i*) ! *0*›] *can-del*
trail-length-ge2[*OF UV list-invs-U′ struct-invs-V* - ‹*xs* ! *i* > *0*›, *of Laa*]
unfolding *S T can-delete-def*
by (*auto dest*: *no-dup-same-annotD*)

have *star*: ‹*remove-one-annot-true-clause T* (*mark-garbage-l* (*xs* ! *i*) *T*)›
unfolding *st T mark-garbage-l-def prod.simps*
apply (*rule remove-one-annot-true-clause.delete*)
subgoal using *in-dom i-le* **unfolding** *st prod.case T* **by** *auto*
subgoal using *can-del* **unfolding** *T can-delete-def* **by** *auto*
subgoal using *not-annot* **unfolding** *T* **by** *auto*
done
moreover have ‹*get-trail-l* (*mark-garbage-l* (*xs* ! *i*) *T*) = *get-trail-l T*›
by (*cases T*) (*auto simp*: *mark-garbage-l-def*)
ultimately show ‹*mark-to-delete-clauses-l-inv S xs0*
(*i*, *mark-garbage-l* (*xs* ! *i*) *T*, *delete-index-and-swap xs i*)›
using *inv*
unfolding *mark-to-delete-clauses-l-inv-def prod.simps st*
by *force*

show ‹*I xs0* (*i*, *mark-garbage-l* (*xs* ! *i*) *T*, *delete-index-and-swap xs i*)›
using *rem star* **unfolding** *st I-def* **by** *simp*
qed
have
mark-to-delete-clauses-l-inv-keep:
‹*mark-to-delete-clauses-l-inv S xs0* (*i* + *1*, *T*, *xs*)› **and**
I-keep: ‹*I xs0* (*i* + *1*, *T*, *xs*)›
if
‹*mark-to-delete-clauses-l-pre S*› **and**
inv: ‹*mark-to-delete-clauses-l-inv S xs0 s*› **and**
I: ‹*I xs0 s*› **and**
cond: ‹*case s of* (*i*, *S*, *xs*) ⇒ *i* < *length xs*› **and**
st: ‹*sT* = (*T*, *xs*)›
‹*s* = (*i*, *sT*)› **and**
dom: ‹¬ *xs* ! *i* ∉# *dom-m* (*get-clauses-l T*)› **and**
‹*0* < *length* (*get-clauses-l T* ∝ (*xs* ! *i*))› **and**
‹*can-delete T* (*xs* ! *i*) *b*› **and**
‹*i* < *length xs*› **and**
‹¬ *b*›
for *x s i T b xs0 sT xs*
proof −
show ‹*mark-to-delete-clauses-l-inv S xs0* (*i* + *1*, *T*, *xs*)›
using *inv*

unfolding *mark-to-delete-clauses-l-inv-def prod.simps st*
by *fast*
show ‹*I xs0* (*i* + *1*, *T*, *xs*)›
using *I* **unfolding** *I-def st prod.simps* **.**
qed

show *?thesis*
unfolding *mark-to-delete-clauses-l-def collect-valid-indices-def*
apply (*rule ASSERT-refine-left*)
apply (*rule mark0*)
apply (*subst intro-spec-iff*)
apply (*intro ballI*)
subgoal for *xs0*
apply (*refine-vcg*
WHILEIT-rule-stronger-inv[**where** *I'*=‹*I xs0*› **and**
R= ‹*measure* (λ(*i* :: *nat*, *N*, *xs0*). *Suc* (*length xs0*) − *i*)›])
subgoal by *auto*
subgoal using *list-invs confl upd ST struct-invs* **unfolding** *mark-to-delete-clauses-l-inv-def*
by (*cases S'*) *force*
subgoal by (*rule I0*)
subgoal
by (*rule mark-to-delete-clauses-l-inv-notin*)
subgoal
by (*rule I-notin*)
subgoal
by *auto*
subgoal
by (*rule length-ge0*)
subgoal
by *auto*
subgoal — delete clause
by (*rule mark-to-delete-clauses-l-inv-del*)
subgoal
by (*rule I-del*)
subgoal
by *auto*
subgoal — Keep clause
by (*rule mark-to-delete-clauses-l-inv-keep*)
subgoal
by (*rule I-keep*)
subgoal
by *auto*
subgoal
unfolding *I-def* **by** *blast*
subgoal
unfolding *mark-to-delete-clauses-l-inv-def* **by** *auto*
done
done
qed

definition *GC-clauses* :: ‹*nat clauses-l* ⇒ *nat clauses-l* ⇒ (*nat clauses-l* × (*nat* ⇒ *nat option*)) *nres*›
where
‹*GC-clauses N N'* = *do* {
xs ← *SPEC*(λ*xs*. *set-mset* (*dom-m N*) ⊆ *set xs*);
(*N*, *N'*, *m*) ← *nfoldli*
xs

($\lambda(N, N', m)$. *True*)
(λC (N, N', m).
if $C \in\#$ *dom-m* N
then do {
$C' \leftarrow$ *SPEC*($\lambda i.$ $i \notin\#$ *dom-m* $N' \wedge i \neq 0$);
RETURN (*fmdrop* C N, *fmupd* C' ($N \propto C$, *irred* N C) N', $m(C \mapsto C')$)
}
else
RETURN (N, N', m))
$(N, N', (\lambda$-. *None*));
RETURN (N', m)
}›

inductive *GC-remap*
:: ‹($'a$, $'b$) *fmap* $\times$ ($'a \Rightarrow$ $'c$ *option*) $\times$ ($'c$, $'b$) *fmap* $\Rightarrow$ ($'a$, $'b$) *fmap* $\times$ ($'a \Rightarrow$ $'c$ *option*) $\times$ ($'c$, $'b$) *fmap* $\Rightarrow$ *bool*›
where
remap-cons:
‹*GC-remap* (N, m, *new*) (*fmdrop* C N, $m(C \mapsto C')$, *fmupd* C' (*the* (*fmlookup* N C)) *new*)›
if ‹$C' \notin\#$ *dom-m new*› **and**
‹$C \in\#$ *dom-m* N› **and**
‹$C \notin$ *dom* m› **and**
‹$C' \notin$ *ran* m›

lemma *GC-remap-ran-m-old-new*:
‹*GC-remap* (*old*, m, *new*) (*old'*, m', *new'*) $\Longrightarrow$ *ran-m old* + *ran-m new* = *ran-m old'* + *ran-m new'*›
by (*induction* (*old*, m, *new*) (*old'*, m', *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*)

lemma *GC-remap-init-clss-l-old-new*:
‹*GC-remap* (*old*, m, *new*) (*old'*, m', *new'*) $\Longrightarrow$
init-clss-l old + *init-clss-l new* = *init-clss-l old'* + *init-clss-l new'*›
by (*induction* (*old*, m, *new*) (*old'*, m', *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*)

lemma *GC-remap-learned-clss-l-old-new*:
‹*GC-remap* (*old*, m, *new*) (*old'*, m', *new'*) $\Longrightarrow$
learned-clss-l old + *learned-clss-l new* = *learned-clss-l old'* + *learned-clss-l new'*›
by (*induction* (*old*, m, *new*) (*old'*, m', *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*)

lemma *GC-remap-ran-m-remap*:
‹*GC-remap* (*old*, m, *new*) (*old'*, m', *new'*) $\Longrightarrow$ $C \in\#$ *dom-m old* $\Longrightarrow$ $C \notin\#$ *dom-m old'* $\Longrightarrow$
m' $C \neq$ *None* $\wedge$
fmlookup new' (*the* (m' C)) = *fmlookup old* C›
by (*induction* (*old*, m, *new*) (*old'*, m', *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*)

lemma *GC-remap-ran-m-no-rewrite-map*:
‹*GC-remap* (*old*, m, *new*) (*old'*, m', *new'*) $\Longrightarrow$ $C \notin\#$ *dom-m old* $\Longrightarrow$ m' C = m C›
by (*induction* (*old*, m, *new*) (*old'*, m', *new'*) *rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*)

lemma *GC-remap-ran-m-no-rewrite-fmap*:

‹GC-remap (old, m, new) (old′, m′, new′) ⟹ C ∈# dom-m new ⟹
  C ∈# dom-m new′ ∧ fmlookup new C = fmlookup new′ C›
 by (*induction* (*old*, *m*, *new*) (*old′*, *m′*, *new′*) *rule*: *GC-remap.induct*)
  (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*)

lemma *rtranclp-GC-remap-init-clss-l-old-new*:
 ‹*GC-remap*** *S S′* ⟹
  init-clss-l (*fst S*) + *init-clss-l* (*snd* (*snd S*)) = *init-clss-l* (*fst S′*) + *init-clss-l* (*snd* (*snd S′*))›
 by (*induction rule*: *rtranclp-induct*)
  (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*
   dest: *GC-remap-init-clss-l-old-new*)

lemma *rtranclp-GC-remap-learned-clss-l-old-new*:
 ‹*GC-remap*** *S S′* ⟹
  learned-clss-l (*fst S*) + *learned-clss-l* (*snd* (*snd S*)) =
   learned-clss-l (*fst S′*) + *learned-clss-l* (*snd* (*snd S′*))›
 by (*induction rule*: *rtranclp-induct*)
  (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin split*: *if-splits*
   dest: *GC-remap-learned-clss-l-old-new*)

lemma *rtranclp-GC-remap-ran-m-no-rewrite-fmap*:
 ‹*GC-remap*** *S S′* ⟹ *C* ∈# *dom-m* (*snd* (*snd S*)) ⟹
  C ∈# *dom-m* (*snd* (*snd S′*)) ∧ *fmlookup* (*snd* (*snd S*)) *C* = *fmlookup* (*snd* (*snd S′*)) *C*›
 by (*induction rule*: *rtranclp-induct*)
  (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin dest*: *GC-remap-ran-m-no-rewrite-fmap*)

lemma *GC-remap-ran-m-no-rewrite*:
 ‹*GC-remap S S′* ⟹ *C* ∈# *dom-m* (*fst S*) ⟹ *C* ∈# *dom-m* (*fst S′*) ⟹
    fmlookup (*fst S*) *C* = *fmlookup* (*fst S′*) *C*›
 by (*induction rule*: *GC-remap.induct*)
  (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin distinct-mset-dom*
    distinct-mset-set-mset-remove1-mset
   dest: *GC-remap-ran-m-remap*)

lemma *GC-remap-ran-m-lookup-kept*:
 assumes
  ‹*GC-remap*** *S y*› **and**
  ‹*GC-remap y z*› **and**
  ‹*C* ∈# *dom-m* (*fst S*)› **and**
  ‹*C* ∈# *dom-m* (*fst z*)› **and**
  ‹*C* ∉# *dom-m* (*fst y*)›
 shows ‹*fmlookup* (*fst S*) *C* = *fmlookup* (*fst z*) *C*›
 using *assms* **by** (*smt GC-remap.cases fmlookup-drop fst-conv in-dom-m-lookup-iff*)

lemma *rtranclp-GC-remap-ran-m-no-rewrite*:
 ‹*GC-remap*** *S S′* ⟹ *C* ∈# *dom-m* (*fst S*) ⟹ *C* ∈# *dom-m* (*fst S′*) ⟹
  fmlookup (*fst S*) *C* = *fmlookup* (*fst S′*) *C*›
 apply (*induction rule*: *rtranclp-induct*)
 subgoal by *auto*
 subgoal for *y z*
  by (*cases* ‹*C* ∈# *dom-m* (*fst y*)›)
  (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin dest*: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
    intro: *GC-remap-ran-m-lookup-kept*)
 done

lemma *GC-remap-ran-m-no-lost*:
‹*GC-remap S S'* ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹ *C* ∈# *dom-m* (*fst S*)›
by (*induction rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin distinct-mset-dom distinct-mset-set-mset-remove1-mset*
dest: *GC-remap-ran-m-remap*)

lemma *rtranclp-GC-remap-ran-m-no-lost*:
‹*GC-remap*** *S S'* ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹ *C* ∈# *dom-m* (*fst S*)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
by (*cases* ‹*C* ∈# *dom-m* (*fst y*)›)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*
dest: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost*)
done

lemma *GC-remap-ran-m-no-new-lost*:
‹*GC-remap S S'* ⟹ *dom* (*fst* (*snd S*)) ⊆ *set-mset* (*dom-m* (*fst S*)) ⟹
dom (*fst* (*snd S'*)) ⊆ *set-mset* (*dom-m* (*fst S*))›
by (*induction rule*: *GC-remap.induct*)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin distinct-mset-dom*
distinct-mset-set-mset-remove1-mset
dest: *GC-remap-ran-m-remap*)

lemma *rtranclp-GC-remap-ran-m-no-new-lost*:
‹*GC-remap*** *S S'* ⟹ *dom* (*fst* (*snd S*)) ⊆ *set-mset* (*dom-m* (*fst S*)) ⟹
dom (*fst* (*snd S'*)) ⊆ *set-mset* (*dom-m* (*fst S*))›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
apply (*cases* ‹*C* ∈# *dom-m* (*fst y*)›)
apply (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*
dest: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost*)
apply (*smt GC-remap-ran-m-no-rewrite-map contra-subsetD domI prod.collapse rtranclp-GC-remap-ran-m-no-lost*)
apply (*smt GC-remap-ran-m-no-rewrite-map contra-subsetD domI prod.collapse rtranclp-GC-remap-ran-m-no-lost*)
done
done

lemma *rtranclp-GC-remap-map-ran*:
assumes
‹*GC-remap*** *S S'*› **and**
‹(*the* ∘∘ *fst*) (*snd S*) ‘# *mset-set* (*dom* (*fst* (*snd S*))) = *dom-m* (*snd* (*snd S*))› **and**
‹*finite* (*dom* (*fst* (*snd S*)))›
shows ‹*finite* (*dom* (*fst* (*snd S'*))) ∧
(*the* ∘∘ *fst*) (*snd S'*) ‘# *mset-set* (*dom* (*fst* (*snd S'*))) = *dom-m* (*snd* (*snd S'*))›
using *assms*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step y z*) **note** *star* = *this*(*1*) **and** *st* = *this*(*2*) **and** *IH* = *this*(*3*) **and** *H* = *this*(*4*−)
from *st*

show *?case*
proof *cases*
case (*remap-cons* C' *new* C N m)
have ‹$C \notin$ *dom* m›
using *step remap-cons* **by** *auto*
then have [*simp*]: ‹{#*the* (*if* $x = C$ *then Some* C' *else* m x). $x \in\#$ *mset-set* (*dom* m)#} =
{#*the* (m x). $x \in\#$ *mset-set* (*dom* m)#}›
apply (*auto intro*!: *image-mset-cong split*: *if-splits*)
by (*metis empty-iff finite-set-mset-mset-set local.remap-cons*(*5*) *mset-set.infinite set-mset-empty*)

show *?thesis*
using *step remap-cons*
by (*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin*
dest: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost dest*:)
qed
qed

lemma *rtranclp-GC-remap-ran-m-no-new-map*:
‹*GC-remap*** S S' $\Longrightarrow$ $C \in\#$ *dom-m* (*fst* S') $\Longrightarrow$ $C \in\#$ *dom-m* (*fst* S)›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for y z
by (*cases* ‹$C \in\#$ *dom-m* (*fst* y)›)
(*auto simp*: *ran-m-lf-fmdrop ran-m-mapsto-upd-notin dest*: *GC-remap-ran-m-remap GC-remap-ran-m-no-rewrite*
intro: *GC-remap-ran-m-lookup-kept GC-remap-ran-m-no-lost*)
done

lemma *rtranclp-GC-remap-learned-clss-lD*:
‹*GC-remap*** (N, x, m) (N', x', m') $\Longrightarrow$ *learned-clss-l* N + *learned-clss-l* m = *learned-clss-l* N' +
learned-clss-l m'›
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*: *GC-remap-learned-clss-l-old-new*)

lemma *rtranclp-GC-remap-learned-clss-l*:
‹*GC-remap*** (*x1a*, *Map.empty*, *fmempty*) (*fmempty*, m, *x1ad*) $\Longrightarrow$ *learned-clss-l x1ad* = *learned-clss-l*
x1a›
by (*auto dest*!: *rtranclp-GC-remap-learned-clss-lD*[*of* - - - - - -])

lemma *remap-cons2*:
assumes
‹$C' \notin\#$ *dom-m new*› **and**
‹$C \in\#$ *dom-m* N› **and**
‹(*the* ∘∘ *fst*) (*snd* (N, m, *new*)) ‘# *mset-set* (*dom* (*fst* (*snd* (N, m, *new*)))) =
dom-m (*snd* (*snd* (N, m, *new*)))› **and**
‹$\bigwedge x$. $x \in\#$ *dom-m* (*fst* (N, m, *new*)) $\Longrightarrow$ $x \notin$ *dom* (*fst* (*snd* (N, m, *new*)))› **and**
‹*finite* (*dom* m)›
shows
‹*GC-remap* (N, m, *new*) (*fmdrop* C N, m($C \mapsto C'$), *fmupd* C' (*the* (*fmlookup* N C)) *new*)›
proof −
have *3*: ‹$C \in$ *dom* m $\Longrightarrow$ *False*›
apply (*drule mk-disjoint-insert*)
using *assms*
apply (*auto 5 5 simp*: *ran-def*)
done

have *4*: ‹*False*› **if** C': ‹$C' \in$ *ran m*›
proof −
obtain *a* **where** *a*: ‹*a* ∈ *dom m*› **and** [*simp*]: ‹*m a* = *Some* C'›
using *that* C' **unfolding** *ran-def*
by *auto*
show *False*
using *mk-disjoint-insert*[*OF a*] *assms* **by** (*auto simp*: *union-single-eq-member*)
qed

show *?thesis*
apply (*rule remap-cons*)
apply (*rule assms*(*1*))
apply (*rule assms*(*2*))
apply (*use 3* **in** *fast*)
apply (*use 4* **in** *fast*)
done
qed

inductive-cases *GC-remapE*: ‹*GC-remap S T*›

lemma *rtranclp-GC-remap-finite-map*:
‹$GC\text{-}remap^{**}$ *S* S' ⟹ *finite* (*dom* (*fst* (*snd S*))) ⟹ *finite* (*dom* (*fst* (*snd* S')))›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z*
by (*auto elim*: *GC-remapE*)
done

lemma *rtranclp-GC-remap-old-dom-map*:
‹$GC\text{-}remap^{**}$ *R S* ⟹ (⋀*x*. *x* ∈# *dom-m* (*fst R*) ⟹ *x* ∉ *dom* (*fst* (*snd R*))) ⟹
(⋀*x*. *x* ∈# *dom-m* (*fst S*) ⟹ *x* ∉ *dom* (*fst* (*snd S*)))›
apply (*induction rule*: *rtranclp-induct*)
subgoal by *auto*
subgoal for *y z x*
by (*fastforce elim*!: *GC-remapE simp*: *distinct-mset-dom distinct-mset-set-mset-remove1-mset*)
done

lemma *remap-cons2-rtranclp*:
assumes
‹(*the* ∘∘ *fst*) (*snd R*) ‘# *mset-set* (*dom* (*fst* (*snd R*))) = *dom-m* (*snd* (*snd R*))› **and**
‹⋀*x*. *x* ∈# *dom-m* (*fst R*) ⟹ *x* ∉ *dom* (*fst* (*snd R*))› **and**
‹*finite* (*dom* (*fst* (*snd R*)))› **and**
st: ‹$GC\text{-}remap^{**}$ *R S*› **and**
C': ‹C' ∉# *dom-m* (*snd* (*snd S*))› **and**
C: ‹*C* ∈# *dom-m* (*fst S*)›
shows
‹$GC\text{-}remap^{**}$ *R* (*fmdrop C* (*fst S*), (*fst* (*snd S*))(*C* ↦ C'), *fmupd* C' (*the* (*fmlookup* (*fst S*) *C*)) (*snd* (*snd S*)))›
proof −
have
1: ‹(*the* ∘∘ *fst*) (*snd S*) ‘# *mset-set* (*dom* (*fst* (*snd S*))) = *dom-m* (*snd* (*snd S*))› **and**
2: ‹⋀*x*. *x* ∈# *dom-m* (*fst S*) ⟹ *x* ∉ *dom* (*fst* (*snd S*))› **and**
3: ‹*finite* (*dom* (*fst* (*snd S*)))›

using *assms*(*1*) *assms*(*3*) *assms*(*4*) *rtranclp-GC-remap-map-ran* **apply** *blast*
apply (*meson assms*(*2*) *assms*(*4*) *rtranclp-GC-remap-old-dom-map*)
using *assms*(*3*) *assms*(*4*) *rtranclp-GC-remap-finite-map* **by** *blast*
have *5*: ‹*GC-remap S*
(*fmdrop C* (*fst S*), (*fst* (*snd S*))(*C* ↦ *C′*), *fmupd C′* (*the* (*fmlookup* (*fst S*) *C*)) (*snd* (*snd S*)))›
using *remap-cons2*[*OF C′ C*, *of* ‹(*fst* (*snd S*))›] *1 2 3* **by** (*cases S*) *auto*
show *?thesis*
using *5 st* **by** *simp*
qed

lemma (**in** −) *fmdom-fmrestrict-set*: ‹*fmdrop xa* (*fmrestrict-set s N*) = *fmrestrict-set* (*s* − {*xa*}) *N*›
by (*rule fmap-ext-fmdom*)
(*auto simp*: *fset-fmdom-fmrestrict-set fmember.rep-eq notin-fset*)

lemma (**in** −) *GC-clauses-GC-remap*:
‹*GC-clauses N fmempty* ≤ *SPEC*(λ(*N′′*, *m*). *GC-remap*∗∗ (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N′′*) ∧
0 ∉# *dom-m N′′*)›
proof −
let *?remap* = ‹(*GC-remap*)∗∗ (*N*, λ-. *None*, *fmempty*)›
note *remap* = *remap-cons2-rtranclp*[*of* ‹(*N*, λ-. *None*, *fmempty*)›, *of* ‹(*a*, *b*, *c*)› **for** *a b c*, *simplified*]
define *I* **where**
‹*I a b* ≡ (λ(*old* :: *nat clauses-l*, *new* :: *nat clauses-l*, *m* :: *nat* ⇒ *nat option*).
?remap (*old*, *m*, *new*) ∧ *0* ∉# *dom-m new* ∧
set-mset (*dom-m old*) ⊆ *set b*)›
for *a b* :: ‹*nat list*›
have *I0*: ‹*set-mset* (*dom-m N*) ⊆ *set x* ⟹ *I* [] *x* (*N*, *fmempty*, λ-. *None*)› **for** *x*
unfolding *I-def*
by (*auto intro*!: *fmap-ext-fmdom simp*: *fset-fmdom-fmrestrict-set fmember.rep-eq*
notin-fset dom-m-def)

have *I-drop*: ‹*I* (*l1* @ [*xa*]) *l2*
(*fmdrop xa a*, *fmupd xb* (*a* ∝ *xa*, *irred a xa*) *aa*, *ba*(*xa* ↦ *xb*))›
if
‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
‹*x* = *l1* @ *xa* # *l2*› **and**
‹*I l1* (*xa* # *l2*) *σ*› **and**
‹*case σ of* (*N*, *N′*, *m*) ⇒ *True*› **and**
‹*σ* = (*a*, *b*)› **and**
‹*b* = (*aa*, *ba*)› **and**
‹*xa* ∈# *dom-m a*› **and**
‹*xb* ∉# *dom-m aa* ∧ *xb* ≠ *0*›
for *x xa l1 l2 σ a b aa ba xb*
proof −
have ‹*insert xa* (*set l2*) − *set l1* − {*xa*} = *set l2* − *insert xa* (*set l1*)›
by *auto*
have ‹*GC-remap*∗∗ (*N*, *Map.empty*, *fmempty*)
(*fmdrop xa a*, *ba*(*xa* ↦ *xb*), *fmupd xb* (*the* (*fmlookup a xa*)) *aa*)›
by (*rule remap*)
(*use that* **in** ‹*auto simp*: *I-def*›)
then show *?thesis*
using *that distinct-mset-dom*[*of a*] *distinct-mset-dom*[*of aa*] **unfolding** *I-def prod.simps*
apply (*auto dest*!: *mset-le-subtract*[*of* ‹*dom-m* -› - ‹{#*xa*#}›] *simp*: *mset-set.insert-remove*)
by (*metis Diff-empty Diff-insert0 add-mset-remove-trivial finite-set-mset*
finite-set-mset-mset-set insert-subset-eq-iff mset-set.remove set-mset-mset subseteq-remove1)
qed

have *I-notin*: ‹*I* (*l1* @ [*xa*]) *l2* (*a*, *aa*, *ba*)›
  if
    ‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
    ‹*x* = *l1* @ *xa* # *l2*› **and**
    ‹*I l1* (*xa* # *l2*) *σ*› **and**
    ‹*case σ of* (*N*, *N′*, *m*) ⇒ *True*› **and**
    ‹*σ* = (*a*, *b*)› **and**
    ‹*b* = (*aa*, *ba*)› **and**
    ‹*xa* ∉# *dom-m a*›
    for *x xa l1 l2 σ a b aa ba*
proof −
  show *?thesis*
    using *that* **unfolding** *I-def*
    by (*auto dest*!: *multi-member-split*)
qed
have *early-break*: ‹*GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *x2*, *x1*)›
   if
     ‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
     ‹*x* = *l1* @ *l2*› **and**
     ‹*I l1 l2 σ*› **and**
     ‹¬ (*case σ of* (*N*, *N′*, *m*) ⇒ *True*)› **and**
     ‹*σ* = (*a*, *b*)› **and**
     ‹*b* = (*aa*, *ba*)› **and**
     ‹(*aa*, *ba*) = (*x1*, *x2*)›
    for *x l1 l2 σ a b aa ba x1 x2*
 proof −
   show *?thesis* **using** *that* **by** *auto*
 qed

have *final-rel*: ‹*GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *x2*, *x1*)›
if
  ‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
  ‹*I x* [] *σ*› **and**
  ‹*case σ of* (*N*, *N′*, *m*) ⇒ *True*› **and**
  ‹*σ* = (*a*, *b*)› **and**
  ‹*b* = (*aa*, *ba*)› **and**
  ‹(*aa*, *ba*) = (*x1*, *x2*)›
proof −
  show ‹*GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *x2*, *x1*)›
    using *that*
    by (*auto simp*: *I-def*)
qed
have *final-rel*: ‹*GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *x2*, *x1*)› ‹*0* ∉# *dom-m x1*›
  if
    ‹*set-mset* (*dom-m N*) ⊆ *set x*› **and**
    ‹*I x* [] *σ*› **and**
    ‹*case σ of* (*N*, *N′*, *m*) ⇒ *True*› **and**
    ‹*σ* = (*a*, *b*)› **and**
    ‹*b* = (*aa*, *ba*)› **and**
    ‹(*aa*, *ba*) = (*x1*, *x2*)›
  for *x σ a b aa ba x1 x2*
  using *that*
  by (*auto simp*: *I-def*)
show *?thesis*
  unfolding *GC-clauses-def*

apply (*refine-vcg nfoldli-rule*[**where** $I = I$])
subgoal by (*rule I0*)
subgoal by (*rule I-drop*)
subgoal by (*rule I-notin*)
— Final properties:
subgoal for *x l1 l2* σ *a b aa ba x1 x2*
by (*rule early-break*)
subgoal
by (*auto simp*: *I-def*)
subgoal
by (*rule final-rel*) *assumption+*
subgoal
by (*rule final-rel*) *assumption+*
done
qed

definition *cdcl-twl-full-restart-l-prog* **where**
‹*cdcl-twl-full-restart-l-prog S = do* {
— *remove-one-annot-true-clause-imp S*
ASSERT(*mark-to-delete-clauses-l-pre S*);
T ← *mark-to-delete-clauses-l S*;
ASSERT (*mark-to-delete-clauses-l-post S T*);
RETURN T
}›

lemma *cdcl-twl-restart-l-refl*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
confl: ‹*get-conflict-l S = None*› **and**
upd: ‹*clauses-to-update-l S* = {#}›
shows ‹*cdcl-twl-restart-l S S*›
proof −
obtain *M N D NE UE WS Q* **where** *S*: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)›
by (*cases S*)
have [*simp*]: ‹*Propagated L E* ∈ *set M* ⟹ *0* < *E* ⟹*E* ∈# *dom-m N*› **for** *L E*
using *list-invs* **unfolding** *S twl-list-invs-def*
by *auto*
have [*simp*]: ‹*0* ∉# *dom-m N*›
using *list-invs* **unfolding** *S twl-list-invs-def*
by *auto*
have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *ST struct-invs* **unfolding** *twl-struct-invs-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
by (*simp add*: *twl-st twl-st-l*)
have [*intro*]: ‹*Propagated L E* ∈ *set M* ⟹
Propagated L E' ∈ *set M* ⟹ *0* < *E* ⟹ *0* < *E'* ⟹ *N* ∝ *E* = *N* ∝ *E'*› **for** *L E E'*
using *n-d* **unfolding** *S*
by (*auto dest*!: *split-list elim*!: *list-match-lel-lel*)
have [*dest*]: ‹*Propagated L 0* ∈ *set M* ⟹
Propagated L E' ∈ *set M* ⟹
0 < *E'* ⟹ *False*› **for** *E E' L*
using *n-d* **unfolding** *S*

by (*auto dest*!: *split-list elim*!: *list-match-lel-lel*)
show *?thesis*
using *confl upd*
by (*auto simp*: *S cdcl-twl-restart-l.simps valid-trail-reduction-refl*)
qed

definition *cdcl-GC-clauses-pre* :: ⟨′*v twl-st-l* ⇒ *bool*⟩ **where**
⟨*cdcl-GC-clauses-pre S* ⟷ (
∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧
twl-list-invs S ∧ *twl-struct-invs T* ∧
get-conflict-l S = *None* ∧ *clauses-to-update-l S* = {#} ∧
count-decided (*get-trail-l S*) = *0* ∧ (∀ *L*∈*set* (*get-trail-l S*). *mark-of L* = *0*)
) ⟩

definition *cdcl-GC-clauses* :: ⟨′*v twl-st-l* ⇒ ′*v twl-st-l nres*⟩ **where**
⟨*cdcl-GC-clauses* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). *do* {
ASSERT(*cdcl-GC-clauses-pre* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*));
b ← *SPEC*(λ*b*. *True*);
if b then do {
(*N*′, -) ← *SPEC* (λ(*N*′′, *m*). *GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N*′′) ∧
0 ∉# *dom-m N*′′);
RETURN (*M*, *N*′, *D*, *NE*, *UE*, *WS*, *Q*)
}
else RETURN (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)})⟩

lemma *cdcl-GC-clauses-cdcl-twl-restart-l*:
assumes
ST: ⟨(*S*, *T*) ∈ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs S*⟩ **and**
struct-invs: ⟨*twl-struct-invs T*⟩ **and**
confl: ⟨*get-conflict-l S* = *None*⟩ **and**
upd: ⟨*clauses-to-update-l S* = {#}⟩ **and**
count-dec: ⟨*count-decided* (*get-trail-l S*) = *0*⟩ **and**
mark: ⟨∀ *L*∈*set* (*get-trail-l S*). *mark-of L* = *0*⟩
shows ⟨*cdcl-GC-clauses S* ≤ *SPEC* (λ*T*. *cdcl-twl-restart-l S T* ∧
get-trail-l S = *get-trail-l T*)⟩
proof −
show *?thesis*
unfolding *cdcl-GC-clauses-def*
apply *refine-vcg*
subgoal using *assms* **unfolding** *cdcl-GC-clauses-pre-def* **by** *blast*
subgoal using *confl upd count-dec mark* **by** (*auto simp*: *cdcl-twl-restart-l.simps*
valid-trail-reduction-refl
dest: *rtranclp-GC-remap-init-clss-l-old-new rtranclp-GC-remap-learned-clss-l-old-new*)
subgoal
using *cdcl-twl-restart-l-refl*[*OF assms*(*1*−*5*)] **by** *simp*
subgoal
using *cdcl-twl-restart-l-refl*[*OF assms*(*1*−*5*)] **by** *simp*
subgoal
using *cdcl-twl-restart-l-refl*[*OF assms*(*1*−*5*)] **by** *simp*
done
qed

lemma *remove-one-annot-true-clause-cdcl-twl-restart-l-spec*:
assumes
ST: ⟨(*S*, *T*) ∈ *twl-st-l None*⟩ **and**

list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
confl: ‹*get-conflict-l S = None*› **and**
upd: ‹*clauses-to-update-l S* = {#}›
shows ‹*SPEC*(*remove-one-annot-true-clause*** *S*) ≤ *SPEC*(*cdcl-twl-restart-l S*)›
proof −
have ‹*cdcl-twl-restart-l S U′*›
if *rem*: ‹*remove-one-annot-true-clause*** *S U′*› **for** *U′*
proof −
have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *ST struct-invs* **unfolding** *twl-struct-invs-def*
$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def*
by (*simp add*: *twl-st twl-st-l*)
have ‹*cdcl-twl-restart-l*** *S U′*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*of S U′ T*, *OF rem list-invs confl upd ST struct-invs*]
apply −
apply *normalize-goal*+
by *auto*
then show ‹*cdcl-twl-restart-l S U′*›
using *cdcl-twl-restart-l-refl*[*OF ST list-invs struct-invs confl upd*]
tranclp-cdcl-twl-restart-l-cdcl-is-cdcl-twl-restart-l[*of S U′*, *OF - n-d*]
by (*meson rtranclp-into-tranclp2*)
qed
then show *?thesis*
by *auto*
qed

definition (**in** −) *cdcl-twl-local-restart-l-spec* :: ‹*′v twl-st-l* ⇒ *′v twl-st-l nres*› **where**
‹*cdcl-twl-local-restart-l-spec* = (λ(*M*, *N*, *D*, *NE*, *UE*, *W*, *Q*). *do* {
(*M*, *Q*) ← *SPEC*(λ(*M′*, *Q′*). (∃ *K M2*. (*Decided K* # *M′*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q′ = {#}) ∨ (*M′* = *M* ∧ *Q′* = *Q*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *W*, *Q*)
})›

definition *cdcl-twl-restart-l-prog* **where**
‹*cdcl-twl-restart-l-prog S* = *do* {
b ← *SPEC*(λ-. *True*);
if b then cdcl-twl-local-restart-l-spec S else cdcl-twl-full-restart-l-prog S
}›

lemma *cdcl-twl-local-restart-l-spec-cdcl-twl-restart-l*:
assumes *inv*: ‹*restart-abs-l-pre S False*›
shows ‹*cdcl-twl-local-restart-l-spec S* ≤ *SPEC* (*cdcl-twl-restart-l S*)›
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S = None*›
using *inv* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*

apply − **apply** *normalize-goal+*
by (*auto simp*: *twl-st-l twl-st*)
have *S*: ‹*S* = (*get-trail-l S*, *snd S*)›
by (*cases S*) *auto*

obtain *M N D NE UE W Q* **where**
S: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *W*, *Q*)›
by (*cases S*)
have *restart*: ‹*cdcl-twl-restart-l S* (*M'*, *N*, *D*, *NE*, *UE*, *W*, *Q'*)›
if *decomp'*: ‹($\exists$ *K M2*. (*Decided K* # *M'*, *M2*) $\in$ *set* (*get-all-ann-decomposition M*) $\wedge$
Q' = {#}) $\vee$ (*M'* = *M* $\wedge$ *Q'* = *Q*)›
for *M' K M2 Q'*
proof −
consider
(*nope*) ‹*M* = *M'*› **and** ‹*Q'* = *Q*› |
(*decomp*) *K M2* **where** ‹(*Decided K* # *M'*, *M2*) $\in$ *set* (*get-all-ann-decomposition M*)› **and**
‹*Q'* = {#}›
using *decomp'* **by** *auto*
then show *?thesis*
proof *cases*
case [*simp*]: *nope*
have *valid*: ‹*valid-trail-reduction M M'*›
by (*use valid-trail-reduction.keep-red*[*of M'*] **in** ‹*auto simp*: *S*›)
have
S1: ‹*S* = (*M*, *N*, *None*, *NE*, *UE*, {#}, *Q*)› **and**
S2 : ‹(*M'*, *N*, *D*, *NE*, *UE*, *W*, *Q'*) = (*M'*, *N*, *None*, *NE* + *mset* '# {#}, *UE* + *mset* '# {#},
{#}, *Q*)›
using *confl upd* **unfolding** *S*
by *auto*
have
‹$\forall$ *C*$\in$#*clauses-to-update-l S*. *C* $\in$# *dom-m* (*get-clauses-l S*)› **and**
dom0: ‹*0* $\notin$# *dom-m* (*get-clauses-l S*)› **and**
annot: ‹$\bigwedge$*L C*. *Propagated L C* $\in$ *set* (*get-trail-l S*) $\Longrightarrow$
0 < *C* $\Longrightarrow$
(*C* $\in$# *dom-m* (*get-clauses-l S*) $\wedge$
L $\in$ *set* (*watched-l* (*get-clauses-l S* $\propto$ *C*)) $\wedge$
(*length* (*get-clauses-l S* $\propto$ *C*) > *2* $\longrightarrow$ *L* = *get-clauses-l S* $\propto$ *C* ! *0*))› **and**
‹*distinct-mset* (*clauses-to-update-l S*)›
using *list-invs* **unfolding** *twl-list-invs-def S*[*symmetric*] **by** *auto*
have *n-d*: ‹*no-dup M*›
using *struct-invs ST* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *twl-st-l twl-st S*)
have *propa-MM*: ‹*Propagated L E* $\in$ *set M* $\Longrightarrow$ *Propagated L E'* $\in$ *set M'* $\Longrightarrow$ *E*=*E'*› **for** *L E E'*
using *n-d*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E'*› *M*]
dest: *no-dup-same-annotD*
elim!: *list-match-lel-lel*)

show *?thesis*
unfolding *S*[*symmetric*] *S1 S2*
apply (*rule cdcl-twl-restart-l.intros*)
subgoal by (*rule valid*)
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal using *propa-MM annot* **unfolding** *S* **by** *fastforce*
subgoal using *propa-MM annot* **unfolding** *S* **by** *fastforce*
subgoal using *propa-MM annot* **unfolding** *S* **by** *fastforce*
subgoal using *dom0* **unfolding** *S* **by** *auto*
subgoal by *auto*
done
next
case *decomp* **note** $decomp = this(1)$ **and** $Q = this(2)$
have *valid*: ‹*valid-trail-reduction M M'*›
by (*use valid-trail-reduction.backtrack-red*[*OF decomp*, *of M'*] **in** ‹*auto simp*: *S*›)
have
‹$\forall C \in \#$*clauses-to-update-l S*. $C \in \#$ *dom-m* (*get-clauses-l S*)› **and**
dom0: ‹$0 \notin \#$ *dom-m* (*get-clauses-l S*)› **and**
annot: ‹$\bigwedge L\ C$. *Propagated L C* $\in$ *set* (*get-trail-l S*) $\Longrightarrow$
$0 < C \Longrightarrow$
($C \in \#$ *dom-m* (*get-clauses-l S*) $\wedge$
$L \in$ *set* (*watched-l* (*get-clauses-l S* $\propto$ *C*)) $\wedge$
(*length* (*get-clauses-l S* $\propto$ *C*) $> 2 \longrightarrow L =$ *get-clauses-l S* $\propto$ *C* ! *0*))› **and**
‹*distinct-mset* (*clauses-to-update-l S*)›
using *list-invs* **unfolding** *twl-list-invs-def S*[*symmetric*] **by** *auto*
obtain *M3* **where** *M*: ‹$M = M3$ @ *Decided K* # *M'*›
using *decomp* **by** *auto*
have *n-d*: ‹*no-dup M*›
using *struct-invs ST* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*auto simp*: *twl-st-l twl-st S*)
have
S1: ‹*S* = (*M*, *N*, *None*, *NE*, *UE*, {#}, *Q*)› **and**
S2 : ‹(*M'*, *N*, *D*, *NE*, *UE*, *W*, *Q'*) = (*M'*, *N*, *None*, *NE* + *mset* ‘# {#}, *UE* + *mset* ‘# {#}, {#}, {#})›
using *confl upd* **unfolding** *S Q*
by *auto*
have *propa-MM*: ‹*Propagated L E* $\in$ *set M* $\Longrightarrow$ *Propagated L E'* $\in$ *set M'* $\Longrightarrow$ *E=E'*› **for** *L E E'*
using *n-d* **unfolding** *M*
by (*auto simp*: *S twl-list-invs-def*
dest!: *split-list*[*of* ‹*Propagated L E*› *M*]
split-list[*of* ‹*Propagated L E'*› *M*]
dest: *no-dup-same-annotD*
elim!: *list-match-lel-lel*)

show *?thesis*
unfolding *S*[*symmetric*] *S1 S2*
apply (*rule cdcl-twl-restart-l.intros*)
subgoal by (*rule valid*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *propa-MM annot* **unfolding** *S* **by** *fastforce*
subgoal using *propa-MM annot* **unfolding** *S* **by** *fastforce*
subgoal using *propa-MM annot* **unfolding** *S* **by** *fastforce*
subgoal using *dom0* **unfolding** *S* **by** *auto*
subgoal using *decomp* **unfolding** *S* **by** *auto*
done
qed
qed
show *?thesis*

apply (*subst S*)
unfolding *cdcl-twl-local-restart-l-spec-def prod.case RES-RETURN-RES2 less-eq-nres.simps uncurry-def*
apply *clarify*
apply (*rule restart*)
apply *assumption*
done
qed

definition (**in** −) *cdcl-twl-local-restart-l-spec0* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*cdcl-twl-local-restart-l-spec0* = (λ(*M*, *N*, *D*, *NE*, *UE*, *W*, *Q*). *do* {
(*M*, *Q*) ← *SPEC*(λ(*M'*, *Q'*). (∃ *K M2*. (*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q' = {#} ∧ *count-decided M'* = *0*) ∨ (*M'* = *M* ∧ *Q'* = *Q* ∧ *count-decided M'* = *0*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *W*, *Q*)
})›

lemma *cdcl-twl-local-restart-l-spec0-cdcl-twl-local-restart-l-spec*:
‹*cdcl-twl-local-restart-l-spec0 S* ≤ ⇓{(*S*, *S'*). *S* = *S'* ∧ *count-decided* (*get-trail-l S*) = *0*}
(*cdcl-twl-local-restart-l-spec S*)›
unfolding *cdcl-twl-local-restart-l-spec0-def*
cdcl-twl-local-restart-l-spec-def
by *refine-vcg* (*auto simp*: *RES-RETURN-RES2*)

definition *cdcl-twl-full-restart-l-GC-prog-pre*
:: ‹*'v twl-st-l* ⇒ *bool*›
where
‹*cdcl-twl-full-restart-l-GC-prog-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧ *twl-list-invs S* ∧
get-conflict T = *None*)›

definition *cdcl-twl-full-restart-l-GC-prog* **where**
‹*cdcl-twl-full-restart-l-GC-prog S* = *do* {
ASSERT(*cdcl-twl-full-restart-l-GC-prog-pre S*);
S' ← *cdcl-twl-local-restart-l-spec0 S*;
T ← *remove-one-annot-true-clause-imp S'*;
ASSERT(*mark-to-delete-clauses-l-pre T*);
U ← *mark-to-delete-clauses-l T*;
V ← *cdcl-GC-clauses U*;
ASSERT(*cdcl-twl-restart-l S V*);
RETURN V
}›

lemma *cdcl-twl-full-restart-l-prog-spec*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}›
shows ‹*cdcl-twl-full-restart-l-prog S* ≤ ⇓ *Id* (*SPEC*(*remove-one-annot-true-clause*** *S*))›
proof −
have *mark-to-delete-clauses-l*:
‹*mark-to-delete-clauses-l x* ≤ *SPEC* (λ*T*. *ASSERT* (*mark-to-delete-clauses-l-post U T*) ⤜
(λ-. *RETURN T*)

$\leq$ *SPEC* (*remove-one-annot-true-clause*** *U*))›
if
Ux: ‹(*x*, *U*) ∈ *Id*› **and**
U: ‹*U* ∈ *Collect* (*remove-one-annot-true-clause*** *S*)›
for *x U*
proof −
from *U* **have** *SU*: ‹*remove-one-annot-true-clause*** *S U*› **by** *simp*
have *x*: ‹*x* = *U*›
using *Ux* **by** *auto*
obtain *V* **where**
SU': ‹*cdcl-twl-restart-l*** *S U*› **and**
UV: ‹(*U*, *V*) ∈ *twl-st-l None*› **and**
TV: ‹*cdcl-twl-restart*** *T V*› **and**
struct-invs-V: ‹*twl-struct-invs V*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF SU list-invs confl upd ST struct-invs*]
by *auto*
have
confl-U: ‹*get-conflict-l U* = *None*› **and**
upd-U: ‹*clauses-to-update-l U* = {#}›
using *rtranclp-remove-one-annot-true-clause-get-conflict-l*[*OF SU*]
rtranclp-remove-one-annot-true-clause-clauses-to-update-l[*OF SU*] *confl upd*
by *auto*
have *list-U*: ‹*twl-list-invs U*›
using *SU' list-invs rtranclp-cdcl-twl-restart-l-list-invs* **by** *blast*
have [*simp*]:
‹*remove-one-annot-true-clause*** *U V'* $\Longrightarrow$ *mark-to-delete-clauses-l-post U V'*› **for** *V'*
unfolding *mark-to-delete-clauses-l-post-def*
using *UV struct-invs-V list-U confl-U upd-U*
by *blast*
show *?thesis*
unfolding *x*
by (*rule mark-to-delete-clauses-l-spec*[*OF UV list-U struct-invs-V confl-U upd-U*, *THEN order-trans*])
(*auto intro*: *RES-refine*)
qed
have *1*: ‹*SPEC* (*remove-one-annot-true-clause*** *S*) = *do* {
T ← *SPEC* (*remove-one-annot-true-clause*** *S*);
SPEC (*remove-one-annot-true-clause*** *T*)
}›
by (*auto simp*: *RES-RES-RETURN-RES*)
have *H*: ‹*mark-to-delete-clauses-l-pre T*›
if
‹(*T*, *U*) ∈ *Id*› **and**
‹*U* ∈ *Collect* (*remove-one-annot-true-clause*** *S*)›
for *T U*
proof −
show *?thesis*
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*of S U*, *OF - list-invs confl upd ST struct-invs*] *that list-invs*
unfolding *mark-to-delete-clauses-l-pre-def*
by (*force intro*: *rtranclp-cdcl-twl-restart-l-list-invs*)
qed
show *?thesis*
unfolding *cdcl-twl-full-restart-l-prog-def*
apply (*refine-vcg mark-to-delete-clauses-l*

)
subgoal
using *assms*
unfolding *mark-to-delete-clauses-l-pre-def*
by *blast*
subgoal by *auto*
subgoal by (*auto simp*: *assert-bind-spec-conv*)
done
qed

lemma *valid-trail-reduction-count-dec-ge*:
⟨*valid-trail-reduction M M′* ⟹ *count-decided M* ≥ *count-decided M′*⟩
apply (*induction rule*: *valid-trail-reduction.induct*)
subgoal for *K M M′*
using *trail-renumber-count-dec*
by (*fastforce simp*: *dest*!: *get-all-ann-decomposition-exists-prepend*)
subgoal by (*auto dest*: *trail-renumber-count-dec*)
done

lemma *cdcl-twl-restart-l-count-dec-ge*:
⟨*cdcl-twl-restart-l S T* ⟹ *count-decided* (*get-trail-l S*) ≥ *count-decided* (*get-trail-l T*)⟩
by (*induction rule*: *cdcl-twl-restart-l.induct*)
(*auto dest*!: *valid-trail-reduction-count-dec-ge*)

lemma *valid-trail-reduction-lit-of-nth*:
⟨*valid-trail-reduction M M′* ⟹ *length M* = *length M′* ⟹ *i* < *length M* ⟹
lit-of (*M* ! *i*) = *lit-of* (*M′* ! *i*)⟩
apply (*induction rule*: *valid-trail-reduction.induct*)
subgoal premises *p* **for** *K M″ M2*
using *arg-cong*[*OF p*(*2*), *of length*] *p*(*1*) *arg-cong*[*OF p*(*2*), *of* ⟨λ*xs*. *xs* ! *i*⟩] *p*(*4*)
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
subgoal premises *p*
using *arg-cong*[*OF p*(*1*), *of length*] *p*(*3*) *arg-cong*[*OF p*(*1*), *of* ⟨λ*xs*. *xs* ! *i*⟩] *p*(*4*)
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
done

lemma *cdcl-twl-restart-l-lit-of-nth*:
⟨*cdcl-twl-restart-l S U* ⟹ *i* < *length* (*get-trail-l U*) ⟹ *is-proped* (*get-trail-l U* ! *i*) ⟹
length (*get-trail-l S*) = *length* (*get-trail-l U*) ⟹
lit-of (*get-trail-l S* ! *i*) = *lit-of* (*get-trail-l U* ! *i*)⟩
apply (*induction rule*: *cdcl-twl-restart-l.induct*)
subgoal for *M M′ N N′ NE′ UE′ NE UE Q Q′*
using *valid-trail-reduction-length-leD*[*of M M′*]
valid-trail-reduction-lit-of-nth[*of M M′ i*]
by *auto*
done

lemma *valid-trail-reduction-is-decided-nth*:
⟨*valid-trail-reduction M M′* ⟹ *length M* = *length M′* ⟹ *i* < *length M* ⟹
is-decided (*M* ! *i*) = *is-decided* (*M′* ! *i*)⟩
apply (*induction rule*: *valid-trail-reduction.induct*)
subgoal premises *p* **for** *K M″ M2*
using *arg-cong*[*OF p*(*2*), *of length*] *p*(*1*) *arg-cong*[*OF p*(*3*), *of* ⟨λ*xs*. *xs* ! *i*⟩] *p*(*4*)
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*

dest!: *get-all-ann-decomposition-exists-prepend*)
subgoal premises *p*
using *arg-cong*[*OF p*(*1*), *of length*] *p*(*3*) *arg-cong*[*OF p*(*2*), *of* ‹λ*xs*. *xs* ! *i*›] *p*(*4*)
by (*auto simp*: *nth-map nth-append nth-Cons split*: *if-splits*
dest!: *get-all-ann-decomposition-exists-prepend*)
done

lemma *cdcl-twl-restart-l-mark-of-same-or-0*:
‹*cdcl-twl-restart-l S U* ⟹ *i* < *length* (*get-trail-l U*) ⟹ *is-proped* (*get-trail-l U* ! *i*) ⟹
length (*get-trail-l S*) = *length* (*get-trail-l U*) ⟹
(*mark-of* (*get-trail-l U* ! *i*) > *0* ⟹ *mark-of* (*get-trail-l S* ! *i*) > *0* ⟹
mset (*get-clauses-l S* ∝ *mark-of* (*get-trail-l S* ! *i*))
= *mset* (*get-clauses-l U* ∝ *mark-of* (*get-trail-l U* ! *i*)) ⟹ *P*) ⟹
(*mark-of* (*get-trail-l U* ! *i*) = *0* ⟹ *P*) ⟹ *P*›
apply (*induction rule*: *cdcl-twl-restart-l.induct*)
subgoal for *M M′ N N′ NE′ UE′ NE UE Q Q′*
using *valid-trail-reduction-length-leD*[*of M M′*]
valid-trail-reduction-lit-of-nth[*of M M′ i*]
valid-trail-reduction-is-decided-nth[*of M M′ i*]
split-list[*of* ‹*M* ! *i*› *M*, *OF nth-mem*] *split-list*[*of* ‹*M′* ! *i*› *M′*, *OF nth-mem*]
by (*cases* ‹*M* ! *i*›; *cases* ‹*M′* ! *i*›)
(*force simp*: *all-conj-distrib*)+
done

lemma *cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l*:
assumes
ST: ‹(*S*, *S′*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs S′*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs S′*›
shows ‹*cdcl-twl-full-restart-l-GC-prog S* ≤ ⇓ *Id* (*SPEC* (λ*T*. *cdcl-twl-restart-l S T*))›
proof −
let *?f* = ‹(λ*S T*. *cdcl-twl-restart-l S T*)›
let *?f1* = ‹λ*S S′*. *?f S S′* ∧ *count-decided* (*get-trail-l S′*) = *0*›
let *?f2* = ‹λ*S S′*. *?f1 S S′* ∧ (∀*L* ∈ *set* (*get-trail-l S′*). *mark-of L* = *0*) ∧
length (*get-trail-l S*) = *length* (*get-trail-l S′*)›
have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *struct-invs ST* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def*
by (*simp add*: *twl-st*)
then have *alt-def*: ‹*SPEC* (*?f S*) ≥ *do* {
S′ ← *SPEC* (λ*S′*. *?f1 S S′*);
T ← *SPEC* (*?f2 S′*);
U ← *SPEC* (*?f2 T*);
V ← *SPEC* (*?f2 U*);
RETURN V
}›
using *cdcl-twl-restart-l-refl*[*OF assms*(*1*−*4*)]
apply (*auto simp*: *RES-RES-RETURN-RES*)
by (*meson cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*)
have *1*: ‹*remove-one-annot-true-clause-imp T* ≤ *SPEC* (*?f2 U*)›
if
‹(*T*, *U*) ∈ *Id*› **and**

⟨*U* ∈ *Collect* (λ*S'*. *?f1 S S'*)⟩
for *T U*
proof −
have ⟨*T* = *U*⟩ **and** ⟨*?f S T*⟩ **and** *count-0*: ⟨*count-decided* (*get-trail-l T*) = *0*⟩
using *that* **by** *auto*
have *confl*: ⟨*get-conflict-l T* = *None*⟩
using ⟨*?f S T*⟩
by (*auto simp*: *cdcl-twl-restart-l.simps*)
obtain *T'* **where**
TT': ⟨(*T*, *T'*) ∈ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs T*⟩ **and**
struct-invs: ⟨*twl-struct-invs T'*⟩ **and**
clss-upd: ⟨*clauses-to-update-l T* = {#}⟩ **and**
⟨*cdcl-twl-restart S' T'*⟩
using *cdcl-twl-restart-l-invs*[*OF assms*(*1*−*3*) ⟨*?f S T*⟩] **by** *blast*
show *?thesis*
unfolding ⟨*T* = *U*⟩[*symmetric*]
by (*rule remove-one-annot-true-clause-imp-spec-lev0*[*OF TT' list-invs struct-invs confl clss-upd*, *THEN order-trans*])
(*use count-0 remove-one-annot-true-clause-cdcl-twl-restart-l-spec*[*OF TT' list-invs struct-invs confl clss-upd*] *n-d* ⟨*cdcl-twl-restart-l S T*⟩
remove-one-annot-true-clause-map-mark-of-same-or-0[*of T*] **in**
⟨*auto dest*: *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
simp: *rtranclp-remove-one-annot-true-clause-count-dec*⟩)
qed

have *mark-to-delete-clauses-l-pre*: ⟨*mark-to-delete-clauses-l-pre U*⟩
if
⟨(*T*, *T'*) ∈ *Id*⟩ **and**
⟨*T'* ∈ *Collect* (*?f1 S*)⟩ **and**
⟨(*U*, *U'*) ∈ *Id*⟩ **and**
⟨*U'* ∈ *Collect* (*?f2 T'*)⟩
for *T T' U U'*
proof −
have ⟨*T* = *T'*⟩ ⟨*U* = *U'*⟩ **and** ⟨*?f T U*⟩ **and** ⟨*?f S T*⟩
using *that* **by** *auto*
then have ⟨*?f S U*⟩
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
by *blast*
have *confl*: ⟨*get-conflict-l U* = *None*⟩
using ⟨*?f T U*⟩
by (*auto simp*: *cdcl-twl-restart-l.simps*)
obtain *U'* **where**
TT': ⟨(*U*, *U'*) ∈ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs U*⟩ **and**
struct-invs: ⟨*twl-struct-invs U'*⟩ **and**
clss-upd: ⟨*clauses-to-update-l U* = {#}⟩ **and**
⟨*cdcl-twl-restart S' U'*⟩
using *cdcl-twl-restart-l-invs*[*OF assms*(*1*−*3*) ⟨*?f S U*⟩] **by** *blast*
then show *?thesis*
unfolding *mark-to-delete-clauses-l-pre-def*
by *blast*
qed
have *2*: ⟨*mark-to-delete-clauses-l U* ≤ *SPEC* (*?f2 U'*)⟩
if
⟨(*T*, *T'*) ∈ *Id*⟩ **and**

⟨$T' \in$ *Collect* (*?f1 S*)⟩ **and**
UU': ⟨(U, U') $\in$ *Id*⟩ **and**
U: ⟨$U' \in$ *Collect* (*?f2 T'*)⟩ **and**
pre: ⟨*mark-to-delete-clauses-l-pre U*⟩
for *T T' U U'*
proof −
have ⟨$T = T'$⟩ ⟨$U = U'$⟩ **and** ⟨*?f T U*⟩ **and** ⟨*?f S T*⟩
using *that* **by** *auto*
then have *SU*: ⟨*?f S U*⟩
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
by *blast*

obtain *V* **where**
TV: ⟨(U, V) $\in$ *twl-st-l None*⟩ **and**
struct: ⟨*twl-struct-invs V*⟩ **and**
list-invs: ⟨*twl-list-invs U*⟩
using *pre* **unfolding** *mark-to-delete-clauses-l-pre-def*
by *auto*
have *confl*: ⟨*get-conflict-l U* = *None*⟩ **and** *upd*: ⟨*clauses-to-update-l U* = {#}⟩ **and** *UU*[*simp*]: ⟨U' = U⟩
using *U UU'*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
show *?thesis*
by (*rule mark-to-delete-clauses-l-spec*[*OF TV list-invs struct confl upd*, *THEN order-trans*], *subst Down-id-eq*)
(*use remove-one-annot-true-clause-cdcl-twl-restart-l-spec*[*OF TV list-invs struct confl upd*] *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*[*OF - - n-d*, *of T*] *that* *ST* **in** *auto*)
qed
have *3*: ⟨*cdcl-GC-clauses V* $\leq$ *SPEC* (*?f2 V'*)⟩
if
⟨(T, T') $\in$ *Id*⟩ **and**
⟨$T' \in$ *Collect* (*?f1 S*)⟩ **and**
⟨(U, U') $\in$ *Id*⟩ **and**
⟨$U' \in$ *Collect* (*?f2 T'*)⟩ **and**
⟨*mark-to-delete-clauses-l-pre U*⟩ **and**
⟨(V, V') $\in$ *Id*⟩ **and**
⟨$V' \in$ *Collect* (*?f2 U'*)⟩
for *T T' U U' V V'*
proof −
have *eq*: ⟨$U' = U$⟩
using *that* **by** *auto*
have *st*: ⟨$T = T'$⟩ ⟨$U = U'$⟩ ⟨$V = V'$⟩ **and** ⟨*?f S T*⟩ **and** ⟨*?f T U*⟩ **and** ⟨*?f U V*⟩ **and**
le-UV: ⟨*length* (*get-trail-l U*) = *length* (*get-trail-l V*)⟩ **and**
mark0: ⟨$\forall$ *L*$\in$*set* (*get-trail-l V'*). *mark-of L* = *0*⟩ **and**
count-dec: ⟨*count-decided* (*get-trail-l V'*) = *0*⟩
using *that* **by** *auto*
then have ⟨*?f S V*⟩
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*
by *blast*
have *mark*: ⟨*mark-of* (*get-trail-l V* ! *i*) = *0*⟩ **if** ⟨$i <$ *length* (*get-trail-l V*)⟩ **for** *i*
using *that*
by (*elim cdcl-twl-restart-l-mark-of-same-or-0*[*OF* ⟨*?f U V*⟩, *of i*])
(*use st that le-UV count-dec mark0* **in**
⟨*auto simp*: *count-decided-0-iff is-decided-no-proped-iff*⟩)
then have *count-dec*: ⟨*count-decided* (*get-trail-l V'*) = *0*⟩ **and**

mark: ‹⋀*L*. *L* ∈ *set* (*get-trail-l V'*) ⟹ *mark-of L* = *0*›
using *cdcl-twl-restart-l-count-dec-ge*[*OF* ‹*?f U V*›] *that*
by *auto*
obtain *W* **where**
UV: ‹(*V*, *W*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs V*› **and**
clss: ‹*clauses-to-update-l V* = {#}› **and**
‹*cdcl-twl-restart S' W*› **and**
struct: ‹*twl-struct-invs W*›
using *cdcl-twl-restart-l-invs*[*OF assms*(*1*,*2*,*3*) ‹*?f S V*›] **unfolding** *eq* **by** *blast*
have *confl*: ‹*get-conflict-l V* = *None*›
using ‹*?f S V*› **unfolding** *eq*
by (*auto simp*: *cdcl-twl-restart-l.simps*)
show *?thesis*
unfolding *eq*
by (*rule cdcl-GC-clauses-cdcl-twl-restart-l*[*OF UV list-invs struct confl clss*, *THEN order-trans*])
(*use count-dec cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*[*OF - - n-d*, *of U'*]
‹*?f S V*› *eq mark* **in** ‹*auto simp*: ‹*V* = *V'*››)
qed
have *cdcl-twl-restart-l*: ‹*cdcl-twl-restart-l S W*›
if
‹(*T*, *T'*) ∈ *Id*› **and**
‹*T'* ∈ *Collect* (*?f1 S*)› **and**
‹(*U*, *U'*) ∈ *Id*› **and**
‹*U'* ∈ *Collect* (*?f2 T'*)› **and**
‹*mark-to-delete-clauses-l-pre U*› **and**
‹(*V*, *V'*) ∈ *Id*› **and**
‹*V'* ∈ *Collect* (*?f2 U'*)› **and**
‹(*W*, *W'*) ∈ *Id*› **and**
‹*W'* ∈ *Collect* (*?f2 V'*)›
for *T T' U U' V V' W W'*
using *n-d cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*[*of S T U*]
cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[*of S U V*]
cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l[*of S V W*] *that*
by *fast*

show *?thesis*
unfolding *cdcl-twl-full-restart-l-GC-prog-def*
apply (*rule order-trans*)
prefer *2* **apply** (*rule ref-two-step'*)
apply (*rule alt-def*)
apply *refine-rcg*
subgoal
using *assms* **unfolding** *cdcl-twl-full-restart-l-GC-prog-pre-def*
by *fastforce*
subgoal
by (*rule cdcl-twl-local-restart-l-spec0-cdcl-twl-local-restart-l-spec*[*THEN order-trans*],
subst (*3*) *Down-id-eq*[*symmetric*],
rule order-trans,
rule ref-two-step',
rule cdcl-twl-local-restart-l-spec-cdcl-twl-restart-l,
unfold restart-abs-l-pre-def, *rule exI*[*of - S'*])
(*use assms* **in** ‹*auto simp*: *restart-prog-pre-def conc-fun-RES*›)
subgoal
by (*rule 1*)
subgoal for *T T' U U'*

by (*rule mark-to-delete-clauses-l-pre*)
subgoal for $T\ T'\ U\ U'$
by (*rule 2*)
subgoal for $T\ T'\ U\ U'\ V\ V'$
by (*rule 3*)
subgoal for $T\ T'\ U\ U'\ V\ V'\ W\ W'$
by (*rule cdcl-twl-restart-l*)
done
qed

context *twl-restart-ops*
begin

definition *restart-prog-l*
:: *'v twl-st-l* ⇒ *nat* ⇒ *bool* ⇒ (*'v twl-st-l* × *nat*) *nres*
where
‹*restart-prog-l S n brk* = *do* {
ASSERT(*restart-abs-l-pre S brk*);
b ← *restart-required-l S n*;
b2 ← *SPEC*(λ-. *True*);
if b2 ∧ *b* ∧ ¬*brk then do* {
T ← *cdcl-twl-full-restart-l-GC-prog S*;
RETURN (*T*, *n* + *1*)
}
else if b ∧ ¬*brk then do* {
T ← *cdcl-twl-restart-l-prog S*;
RETURN (*T*, *n* + *1*)
}
else
RETURN (*S*, *n*)
}›

lemma *restart-prog-l-restart-abs-l*:
‹(*uncurry2 restart-prog-l*, *uncurry2 restart-abs-l*) ∈ *Id* $\times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$ ⟨*Id*⟩*nres-rel*›
proof −
have *cdcl-twl-full-restart-l-prog*: ‹*cdcl-twl-full-restart-l-prog S* ≤ *SPEC* (*cdcl-twl-restart-l S*)›
if
inv: ‹*restart-abs-l-pre S brk*› **and**
‹(*b*, *ba*) ∈ *bool-rel*› **and**
‹*b* ∈ {*b*. *b* ⟶ *f n* < *size* (*get-learned-clss-l S*)}› **and**
‹*ba* ∈ {*b*. *b* ⟶ *f n* < *size* (*get-learned-clss-l S*)}› **and**
brk: ‹¬*brk*›
for *b ba S brk n*
proof −
obtain *T* **where**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
list-invs: ‹*twl-list-invs S*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
stgy-invs: ‹*twl-stgy-invs T*› **and**
confl: ‹*get-conflict-l S* = *None*›
using *inv brk* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal*+
by (*auto simp*: *twl-st*)

show *?thesis*
using *cdcl-twl-full-restart-l-prog-spec*[*OF ST list-invs struct-invs confl upd*]
remove-one-annot-true-clause-cdcl-twl-restart-l-spec[*OF ST list-invs struct-invs confl upd*]
by (*rule conc-trans-additional*)
qed
have *cdcl-twl-full-restart-l-GC-prog*:
⟨*cdcl-twl-full-restart-l-GC-prog S* ≤ *SPEC* (*cdcl-twl-restart-l S*)⟩
if
inv: ⟨*restart-abs-l-pre S brk*⟩ **and**
brk: ⟨*ba* ∧ *b2a* ∧ ¬ *brk*⟩
for *ba b2a brk S*
proof −
obtain *T* **where**
ST: ⟨(*S*, *T*) ∈ *twl-st-l None*⟩ **and**
struct-invs: ⟨*twl-struct-invs T*⟩ **and**
list-invs: ⟨*twl-list-invs S*⟩ **and**
upd: ⟨*clauses-to-update-l S* = {#}⟩ **and**
stgy-invs: ⟨*twl-stgy-invs T*⟩ **and**
confl: ⟨*get-conflict-l S* = *None*⟩
using *inv brk* **unfolding** *restart-abs-l-pre-def restart-prog-pre-def*
apply − **apply** *normalize-goal*+
by (*auto simp*: *twl-st*)
show *?thesis*
by (*rule cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l*[*unfolded Down-id-eq*, *OF ST list-invs struct-invs confl upd stgy-invs*])
qed

have ⟨*restart-prog-l S n brk* ≤ ⇓ *Id* (*restart-abs-l S n brk*)⟩ **for** *S n brk*
unfolding *restart-prog-l-def restart-abs-l-def restart-required-l-def cdcl-twl-restart-l-prog-def*
apply (*refine-vcg*)
subgoal by *auto*
subgoal by (*rule cdcl-twl-full-restart-l-GC-prog*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule cdcl-twl-local-restart-l-spec-cdcl-twl-restart-l*) *auto*
subgoal by (*rule cdcl-twl-full-restart-l-prog*) *auto*
subgoal by *auto*
done
then show *?thesis*
apply −
unfolding *uncurry-def*
apply (*intro frefI nres-relI*)
by *force*
qed

definition *cdcl-twl-stgy-restart-abs-early-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**
⟨*cdcl-twl-stgy-restart-abs-early-l* S_0 =
do {
ebrk ← *RES UNIV*;
(-, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(ebrk,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(-, *brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-l S*;

```
        (brk, T) ← cdcl-twl-o-prog-l T;
        (T, n) ← restart-abs-l T n brk;
 ebrk ← RES UNIV;
        RETURN (ebrk, brk, T, n)
      })
      (ebrk, False, S₀, 0);
    if ¬brk then do {
      (brk, T, -) ← WHILE_T^λ(brk, T, n). cdcl-twl-stgy-restart-abs-l-inv S₀ brk T n
      (λ(brk, -). ¬brk)
      (λ(brk, S, n).
      do {
        T ← unit-propagation-outer-loop-l S;
        (brk, T) ← cdcl-twl-o-prog-l T;
        (T, n) ← restart-abs-l T n brk;
        RETURN (brk, T, n)
      })
      (False, T, n);
      RETURN T
    } else RETURN T
  }›
```

definition *cdcl-twl-stgy-restart-abs-bounded-l* :: *'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres* **where**

```
  ‹cdcl-twl-stgy-restart-abs-bounded-l S₀ =
  do {
    ebrk ← RES UNIV;
    (-, brk, T, n) ← WHILE_T^λ(ebrk, brk, T, n). cdcl-twl-stgy-restart-abs-l-inv S₀ brk T n
      (λ(ebrk, brk, -). ¬brk ∧ ¬ebrk)
      (λ(-, brk, S, n).
      do {
        T ← unit-propagation-outer-loop-l S;
        (brk, T) ← cdcl-twl-o-prog-l T;
        (T, n) ← restart-abs-l T n brk;
 ebrk ← RES UNIV;
        RETURN (ebrk, brk, T, n)
      })
      (ebrk, False, S₀, 0);
    RETURN (brk, T)
  }›
```

definition *cdcl-twl-stgy-restart-prog-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**

```
  ‹cdcl-twl-stgy-restart-prog-l S₀ =
  do {
    (brk, T, n) ← WHILE_T^λ(brk, T, n). cdcl-twl-stgy-restart-abs-l-inv S₀ brk T n
      (λ(brk, -). ¬brk)
      (λ(brk, S, n).
      do {
 T ← unit-propagation-outer-loop-l S;
 (brk, T) ← cdcl-twl-o-prog-l T;
 (T, n) ← restart-prog-l T n brk;
 RETURN (brk, T, n)
      })
      (False, S₀, 0);
    RETURN T
  }›
```

definition *cdcl-twl-stgy-restart-prog-early-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**
‹*cdcl-twl-stgy-restart-prog-early-l* S_0 =
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(ebrk,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-l S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-l T*;
(*T*, *n*) ← *restart-prog-l T n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk*, *T*, *n*)
})
(*ebrk*, *False*, S_0, *0*);
if ¬*brk then do* {
(*brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-l S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-l T*;
(*T*, *n*) ← *restart-prog-l T n brk*;
RETURN (*brk*, *T*, *n*)
})
(*False*, *T*, *n*);
RETURN T
}
else RETURN T
}›

lemma *cdcl-twl-stgy-restart-prog-early-l-cdcl-twl-stgy-restart-abs-early-l*:
‹(*cdcl-twl-stgy-restart-prog-early-l*, *cdcl-twl-stgy-restart-abs-early-l*) ∈ {(*S*, *S'*).
(*S*, *S'*) ∈ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} $\rightarrow_f$ ⟨*Id*⟩ *nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ -›)
proof −
have [*refine0*]: ‹((*False*, *S*, *0*), (*False*, *T* , *0*)) ∈ *bool-rel* $\times_r$ *?R* $\times_r$ *nat-rel*›
if ‹(*S*, *T*) ∈ *?R*›
for *S T*
using *that* **by** *auto*
have [*refine0*]: ‹*unit-propagation-outer-loop-l x1c* ≤ ⇓ *Id* (*unit-propagation-outer-loop-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
have [*refine0*]: ‹*cdcl-twl-o-prog-l x1c* ≤ ⇓ *Id* (*cdcl-twl-o-prog-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-early-l-def cdcl-twl-stgy-restart-prog-def uncurry-def*
cdcl-twl-stgy-restart-abs-early-l-def
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** *R* = ‹{((*brk* :: *bool*, *S*, *n* :: *nat*), (*brk'*, *S'*, *n'*)).

$(S, S') \in Id \wedge brk = brk' \wedge n = n'\}$›]
WHILEIT-refine[**where** R = ‹$\{((ebrk :: bool, brk :: bool, S, n :: nat), (ebrk', brk', S', n'))$.
$(S, S') \in Id \wedge brk = brk' \wedge n = n' \wedge ebrk = ebrk'\}$›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry2*]
restart-prog-l-restart-abs-l[*THEN fref-to-Down-curry2*])
subgoal by *auto*
subgoal for *x y xa x' x1 x2 x1a x2a*
by *fastforce*
subgoal by *auto*
subgoal
by (*simp add*: *twl-st*)
subgoal by (*auto simp*: *twl-st*)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def*
by (*auto simp*: *twl-st*)
subgoal by *auto*
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def*
by (*auto simp*: *twl-st*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

lemma *cdcl-twl-stgy-restart-abs-early-l-cdcl-twl-stgy-restart-abs-early-l*:
‹(*cdcl-twl-stgy-restart-abs-early-l*, *cdcl-twl-stgy-restart-prog-early*) $\in$
$\{(S, S').\ (S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$
clauses-to-update-l S $= \{\#\}\} \rightarrow_f$
$\langle\{(S, S').\ (S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S*$\}\rangle$ *nres-rel*›
unfolding *cdcl-twl-stgy-restart-abs-early-l-def cdcl-twl-stgy-restart-prog-early-def uncurry-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** R = ‹$\{((brk :: bool, S, n :: nat), (brk', S', n'))$.
$(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge brk = brk' \wedge n = n' \wedge$
clauses-to-update-l S $= \{\#\}\}$›]
WHILEIT-refine[**where** R = ‹$\{((ebrk :: bool, brk :: bool, S, n :: nat), (ebrk' :: bool, brk', S', n'))$.
$(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge brk = brk' \wedge n = n' \wedge ebrk = ebrk' \wedge$
clauses-to-update-l S $= \{\#\}\}$›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry2*])
subgoal by *simp*
subgoal for *x y - - xa x' x1 x2 x1a x2a*
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def*
apply (*rule-tac x=y* **in** *exI*)
apply (*rule-tac x=*‹*fst* (*snd* (*snd x'*))› **in** *exI*)
by *auto*
subgoal by *fast*
subgoal

unfolding *cdcl-twl-stgy-restart-prog-inv-def*
cdcl-twl-stgy-restart-abs-l-inv-def
apply (*simp only*: *prod.case*)
apply (*normalize-goal*)+
by (*simp add*: *twl-st-l twl-st*)
subgoal by (*auto simp*: *twl-st-l twl-st*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *x y - - xa x′ x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e xb x′a x1f x2f x1g*
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def*
apply (*rule-tac x=y* **in** *exI*)
apply (*rule-tac x=*‹*fst* (*snd x′a*)› **in** *exI*)
by *auto*
subgoal by *auto*
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
cdcl-twl-stgy-restart-abs-l-inv-def
apply (*simp only*: *prod.case*)
apply (*normalize-goal*)+
by (*simp add*: *twl-st-l twl-st*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-early-l-cdcl-twl-stgy-restart-prog-early*:
‹(*cdcl-twl-stgy-restart-prog-early-l*, *cdcl-twl-stgy-restart-prog-early*)
$\in$ {(*S*, *S′*). (*S*, *S′*) $\in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$ *clauses-to-update-l S* = {#}} $\rightarrow_f$
⟨{(*S*, *S′*). (*S*, *S′*) $\in$ *twl-st-l None* $\wedge$ *twl-list-invs S*}⟩*nres-rel*›
apply (*intro frefI nres-relI*)
apply (*rule order-trans*)
defer
apply (*rule cdcl-twl-stgy-restart-abs-early-l-cdcl-twl-stgy-restart-abs-early-l*[*THEN fref-to-Down*])
apply *fast*
apply *assumption*
apply (*rule cdcl-twl-stgy-restart-prog-early-l-cdcl-twl-stgy-restart-abs-early-l*[*THEN fref-to-Down*, *simplified*])
apply *simp*
done

lemma *cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-abs-l*:
‹(*cdcl-twl-stgy-restart-prog-l*, *cdcl-twl-stgy-restart-abs-l*) $\in$ {(*S*, *S′*).
(*S*, *S′*) $\in$ *Id* $\wedge$ *twl-list-invs S* $\wedge$ *clauses-to-update-l S* = {#}} $\rightarrow_f$ ⟨*Id*⟩ *nres-rel*›
(**is** ‹- $\in$ *?R* $\rightarrow_f$ -›)
proof −
have [*refine0*]: ‹((*False*, *S*, *0*), (*False*, *T* , *0*)) $\in$ *bool-rel* $\times_r$ *?R* $\times_r$ *nat-rel*›
if ‹(*S*, *T*) $\in$ *?R*›
for *S T*
using *that* **by** *auto*
have [*refine0*]: ‹*unit-propagation-outer-loop-l x1c* $\leq \Downarrow$ *Id* (*unit-propagation-outer-loop-l x1a*)›
if ‹(*x1c*, *x1a*) $\in$ *Id*›

for *x1c x1a*
using *that* **by** *auto*
have [*refine0*]: ‹*cdcl-twl-o-prog-l x1c* $\leq \Downarrow$ *Id* (*cdcl-twl-o-prog-l x1a*)›
if ‹(*x1c*, *x1a*) ∈ *Id*›
for *x1c x1a*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-l-def cdcl-twl-stgy-restart-prog-def uncurry-def cdcl-twl-stgy-restart-abs-l-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** *R* = ‹{((*brk* :: *bool*, *S*, *n* :: *nat*), (*brk'*, *S'*, *n'*)). (*S*, *S'*) ∈ *Id* ∧ *brk* = *brk'* ∧ *n* = *n'*}›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry2*]
restart-prog-l-restart-abs-l[*THEN fref-to-Down-curry2*])
subgoal by *auto*
subgoal for *x y xa x' x1 x2 x1a x2a*
by *fastforce*
subgoal by *auto*
subgoal
by (*simp add*: *twl-st*)
subgoal by (*auto simp*: *twl-st*)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def*
by (*auto simp*: *twl-st*)
subgoal by *auto*
done
qed

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-prog*:
‹(*cdcl-twl-stgy-restart-prog-l*, *cdcl-twl-stgy-restart-prog*)
∈ {(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} $\rightarrow_f$
⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S*}⟩*nres-rel*›
apply (*intro frefI nres-relI*)
apply (*rule order-trans*)
defer
apply (*rule cdcl-twl-stgy-restart-abs-l-cdcl-twl-stgy-restart-abs-l*[*THEN fref-to-Down*])
apply *fast*
apply *assumption*
apply (*rule cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-abs-l*[*THEN fref-to-Down*, *simplified*])
apply *simp*
done

definition *cdcl-twl-stgy-restart-prog-bounded-l* :: *'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres* **where**
‹*cdcl-twl-stgy-restart-prog-bounded-l* S_0 =
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(ebrk,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-l S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-l T*;

$(T, n) \leftarrow$ *restart-prog-l T n brk*;
ebrk $\leftarrow$ *RES UNIV*;
RETURN (*ebrk*, *brk*, T, n)
})
(*ebrk*, *False*, S_0, *0*);
RETURN (*brk*, T)
}›

lemma *cdcl-twl-stgy-restart-abs-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*:
‹(*cdcl-twl-stgy-restart-abs-bounded-l*, *cdcl-twl-stgy-restart-prog-bounded*) $\in$
$\{(S, S')$. $(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$
clauses-to-update-l S $= \{\#\}\} \rightarrow_f$
$\langle$*bool-rel* $\times_r$ $\{(S, S')$. $(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S*$\}\rangle$ *nres-rel*›
unfolding *cdcl-twl-stgy-restart-abs-bounded-l-def cdcl-twl-stgy-restart-prog-bounded-def uncurry-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg*
WHILEIT-refine[**where** R = ‹$\{((ebrk :: bool, brk :: bool, S, n :: nat), (ebrk' :: bool, brk', S', n'))$.
$(S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs S* $\wedge$ $brk = brk' \wedge n = n' \wedge ebrk = ebrk' \wedge$
clauses-to-update-l S $= \{\#\}\}$›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry2*])
subgoal by *simp*
subgoal for *x y - - xa x′ x1 x2 x1a x2a*
unfolding *cdcl-twl-stgy-restart-abs-l-inv-def*
apply (*rule-tac x=y* **in** *exI*)
apply (*rule-tac x=*‹*fst* (*snd* (*snd x′*))› **in** *exI*)
by *auto*
subgoal by *fast*
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def*
cdcl-twl-stgy-restart-abs-l-inv-def
apply (*simp only*: *prod.case*)
apply (*normalize-goal*)+
by (*simp add*: *twl-st-l twl-st*)
subgoal by (*auto simp*: *twl-st-l twl-st*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

lemma *cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*:
‹(*cdcl-twl-stgy-restart-prog-bounded-l*, *cdcl-twl-stgy-restart-abs-bounded-l*) $\in \{(S, S')$.
$(S, S') \in Id \wedge$ *twl-list-invs S* $\wedge$ *clauses-to-update-l S* $= \{\#\}\} \rightarrow_f \langle Id\rangle$ *nres-rel*›
(**is** ‹- $\in$ *?R* $\rightarrow_f$ -›)
proof −
have [*refine0*]: ‹((*False*, S, *0*), (*False*, T , *0*)) $\in$ *bool-rel* $\times_r$ *?R* $\times_r$ *nat-rel*›
if ‹$(S, T) \in$ *?R*›
for S T
using *that* **by** *auto*
have [*refine0*]: ‹*unit-propagation-outer-loop-l x1c* $\leq \Downarrow$ *Id* (*unit-propagation-outer-loop-l x1a*)›
if ‹(*x1c*, *x1a*) $\in$ *Id*›
for *x1c x1a*
using *that* **by** *auto*
have [*refine0*]: ‹*cdcl-twl-o-prog-l x1c* $\leq \Downarrow$ *Id* (*cdcl-twl-o-prog-l x1a*)›

if ‹$(x1c, x1a) \in Id$›
for *x1c x1a*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-bounded-l-def cdcl-twl-stgy-restart-prog-def uncurry-def cdcl-twl-stgy-restart-abs-bounded-l-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where** $R = $ ‹$\{((brk :: bool, S, n :: nat), (brk', S', n')).$
$(S, S') \in Id \wedge brk = brk' \wedge n = n'\}$›]
WHILEIT-refine[**where** $R = $ ‹$\{((ebrk :: bool, brk :: bool, S, n :: nat), (ebrk', brk', S', n')).$
$(S, S') \in Id \wedge brk = brk' \wedge n = n' \wedge ebrk = ebrk'\}$›]
unit-propagation-outer-loop-l-spec[*THEN fref-to-Down*]
cdcl-twl-o-prog-l-spec[*THEN fref-to-Down*]
restart-abs-l-restart-prog[*THEN fref-to-Down-curry2*]
restart-prog-l-restart-abs-l[*THEN fref-to-Down-curry2*])
subgoal by *auto*
subgoal for *x y xa x′ x1 x2 x1a x2a*
by *fastforce*
subgoal by *auto*
subgoal
by (*simp add*: *twl-st*)
subgoal by (*auto simp*: *twl-st*)
subgoal
unfolding *cdcl-twl-stgy-restart-prog-inv-def cdcl-twl-stgy-restart-abs-l-inv-def*
by (*auto simp*: *twl-st*)
subgoal by *auto*
done
qed

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-prog-bounded*:
‹(*cdcl-twl-stgy-restart-prog-bounded-l*, *cdcl-twl-stgy-restart-prog-bounded*)
$\in \{(S, S'). (S, S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S \wedge clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}\} \rightarrow_f$
$\langle bool\text{-}rel \times_r \{(S, S'). (S, S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S\}\rangle nres\text{-}rel$›
apply (*intro frefI nres-relI*)
apply (*rule order-trans*)
defer
apply (*rule cdcl-twl-stgy-restart-abs-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*[*THEN fref-to-Down*])
apply *fast*
apply *assumption*
apply (*rule cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*[*THEN fref-to-Down*, *simplified*])
apply *simp*
done

end

end
theory *Watched-Literals-Watch-List*
imports *Watched-Literals-List Weidenbach-Book-Base.Explorer*
begin

1.4 Third Refinement: Remembering watched

1.4.1 Types

type-synonym *clauses-to-update-wl* = ‹*nat multiset*›
type-synonym *'v watcher* = ‹(*nat* × *'v literal* × *bool*)›
type-synonym *'v watched* = ‹*'v watcher list*›
type-synonym *'v lit-queue-wl* = ‹*'v literal multiset*›

type-synonym *'v twl-st-wl* =
‹(*'v*, *nat*) *ann-lits* × *'v clauses-l* ×
'v cconflict × *'v clauses* × *'v clauses* × *'v lit-queue-wl* ×
(*'v literal* ⇒ *'v watched*)›

1.4.2 Access Functions

fun *clauses-to-update-wl* :: ‹*'v twl-st-wl* ⇒ *'v literal* ⇒ *nat* ⇒ *clauses-to-update-wl*› **where**
‹*clauses-to-update-wl* (-, *N*, -, -, -, -, *W*) *L i* =
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*mset* (*drop i* (*map fst* (*W L*))))›

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

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

fun *set-literals-to-update-wl* :: ‹*'v lit-queue-wl* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl*› **where**
‹*set-literals-to-update-wl Q* (*M*, *N*, *D*, *NE*, *UE*, -, *W*) = (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›

fun *get-conflict-wl* :: ‹*'v twl-st-wl* ⇒ *'v cconflict*› **where**
‹*get-conflict-wl* (-, -, *D*, -, -, -, -) = *D*›

fun *get-clauses-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses-l*› **where**
‹*get-clauses-wl* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *N*›

fun *get-unit-learned-clss-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses*› **where**
‹*get-unit-learned-clss-wl* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = *UE*›

fun *get-unit-init-clss-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses*› **where**
‹*get-unit-init-clss-wl* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = *NE*›

fun *get-unit-clauses-wl* :: ‹*'v twl-st-wl* ⇒ *'v clauses*› **where**
‹*get-unit-clauses-wl* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = *NE* + *UE*›

lemma *get-unit-clauses-wl-alt-def*:
‹*get-unit-clauses-wl S* = *get-unit-init-clss-wl S* + *get-unit-learned-clss-wl S*›
by (*cases S*) *auto*

fun *get-watched-wl* :: ‹*'v twl-st-wl* ⇒ (*'v literal* ⇒ *'v watched*)› **where**
‹*get-watched-wl* (-, -, -, -, -, -, *W*) = *W*›

definition *get-learned-clss-wl* **where**
‹*get-learned-clss-wl S* = *learned-clss-lf* (*get-clauses-wl S*)›

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

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

We cannot just extract the literals of the clauses: we cannot be sure that atoms appear *both* positively and negatively in the clauses. If we could ensure that there are no pure literals, the definition of *all-lits-of-mm* can be changed to *all-lits-of-mm Ls* = $\bigcup$# *Ls*.

In this definition K is the blocking literal.

fun *correctly-marked-as-binary* **where**
  ‹*correctly-marked-as-binary N* (*i*, *K*, *b*) ⟷ (*b* ⟷ (*length* (*N* ∝ *i*) = *2*))›

declare *correctly-marked-as-binary.simps*[*simp del*]

abbreviation *distinct-watched* :: ‹′*v watched* ⇒ *bool*› **where**
  ‹*distinct-watched xs* ≡ *distinct* (*map* (λ(*i*, *j*, *k*). *i*) *xs*)›

lemma *distinct-watched-alt-def*: ‹*distinct-watched xs* = *distinct* (*map fst xs*)›
  by (*induction xs*; *auto*)

fun *correct-watching-except* :: ‹*nat* ⇒ *nat* ⇒ ′*v literal* ⇒ ′*v twl-st-wl* ⇒ *bool*› **where**
  ‹*correct-watching-except i j K* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
    (∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*)).
      (*L* = *K* ⟶
        distinct-watched (*take i* (*W L*) @ *drop j* (*W L*)) ∧
        ((∀ (*i*, *K*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧
            K ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
         (∀ (*i*, *K*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)). *b* ⟶ *i* ∈# *dom-m N*) ∧
        filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*take i* (*W L*) @ *drop j* (*W L*))) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))) ∧
      (*L* ≠ *K* ⟶
        distinct-watched (*W L*) ∧
      ((∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
         (∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *b* ⟶ *i* ∈# *dom-m N*) ∧
        filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*W L*)) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))))›

fun *correct-watching* :: ‹′*v twl-st-wl* ⇒ *bool*› **where**
  ‹*correct-watching* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
    (∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*)).
      distinct-watched (*W L*) ∧
     (∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
      (∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *b* ⟶ *i* ∈# *dom-m N*) ∧
      filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*W L*)) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))›

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

lemma *correct-watching-except-correct-watching*:
  assumes
    j: ‹*j* ≥ *length* (*W K*)› **and**
    corr: ‹*correct-watching-except i j K* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›
  shows ‹*correct-watching* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*(*K* := *take i* (*W K*)))›
proof −

have
  H1: ‹$\bigwedge$*L i′ K′ b*. *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*)) ⟹
    (*L* = *K* ⟹
      distinct-watched (*take i* (*W L*) @ *drop j* (*W L*)) ∧
      (((*i′*, *K′*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)) ⟶ *i′* ∈# *dom-m N* ⟶
          K′ ∈ *set* (*N* ∝ *i′*) ∧ *K′* ≠ *L* ∧ *correctly-marked-as-binary N* (*i′*, *K′*, *b*)) ∧
      ((*i′*, *K′*, *b*)∈#*mset* (*take i* (*W L*) @ *drop j* (*W L*)) ⟶ *b* ⟶ *i′* ∈# *dom-m N*) ∧
      filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*take i* (*W L*) @ *drop j* (*W L*))) =
        clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))))› **and**
  H2: ‹$\bigwedge$*L i K′ b*. *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*)) ⟹ (*L* ≠ *K* ⟹
      distinct-watched (*W L*) ∧
      (((*i*, *K′*, *b*)∈#*mset* (*W L*) ⟶ *i* ∈# *dom-m N* ⟶ *K′* ∈ *set* (*N* ∝ *i*) ∧ *K′* ≠ *L* ∧
        (*correctly-marked-as-binary N* (*i*, *K′*, *b*))) ∧
       ((*i*, *K′*, *b*)∈#*mset* (*W L*) ⟶ *b* ⟶ *i* ∈# *dom-m N*) ∧
      filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*W L*)) =
        clause-to-update L (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})))›
  using *corr* **unfolding** *correct-watching-except.simps*
  by *fast+*
show *?thesis*
  unfolding *correct-watching.simps*
  apply (*intro conjI allI impI ballI*)
  subgoal for *L*
    apply (*cases* ‹*L* = *K*›)
    subgoal
      using *H1*[*of L*] *j*
      by (*auto split*: *if-splits*)
    subgoal
      using *H2*[*of L*] *j*
      by (*auto split*: *if-splits*)
    done
  subgoal for *L x*
    apply (*cases* ‹*L* = *K*›)
    subgoal
      using *H1*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j*
      by (*auto split*: *if-splits*)
    subgoal
      using *H2*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›]
      by *auto*
    done
  subgoal for *L*
    apply (*cases* ‹*L* = *K*›)
    subgoal
      using *H1*[*of L* - -] *j*
      by (*auto split*: *if-splits*)
    subgoal
      using *H2*[*of L* - -]
      by *auto*
    done
  subgoal for *L*
    apply (*cases* ‹*L* = *K*›)
    subgoal
      using *H1*[*of L* - -] *j*
      by (*auto split*: *if-splits*)
    subgoal
      using *H2*[*of L* - -]
      by *auto*

    done
  done
qed

fun *watched-by* :: ⟨$'v$ *twl-st-wl* ⇒ $'v$ *literal* ⇒ $'v$ *watched*⟩ **where**
⟨*watched-by* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) *L* = *W L*⟩

fun *update-watched* :: ⟨$'v$ *literal* ⇒ $'v$ *watched* ⇒ $'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl*⟩ **where**
⟨*update-watched L WL* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*(*L*:= *WL*))⟩

lemma *bspec'*: ⟨$x \in a \Longrightarrow \forall x{\in}a.\ P\ x \Longrightarrow P\ x$⟩
by (*rule bspec*)

lemma *correct-watching-exceptD*:
assumes
⟨*correct-watching-except i j L S*⟩ **and**
⟨*L* ∈# *all-lits-of-mm*
(*mset* ‘# *ran-mf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*)⟩ **and**
w: ⟨*w* < *length* (*watched-by S L*)⟩ ⟨*w* ≥ *j*⟩ ⟨*fst* (*watched-by S L* ! *w*) ∈# *dom-m* (*get-clauses-wl S*)⟩
shows ⟨*fst* (*snd* (*watched-by S L* ! *w*)) ∈ *set* (*get-clauses-wl S* ∝ (*fst* (*watched-by S L* ! *w*)))⟩
proof −
have *H*: ⟨⋀*x*. *x*∈*set* (*take i* (*watched-by S L*)) ∪ *set* (*drop j* (*watched-by S L*)) ⟹
case x of (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m* (*get-clauses-wl S*) ⟶ *K* ∈ *set* (*get-clauses-wl S* ∝ *i*) ∧
K ≠ *L*⟩
using *assms*
by (*cases S*; *cases* ⟨*watched-by S L* ! *w*⟩)
(*auto simp add*: *add-mset-eq-add-mset simp del*: *Un-iff*
dest!: *multi-member-split*[*of L*] *dest*: *bspec*)
have ⟨∃ *i*≥*j*. *i* < *length* (*watched-by S L*) ∧
watched-by S L ! *w* = *watched-by S L* ! *i*⟩
by (*rule exI*[*of - w*])
(*use w* **in** *auto*)
then show *?thesis*
using *H*[*of* ⟨*watched-by S L* ! *w*⟩] *w*
by (*cases* ⟨*watched-by S L* ! *w*⟩) (*auto simp*: *in-set-drop-conv-nth*)
qed

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

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

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

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

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

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

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

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

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

lemma *all-lits-of-m-add-mset*:
‹*all-lits-of-m* (*add-mset* L C) = *add-mset* L (*add-mset* ($-L$) (*all-lits-of-m* C))›
by (*cases* L) (*auto simp*: *all-lits-of-m-def*)

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

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

lemma *in-all-lits-of-mm-uminusD*: ‹$x2 \in\#$ *all-lits-of-mm* $N \Longrightarrow -x2 \in\#$ *all-lits-of-mm* N›
by (*auto simp*: *all-lits-of-mm-def*)

lemma *in-all-lits-of-mm-uminus-iff*: ‹$-x2 \in\#$ *all-lits-of-mm* $N \longleftrightarrow x2 \in\#$ *all-lits-of-mm* N›
by (*cases* $x2$) (*auto simp*: *all-lits-of-mm-def*)

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

lemma *all-lits-of-mm-mono*:
‹*set-mset* $A \subseteq$ *set-mset* $B \Longrightarrow$ *set-mset* (*all-lits-of-mm* A) $\subseteq$ *set-mset* (*all-lits-of-mm* B)›
by (*auto simp*: *all-lits-of-mm-def*)

fun *st-l-of-wl* :: ‹($'v$ *literal* $\times$ *nat*) *option* $\Rightarrow$ $'v$ *twl-st-wl* $\Rightarrow$ $'v$ *twl-st-l*› **where**
‹*st-l-of-wl None* (M, N, D, NE, UE, Q, W) = (M, N, D, NE, UE, {#}, Q)›
| ‹*st-l-of-wl* (*Some* (L, j)) (M, N, D, NE, UE, Q, W) =
(M, N, D, NE, UE, (*if* $D \neq$ *None then* {#} *else clauses-to-update-wl* (M, N, D, NE, UE, Q, W) L j,
Q))›

definition *state-wl-l* :: ‹($'v$ *literal* $\times$ *nat*) *option* $\Rightarrow$ ($'v$ *twl-st-wl* $\times$ $'v$ *twl-st-l*) *set*› **where**
‹*state-wl-l* L = {(T, T'). T' = *st-l-of-wl* L T}›

fun *twl-st-of-wl* :: ‹($'v$ *literal* $\times$ *nat*) *option* $\Rightarrow$ ($'v$ *twl-st-wl* $\times$ $'v$ *twl-st*) *set*› **where**
‹*twl-st-of-wl* L = *state-wl-l* L O *twl-st-l* (*map-option fst* L)›

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

lemma [*twl-st-wl*]:
  assumes ‹$(S,\ T) \in$ *state-wl-l L*›
  shows
    ‹*get-trail-l T* = *get-trail-wl S*› **and**
    ‹*get-clauses-l T* = *get-clauses-wl S*› **and**
    ‹*get-conflict-l T* = *get-conflict-wl S*› **and**
    ‹*L* = *None* $\Longrightarrow$ *clauses-to-update-l T* = {#}›
    ‹*L* $\neq$ *None* $\Longrightarrow$ *get-conflict-wl S* $\neq$ *None* $\Longrightarrow$ *clauses-to-update-l T* = {#}›
    ‹*L* $\neq$ *None* $\Longrightarrow$ *get-conflict-wl S* = *None* $\Longrightarrow$ *clauses-to-update-l T* =
      clauses-to-update-wl S (*fst* (*the L*)) (*snd* (*the L*))› **and**
    ‹*literals-to-update-l T* = *literals-to-update-wl S*›
    ‹*get-unit-learned-clauses-l T* = *get-unit-learned-clss-wl S*›
    ‹*get-unit-init-clauses-l T* = *get-unit-init-clss-wl S*›
    ‹*get-unit-learned-clauses-l T* = *get-unit-learned-clss-wl S*›
    ‹*get-unit-clauses-l T* = *get-unit-clauses-wl S*›
  using *assms* **unfolding** *state-wl-l-def all-clss-lf-ran-m*[*symmetric*]
  by (*cases S*; *cases T*; *cases L*; *auto split*: *option.splits simp*: *trail.simps*; *fail*)+

lemma [*twl-st-l*]:
  ‹$(a,\ a') \in$ *state-wl-l None* $\Longrightarrow$
      get-learned-clss-l a' = *get-learned-clss-wl a*›
  unfolding *state-wl-l-def* **by** (*cases a*; *cases a'*)
   (*auto simp*: *get-learned-clss-l-def get-learned-clss-wl-def*)

lemma *remove-one-lit-from-wq-def*:
  ‹*remove-one-lit-from-wq L S* = *set-clauses-to-update-l* (*clauses-to-update-l S* − {#*L*#}) *S*›
  by (*cases S*) *auto*

lemma *correct-watching-set-literals-to-update*[*simp*]:
  ‹*correct-watching* (*set-literals-to-update-wl WS T'*) = *correct-watching T'*›
  by (*cases T'*) (*auto simp*: *correct-watching.simps*)

lemma [*twl-st-wl*]:
  ‹*get-clauses-wl* (*set-literals-to-update-wl W S*) = *get-clauses-wl S*›
  ‹*get-unit-init-clss-wl* (*set-literals-to-update-wl W S*) = *get-unit-init-clss-wl S*›
  by (*cases S*; *auto*; *fail*)+

lemma *get-conflict-wl-set-literals-to-update-wl*[*twl-st-wl*]:
  ‹*get-conflict-wl* (*set-literals-to-update-wl P S*) = *get-conflict-wl S*›
  ‹*get-unit-clauses-wl* (*set-literals-to-update-wl P S*) = *get-unit-clauses-wl S*›
  by (*cases S*; *auto*; *fail*)+

definition *set-conflict-wl* :: ‹'*v clause-l* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl*› **where**
  ‹*set-conflict-wl* = (λ*C* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). (*M*, *N*, *Some* (*mset C*), *NE*, *UE*, {#}, *W*))›

lemma [*twl-st-wl*]: ‹*get-clauses-wl* (*set-conflict-wl D S*) = *get-clauses-wl S*›
  by (*cases S*) (*auto simp*: *set-conflict-wl-def*)

lemma [*twl-st-wl*]:
  ‹*get-unit-init-clss-wl* (*set-conflict-wl D S*) = *get-unit-init-clss-wl S*›
  ‹*get-unit-clauses-wl* (*set-conflict-wl D S*) = *get-unit-clauses-wl S*›
  by (*cases S*; *auto simp*: *set-conflict-wl-def*; *fail*)+

lemma *state-wl-l-mark-of-is-decided*:

‹$(x,\ y) \in$ *state-wl-l* $b \Longrightarrow$
  get-trail-wl $x \neq [] \Longrightarrow$
  is-decided (*hd* (*get-trail-l* y)) = *is-decided* (*hd* (*get-trail-wl* x))›
by (*cases* ‹*get-trail-wl* x›; *cases* ‹*get-trail-l* y›; *cases* ‹*hd* (*get-trail-wl* x)›;
  cases ‹*hd* (*get-trail-l* y)›; *cases* b; *cases* x)
 (*auto simp*: *state-wl-l-def convert-lit.simps st-l-of-wl.simps*)

lemma *state-wl-l-mark-of-is-proped*:
‹$(x,\ y) \in$ *state-wl-l* $b \Longrightarrow$
  get-trail-wl $x \neq [] \Longrightarrow$
  is-proped (*hd* (*get-trail-l* y)) = *is-proped* (*hd* (*get-trail-wl* x))›
by (*cases* ‹*get-trail-wl* x›; *cases* ‹*get-trail-l* y›; *cases* ‹*hd* (*get-trail-wl* x)›;
  cases ‹*hd* (*get-trail-l* y)›; *cases* b; *cases* x)
 (*auto simp*: *state-wl-l-def convert-lit.simps*)

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

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

definition *unit-prop-body-wl-inv* **where**
‹*unit-prop-body-wl-inv* $T\ j\ i\ L \longleftrightarrow$ ($i <$ *length* (*watched-by* $T\ L$) $\wedge j \leq i \wedge$
 (*fst* (*watched-by* $T\ L$! i) $\in\#$ *dom-m* (*get-clauses-wl* T) $\longrightarrow$
 ($\exists\, T'.\ (T,\ T') \in$ *state-wl-l* (*Some* (L, i)) $\wedge j \leq i \wedge$
 unit-propagation-inner-loop-body-l-inv L (*fst* (*watched-by* $T\ L$! i))
  (*remove-one-lit-from-wq* (*fst* (*watched-by* $T\ L$! i)) T')$\wedge$
 $L \in\#$ *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl* T) + *get-unit-clauses-wl* T) $\wedge$
  correct-watching-except $j\ i\ L\ T$)))›

lemma *unit-prop-body-wl-inv-alt-def*:
‹*unit-prop-body-wl-inv* $T\ j\ i\ L \longleftrightarrow$ ($i <$ *length* (*watched-by* $T\ L$) $\wedge j \leq i \wedge$
 (*fst* (*watched-by* $T\ L$! i) $\in\#$ *dom-m* (*get-clauses-wl* T) $\longrightarrow$
 ($\exists\, T'.\ (T,\ T') \in$ *state-wl-l* (*Some* (L, i)) $\wedge$
 unit-propagation-inner-loop-body-l-inv L (*fst* (*watched-by* $T\ L$! i))
  (*remove-one-lit-from-wq* (*fst* (*watched-by* $T\ L$! i)) T')$\wedge$
 $L \in\#$ *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl* T) + *get-unit-clauses-wl* T) $\wedge$
  correct-watching-except $j\ i\ L\ T \wedge$
 get-conflict-wl T = *None* $\wedge$
 length (*get-clauses-wl* $T \propto$ *fst* (*watched-by* $T\ L$! i)) ≥ 2)))›
(**is** ‹*?A* = *?B*›)
proof
 assume *?B*
 then show *?A*
  unfolding *unit-prop-body-wl-inv-def*
  by *blast*
next
 assume *?A*
 then show *?B*
 proof (*cases* ‹*fst* (*watched-by* $T\ L$! i) $\in\#$ *dom-m* (*get-clauses-wl* T)›)
  case *False*
  then show *?B*
   using ‹*?A*› **unfolding** *unit-prop-body-wl-inv-def*
   by *blast*
 next
  case *True*
  then obtain T' **where**
   ‹$i <$ *length* (*watched-by* $T\ L$)›
   ‹$j \leq i$› **and**

TT′: ‹(*T*, *T′*) ∈ *state-wl-l* (*Some* (*L*, *i*))› **and**
inv: ‹*unit-propagation-inner-loop-body-l-inv L* (*fst* (*watched-by T L* ! *i*))
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*)› **and**
‹*L* ∈# *all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl T*) + *get-unit-clauses-wl T*)›
‹*correct-watching-except j i L T*›
using ‹*?A*› **unfolding** *unit-prop-body-wl-inv-def*
by *blast*

obtain *x* **where**
x: ‹(*set-clauses-to-update-l*
(*clauses-to-update-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) +
{#*fst* (*watched-by T L* ! *i*)#})
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*),
x)
∈ *twl-st-l* (*Some L*)› **and**
struct-invs: ‹*twl-struct-invs x*› **and**
‹*twl-stgy-invs x*› **and**
‹*fst* (*watched-by T L* ! *i*)
∈# *dom-m*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*))› **and**
‹*0* < *fst* (*watched-by T L* ! *i*)› **and**
‹*0* < *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) ∝
fst (*watched-by T L* ! *i*))› **and**
‹*no-dup*
(*get-trail-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*))› **and**
‹(*if get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) ∝
fst (*watched-by T L* ! *i*) !
0 =
L
then 0 else 1)
< *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) ∝
fst (*watched-by T L* ! *i*))› **and**
‹*1* −
(*if get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) ∝
fst (*watched-by T L* ! *i*) !
0 =
L
then 0 else 1)
< *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) ∝
fst (*watched-by T L* ! *i*))› **and**
‹*L* ∈ *set* (*watched-l*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) ∝
fst (*watched-by T L* ! *i*)))› **and**
confl: ‹*get-conflict-l* (*remove-one-lit-from-wq* (*fst* (*watched-by T L* ! *i*)) *T′*) = *None*›

using *inv* **unfolding** *unit-propagation-inner-loop-body-l-inv-def* **by** *blast*

have ‹*Multiset.Ball* (*get-clauses* *x*) *struct-wf-twl-cls*›
using *struct-invs* **unfolding** *twl-struct-invs-def twl-st-inv-alt-def* **by** *blast*
moreover have ‹*twl-clause-of* (*get-clauses-wl* *T* ∝ *fst* (*watched-by* *T* *L* ! *i*)) ∈# *get-clauses* *x*›
using *TT′ x True* **by** *auto*
ultimately have *1*: ‹*length* (*get-clauses-wl* *T* ∝ *fst* (*watched-by* *T* *L* ! *i*)) ≥ *2*›
by *auto*
have *2*: ‹*get-conflict-wl* *T* = *None*›
using *confl TT′ x* **by** *auto*
show *?B*
using ‹*?A*› *1 2* **unfolding** *unit-prop-body-wl-inv-def*
by *blast*
qed
qed

definition *propagate-lit-wl-general* :: ‹*′v literal* ⇒ *nat* ⇒ *nat* ⇒ *′v twl-st-wl* ⇒ *′v twl-st-wl*› **where**
‹*propagate-lit-wl-general* = (λ*L′ C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
let N = (*if length* (*N* ∝ *C*) > *2 then N*(*C* ↪ *swap* (*N* ∝ *C*) *0* (*Suc 0* − *i*)) *else N*) *in*
(*Propagated L′ C* # *M*, *N*, *D*, *NE*, *UE*, *add-mset* (−*L′*) *Q*, *W*))›

definition *propagate-lit-wl* :: ‹*′v literal* ⇒ *nat* ⇒ *nat* ⇒ *′v twl-st-wl* ⇒ *′v twl-st-wl*› **where**
‹*propagate-lit-wl* = (λ*L′ C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
let N = *N*(*C* ↪ *swap* (*N* ∝ *C*) *0* (*Suc 0* − *i*)) *in*
(*Propagated L′ C* # *M*, *N*, *D*, *NE*, *UE*, *add-mset* (−*L′*) *Q*, *W*))›

definition *propagate-lit-wl-bin* :: ‹*′v literal* ⇒ *nat* ⇒ *nat* ⇒ *′v twl-st-wl* ⇒ *′v twl-st-wl*› **where**
‹*propagate-lit-wl-bin* = (λ*L′ C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
(*Propagated L′ C* # *M*, *N*, *D*, *NE*, *UE*, *add-mset* (−*L′*) *Q*, *W*))›

definition *keep-watch* **where**
‹*keep-watch* = (λ*L i j* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*(*L* := (*W L*)[*i* := *W L* ! *j*])))›

lemma *length-watched-by-keep-watch*[*twl-st-wl*]:
‹*length* (*watched-by* (*keep-watch L i j S*) *K*) = *length* (*watched-by S K*)›
by (*cases S*) (*auto simp*: *keep-watch-def*)

lemma *watched-by-keep-watch-neq*[*twl-st-wl*, *simp*]:
‹*w* < *length* (*watched-by S L*) ⟹ *watched-by* (*keep-watch L j w S*) *L* ! *w* = *watched-by S L* ! *w*›
by (*cases S*) (*auto simp*: *keep-watch-def*)

lemma *watched-by-keep-watch-eq*[*twl-st-wl*, *simp*]:
‹*j* < *length* (*watched-by S L*) ⟹ *watched-by* (*keep-watch L j w S*) *L* ! *j* = *watched-by S L* ! *w*›
by (*cases S*) (*auto simp*: *keep-watch-def*)

definition *update-clause-wl* :: ‹*′v literal* ⇒ *nat* ⇒ *bool* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *′v twl-st-wl* ⇒ (*nat* × *nat* × *′v twl-st-wl*) *nres*› **where**
‹*update-clause-wl* = (λ(*L*::*′v literal*) *C b j w i f* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
let K′ = (*N*∝*C*) ! *f*;
let N′ = *N*(*C* ↪ *swap* (*N* ∝ *C*) *i f*);
RETURN (*j*, *w*+*1*, (*M*, *N′*, *D*, *NE*, *UE*, *Q*, *W*(*K′* := *W K′* @ [(*C*, *L*, *b*)])))
})›

definition *update-blit-wl* :: ‹'v literal ⇒ nat ⇒ bool ⇒ nat ⇒ nat ⇒ 'v literal ⇒ 'v twl-st-wl ⇒ (nat × nat × 'v twl-st-wl) nres› **where**
‹update-blit-wl = (λ(L::'v literal) C b j w K (M, N, D, NE, UE, Q, W). do {
RETURN (j+1, w+1, (M, N, D, NE, UE, Q, W(L := (W L)[j:=(C, K, b)])))
})›

definition *unit-prop-body-wl-find-unwatched-inv* **where**
‹unit-prop-body-wl-find-unwatched-inv f C S ⟷
get-clauses-wl S ∝ C ≠ [] ∧
(f = None ⟷ (∀ L∈#mset (unwatched-l (get-clauses-wl S ∝ C)). − L ∈ lits-of-l (get-trail-wl S)))›

abbreviation *remaining-nondom-wl* **where**
‹remaining-nondom-wl w L S ≡
(if get-conflict-wl S = None
then size (filter-mset (λ(i, -). i ∉# dom-m (get-clauses-wl S)) (mset (drop w (watched-by S L)))) else 0)›

definition *unit-propagation-inner-loop-wl-loop-inv* **where**
‹unit-propagation-inner-loop-wl-loop-inv L = (λ(j, w, S).
(∃ S'. (S, S') ∈ state-wl-l (Some (L, w)) ∧ j≤ w ∧
unit-propagation-inner-loop-l-inv L (S', remaining-nondom-wl w L S) ∧
correct-watching-except j w L S ∧ w ≤ length (watched-by S L)))›

lemma *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*:
assumes
j-w: ‹j ≤ w› **and**
w-le: ‹w < length (watched-by S L)› **and**
corr: ‹correct-watching-except j w L S›
shows ‹correct-watching-except (Suc j) (Suc w) L (keep-watch L j w S)›
proof −
obtain *M N D NE UE Q W* **where** *S*: ‹S = (M, N, D, NE, UE, Q, W)› **by** (*cases S*)
have
Hneq: ‹⋀La. La∈#all-lits-of-mm (mset '# ran-mf N + (NE + UE)) ⟶
(La ≠ L ⟶
distinct-watched (W La) ∧
(∀ (i, K, b)∈#mset (W La). i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧ K ≠ La ∧
correctly-marked-as-binary N (i, K, b)) ∧
(∀ (i, K, b)∈#mset (W La). b ⟶ i ∈# dom-m N) ∧
{#i ∈# fst '# mset (W La). i ∈# dom-m N#} = clause-to-update La (M, N, D, NE, UE, {#}, {#}))› **and**
Heq: ‹⋀La. La∈#all-lits-of-mm (mset '# ran-mf N + (NE + UE)) ⟶
(La = L ⟶
distinct-watched (take j (W La) @ drop w (W La)) ∧
(∀ (i, K, b)∈#mset (take j (W La) @ drop w (W La)). i ∈# dom-m N ⟶ K ∈ set (N ∝ i) ∧
K ≠ La ∧ correctly-marked-as-binary N (i, K, b)) ∧
(∀ (i, K, b)∈#mset (take j (W La) @ drop w (W La)). b ⟶ i ∈# dom-m N) ∧
{#i ∈# fst '# mset (take j (W La) @ drop w (W La)). i ∈# dom-m N#} =
clause-to-update La (M, N, D, NE, UE, {#}, {#}))›
using *corr* **unfolding** *S correct-watching-except.simps*
by *fast+*

have *eq*: ‹mset (take (Suc j) ((W(L := (W L)[j := W L ! w])) La) @ drop (Suc w) ((W(L := (W L)[j := W L ! w])) La)) =
mset (take j (W La) @ drop w (W La))› **if** [*simp*]: ‹La = L› **for** *La*
using *w-le j-w*

by (*auto simp*: *S take-Suc-conv-app-nth Cons-nth-drop-Suc*[*symmetric*]
list-update-append)

have ⟨*case x of* (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* = *L*⟩ **and**
⟨*x* ∈# *mset* (*take* (*Suc j*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))⟩
for *La* :: ⟨′*a literal*⟩ **and** *x* :: ⟨*nat* × ′*a literal* × *bool*⟩
using *that Heq*[*of L*]
apply (*subst* (*asm*) *eq*)
by (*simp-all add*: *eq*)
moreover have ⟨*case x of* (*i*, *K*, *b*) ⇒ *b* ⟶ *i* ∈# *dom-m N*⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* = *L*⟩ **and**
⟨*x* ∈# *mset* (*take* (*Suc j*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))⟩
for *La* :: ⟨′*a literal*⟩ **and** *x* :: ⟨*nat* × ′*a literal* × *bool*⟩
using *that Heq*[*of L*]
by (*subst* (*asm*) *eq*) *blast*+
moreover have ⟨{#*i* ∈# *fst* ‘#
mset
(*take* (*Suc j*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)).
i ∈# *dom-m N*#} =
clause-to-update La (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* = *L*⟩
for *La* :: ⟨′*a literal*⟩
using *that Heq*[*of L*]
by (*subst eq*) *simp-all*
moreover have ⟨*case x of* (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* ≠ *L*⟩ **and**
⟨*x* ∈# *mset* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)⟩
for *La* :: ⟨′*a literal*⟩ **and** *x* :: ⟨*nat* × ′*a literal* × *bool*⟩
using *that Hneq*[*of La*]
by *simp*
moreover have ⟨*case x of* (*i*, *K*, *b*) ⇒ *b* ⟶ *i* ∈# *dom-m N*⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* ≠ *L*⟩ **and**
⟨*x* ∈# *mset* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)⟩
for *La* :: ⟨′*a literal*⟩ **and** *x* :: ⟨*nat* × ′*a literal* × *bool*⟩
using *that Hneq*[*of La*]
by *auto*
moreover have ⟨{#*i* ∈# *fst* ‘# *mset* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*). *i* ∈# *dom-m N*#} =
clause-to-update La (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))⟩ **and**

$\langle La \neq L\rangle$
for *La* :: $\langle$*'a literal*$\rangle$
using *that Hneq*[*of La*]
by *simp*
moreover have $\langle$*distinct-watched* $((W(L := (W\ L)[j := W\ L\ !\ w]))\ La)\rangle$
if
$\langle La \in\#$ *all-lits-of-mm* (*mset* '# *ran-mf* $N + (NE + UE))\rangle$ **and**
$\langle La \neq L\rangle$
for *La* :: $\langle$*'a literal*$\rangle$
using *that Hneq*[*of La*]
by *simp*
moreover have $\langle$*distinct-watched* (*take* (*Suc j*) $((W(L := (W\ L)[j := W\ L\ !\ w]))\ La)$ @
drop (*Suc w*) $((W(L := (W\ L)[j := W\ L\ !\ w]))\ La))\rangle$
if
$\langle La \in\#$ *all-lits-of-mm* (*mset* '# *ran-mf* $N + (NE + UE))\rangle$ **and**
$\langle La = L\rangle$
for *La* :: $\langle$*'a literal*$\rangle$
using *that Heq*[*of La*]
apply (*subst distinct-mset-mset-distinct*[*symmetric*])
apply (*subst mset-map*)
apply (*subst eq*)
apply (*simp add*: *that*)
apply (*subst mset-map*[*symmetric*])
apply (*subst distinct-mset-mset-distinct*)
apply *simp*
done
ultimately show *?thesis*
unfolding *S keep-watch-def prod.simps correct-watching-except.simps*
by *meson*
qed

lemma *correct-watching-except-update-blit*:
assumes
corr: $\langle$*correct-watching-except* $i\ j\ L\ (a, b, c, d, e, f, g(L := (g\ L)[j' := (x1, C, b')]))\rangle$ **and**
C': $\langle C' \in\#$ *all-lits-of-mm* (*mset* '# *ran-mf* $b + (d + e))\rangle$
$\langle C' \in set\ (b \propto x1)\rangle$
$\langle C' \neq L\rangle$ **and**
corr-watched: $\langle$*correctly-marked-as-binary* $b\ (x1, C', b')\rangle$
shows $\langle$*correct-watching-except* $i\ j\ L\ (a, b, c, d, e, f, g(L := (g\ L)[j' := (x1, C', b')]))\rangle$
proof –
have
Hdisteq: $\langle\bigwedge La\ i'\ K'\ b''.\ La \in\#$*all-lits-of-mm* (*mset* '# *ran-mf* $b + (d + e)) \Longrightarrow$
$(La = L \longrightarrow$
distinct-watched (*take* i $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)$ @ *drop* j $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)))\rangle$ **and**
Heq: $\langle\bigwedge La\ i'\ K'\ b''.\ La \in\#$*all-lits-of-mm* (*mset* '# *ran-mf* $b + (d + e)) \Longrightarrow$
$(La = L \longrightarrow$
$(((i', K', b'') \in\#$*mset* (*take* i $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)$ @ *drop* j $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)) \longrightarrow$
$i' \in\#$ *dom-m* $b \longrightarrow K' \in set\ (b \propto i') \wedge K' \neq La \wedge$ *correctly-marked-as-binary* $b\ (i', K', b''))$
$\wedge$
$((i', K', b'') \in\#$*mset* (*take* i $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)$ @ *drop* j $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)) \longrightarrow$
$b'' \longrightarrow i' \in\#$ *dom-m* $b)) \wedge$
$\{\#i \in\#$ *fst* '# *mset* (*take* i $((g(L := (g\ L)[j' := (x1, C, b')]))\ La)$ @ *drop* j $((g(L := (g\ L)[j' :=$

$(x1,\ C,\ b')])) \ La)).$
$i \in\# \ dom\text{-}m\ b\#\} =$
clause-to-update La $(a,\ b,\ c,\ d,\ e,\ \{\#\},\ \{\#\}))$› **and**
Hdistneq: ‹$\bigwedge La\ i'\ K'\ b''.\ La \in\# all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ b + (d + e)) \Longrightarrow$
$(La \neq L \longrightarrow distinct\text{-}watched\ (((g(L := (g\ L)[j' := (x1,\ C,\ b')]))\ La)))$› **and**
Hneq: ‹$\bigwedge La\ i\ K\ b''.\ La \in\# all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ b + (d + e)) \Longrightarrow La \neq L \Longrightarrow$
$distinct\text{-}watched\ (((g(L := (g\ L)[j' := (x1,\ C,\ b')]))\ La)) \wedge$
$((i,\ K,\ b'') \in\# mset\ ((g(L := (g\ L)[j' := (x1,\ C,\ b')]))\ La) \longrightarrow i \in\# \ dom\text{-}m\ b \longrightarrow$
$K \in set\ (b \propto i) \wedge K \neq La \wedge correctly\text{-}marked\text{-}as\text{-}binary\ b\ (i,\ K,\ b'')) \wedge$
$((i,\ K,\ b'') \in\# mset\ ((g(L := (g\ L)[j' := (x1,\ C,\ b')]))\ La) \longrightarrow b'' \longrightarrow i \in\# \ dom\text{-}m\ b) \wedge$
$\{\#i \in\# \ fst\ `\#\ mset\ ((g(L := (g\ L)[j' := (x1,\ C,\ b')]))\ La).\ i \in\# \ dom\text{-}m\ b\#\} =$
clause-to-update La $(a,\ b,\ c,\ d,\ e,\ \{\#\},\ \{\#\})$›
using *corr* **unfolding** *correct-watching-except.simps*
by *fast+*
define g' **where** ‹$g' = g(L := (g\ L)[j' := (x1,\ C,\ b')])$›
have *g-g'*: ‹$g(L := (g\ L)[j' := (x1,\ C',\ b')]) = g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$›
unfolding *g'-def* **by** *auto*

have *H2*: ‹$fst\ `\#\ mset\ ((g'(L := (g'\ L)[j' := (x1,\ C',\ b')]))\ La) = fst\ `\#\ mset\ (g'\ La)$› **for** *La*
unfolding *g'-def*
by (*auto simp flip*: *mset-map simp*: *map-update*)
have *H3*: ‹$fst\ `\#$
mset
$(take\ i\ ((g'(L := (g'\ L)[j' := (x1,\ C',\ b')]))\ La)\ @$
$drop\ j\ ((g'(L := (g'\ L)[j' := (x1,\ C',\ b')]))\ La)) =$
$fst\ `\#$
mset
$(take\ i\ (g'\ La)\ @$
$drop\ j\ (g'\ La))$› **for** *La*
unfolding *g'-def*
by (*auto simp flip*: *mset-map drop-map simp*: *map-update*)
have [*simp*]:
‹$fst\ `\#\ mset\ ((take\ i\ (g'\ L))[j' := (x1,\ C',\ b')]) = fst\ `\#\ mset\ (take\ i\ (g'\ L))$›
‹$fst\ `\#\ mset\ ((drop\ j\ ((g'\ L)[j' := (x1,\ C',\ b')]))) = fst\ `\#\ mset\ (drop\ j\ (g'\ L))$›
‹$\neg j' < j \Longrightarrow fst\ `\#\ mset\ ((drop\ j\ (g'\ L))[j' - j := (x1,\ C',\ b')]) = fst\ `\#\ mset\ (drop\ j\ (g'\ L))$›
unfolding *g'-def*
apply (*auto simp flip*: *mset-map drop-map simp*: *map-update drop-update-swap*; *fail*)
apply (*auto simp flip*: *mset-map drop-map simp*: *map-update drop-update-swap*; *fail*)
apply (*auto simp flip*: *mset-map drop-map simp*: *map-update drop-update-swap*; *fail*)
done
have ‹$j' < length\ (g'\ L) \Longrightarrow j' < i \Longrightarrow (x1,\ C,\ b') \in set\ ((take\ i\ (g\ L))[j' := (x1,\ C,\ b')])$›
using *nth-mem*[*of* ‹j'› ‹$(take\ i\ (g\ L))[j' := (x1,\ C,\ b')]$›] **unfolding** *g'-def*
by *auto*
then have *H*: ‹$L \in\# all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ b + (d + e)) \Longrightarrow j' < length\ (g'\ L) \Longrightarrow$
$j' < i \Longrightarrow b' \Longrightarrow x1 \in\# \ dom\text{-}m\ b$›
using C' *Heq*[*of L x1 C b'*]
by (*cases* ‹$j' < j$›) (*simp*, *auto*)
have ‹$\neg\ j' < j \Longrightarrow j' - j < length\ (g'\ L) - j \Longrightarrow$
$(x1,\ C,\ b') \in set\ (drop\ j\ ((g\ L)[j' := (x1,\ C,\ b')]))$›
using *nth-mem*[*of* ‹$j'-j$› ‹$drop\ j\ ((g\ L)[j' := (x1,\ C,\ b')])$›] **unfolding** *g'-def*
by *auto*
then have *H'*: ‹$L \in\# all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ b + (d + e)) \Longrightarrow \neg\ j' < j \Longrightarrow$
$j' - j < length\ (g'\ L) - j \Longrightarrow b' \Longrightarrow x1 \in\# \ dom\text{-}m\ b$›
using C' *Heq*[*of L x1 C b'*] **unfolding** *g'-def*
by (*cases* ‹$j' < j$›) *auto*

have *dist*: ‹$La \in\# all\text{-}lits\text{-}of\text{-}mm$ (*mset* ‘# *ran-mf b* + (*d* + *e*)) $\Longrightarrow$
$La = L \Longrightarrow$
distinct-watched (*take i* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*) @ *drop j* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*))›
for *La*
using *Hdisteq*[*of L*] **unfolding** *g-g'*[*symmetric*]
by (*cases* ‹$j' < j$›)
(*auto simp*: *map-update drop-update-swap*)

have ‹$La \in\# all\text{-}lits\text{-}of\text{-}mm$ (*mset* ‘# *ran-mf b* + (*d* + *e*)) $\Longrightarrow$
$La = L \Longrightarrow$
distinct-watched (*take i* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*) @ *drop j* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*)) $\wedge$
((i', K, b'')$\in\#$*mset* (*take i* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*) @ *drop j* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*)) $\longrightarrow$
$i' \in\#$ *dom-m b* $\longrightarrow K \in$ *set* ($b \propto i'$) $\wedge K \neq La \wedge$ *correctly-marked-as-binary b* (i', K, b'')) $\wedge$
((i', K, b'')$\in\#$*mset* (*take i* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*) @ *drop j* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*)) $\longrightarrow$
$b'' \longrightarrow i' \in\#$ *dom-m b*) $\wedge$
{#$i \in\#$ *fst* ‘# *mset* (*take i* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*) @ *drop j* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*)).
$i \in\#$ *dom-m b*#} =
clause-to-update La (*a*, *b*, *c*, *d*, *e*, {#}, {#})› **for** *La i' K b''*
using *C' Heq*[*of La i' K*] *Heq*[*of La i' K b'*] *H H' dist*[*of La*] *corr-watched* **unfolding** *g-g' g'-def*[*symmetric*]
by (*cases* ‹$j' < j$›)
(*auto elim*!: *in-set-upd-cases simp*: *drop-update-swap simp del*: *distinct-append*)
moreover have ‹$La \in\# all\text{-}lits\text{-}of\text{-}mm$ (*mset* ‘# *ran-mf b* + (*d* + *e*)) $\Longrightarrow$
($La \neq L \longrightarrow$
distinct-watched (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*) $\wedge$
($\forall$(i, K, ba)$\in\#$*mset* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*).
$i \in\#$ *dom-m b* $\longrightarrow$
$K \in$ *set* ($b \propto i$) $\wedge$
$K \neq La \wedge$ *correctly-marked-as-binary b* (i, K, ba)) $\wedge$
($\forall$(i, K, ba)$\in\#$*mset* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*).
$ba \longrightarrow i \in\#$ *dom-m b*) $\wedge$
{#$i \in\#$ *fst* ‘# *mset* (($g'(L := (g'\ L)[j' := (x1,\ C',\ b')])$) *La*).
$i \in\#$ *dom-m b*#} =
clause-to-update La (*a*, *b*, *c*, *d*, *e*, {#}, {#}))›
for *La*
using *Hneq Hdistneq*
unfolding *correct-watching-except.simps g-g' g'-def*[*symmetric*]
by *auto*

ultimately show *?thesis*
unfolding *correct-watching-except.simps g-g' g'-def*[*symmetric*]
unfolding *H2 H3*
by *blast*
qed

lemma *correct-watching-except-correct-watching-except-Suc-notin*:
assumes
‹*fst* (*watched-by S L* ! *w*) $\notin\#$ *dom-m* (*get-clauses-wl S*)› **and**
j-w: ‹$j \leq w$› **and**
w-le: ‹$w <$ *length* (*watched-by S L*)› **and**

corr: ⟨*correct-watching-except j w L S*⟩
shows ⟨*correct-watching-except j* (*Suc w*) *L* (*keep-watch L j w S*)⟩
proof −
obtain *M N D NE UE Q W* **where** *S*: ⟨*S* = (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)⟩ **by** (*cases S*)
have [*simp*]: ⟨*fst* (*W L* ! *w*) ∉# *dom-m N*⟩
using *assms* **unfolding** *S* **by** *auto*
have
Hneq: ⟨⋀*La*. *La*∈#*all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*)) ⟶
(*La* ≠ *L* ⟶
distinct-watched (*W La*) ∧
((∀(*i*, *K*, *b*)∈#*mset* (*W La*). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)) ∧
(∀(*i*, *K*, *b*)∈#*mset* (*W La*). *b* ⟶ *i* ∈# *dom-m N*)) ∧
{#*i* ∈# *fst* '# *mset* (*W La*). *i* ∈# *dom-m N*#} = *clause-to-update La* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))⟩ **and**
Heq: ⟨⋀*La*. *La*∈#*all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*)) ⟶
(*La* = *L* ⟶
distinct-watched (*take j* (*W La*) @ *drop w* (*W La*)) ∧
((∀(*i*, *K*, *b*)∈#*mset* (*take j* (*W La*) @ *drop w* (*W La*)). *i* ∈# *dom-m N* ⟶
K ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
(∀(*i*, *K*, *b*)∈#*mset* (*take j* (*W La*) @ *drop w* (*W La*)). *b* ⟶ *i* ∈# *dom-m N*) ∧
{#*i* ∈# *fst* '# *mset* (*take j* (*W La*) @ *drop w* (*W La*)). *i* ∈# *dom-m N*#} =
clause-to-update La (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})))⟩
using *corr* **unfolding** *S correct-watching-except.simps*
by *fast*+

have *eq*: ⟨*mset* (*take j* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @ *drop* (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)) =
remove1-mset (*W L* ! *w*) (*mset* (*take j* (*W La*) @ *drop w* (*W La*)))⟩ **if** [*simp*]: ⟨*La* = *L*⟩ **for** *La*
using *w-le j-w*
by (*auto simp*: *S take-Suc-conv-app-nth Cons-nth-drop-Suc*[*symmetric*]
list-update-append)

have ⟨*case x of* (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* = *L*⟩ **and**
⟨*x* ∈# *mset* (*take j* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))⟩
for *La* :: ⟨'*a literal*⟩ **and** *x* :: ⟨*nat* × '*a literal* × *bool*⟩
using *that Heq*[*of L*] *w-le j-w*
by (*subst* (*asm*) *eq*) (*auto dest*!: *in-diffD*)
moreover have ⟨*case x of* (*i*, *K*, *b*) ⇒ *b* ⟶ *i* ∈# *dom-m N*⟩
if
⟨*La* ∈# *all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*))⟩ **and**
⟨*La* = *L*⟩ **and**
⟨*x* ∈# *mset* (*take j* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))⟩
for *La* :: ⟨'*a literal*⟩ **and** *x* :: ⟨*nat* × '*a literal* × *bool*⟩
using *that Heq*[*of L*] *w-le j-w*
by (*subst* (*asm*) *eq*) (*force dest*: *in-diffD*)+
moreover have ⟨{#*i* ∈# *fst* '#
mset
(*take j* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)).

$i \in\# \ dom\text{-}m\ N\#\} =$
clause-to-update La $(M, N, D, NE, UE, \{\#\}, \{\#\})$›
if
‹*La* $\in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))› **and**
‹*La* = *L*›
for *La* :: ‹′*a literal*›
using *that Heq*[*of L*] *w-le j-w*
by (*subst eq*) (*auto dest*!: *in-diffD simp*: *image-mset-remove1-mset-if*)
moreover have ‹*case x of* $(i, K, b) \Rightarrow i \in\#$ *dom-m N* $\longrightarrow K \in$ *set* $(N \propto i) \wedge K \neq La \wedge$
correctly-marked-as-binary N (i, K, b)›
if
‹*La* $\in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))› **and**
‹*La* $\neq$ *L*› **and**
‹$x \in\#$ *mset* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)›
for *La* :: ‹′*a literal*› **and** *x* :: ‹*nat* × ′*a literal* × *bool*›
using *that Hneq*[*of La*]
by *simp*
moreover have ‹*case x of* $(i, K, b) \Rightarrow b \longrightarrow i \in\#$ *dom-m N*›
if
‹*La* $\in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))› **and**
‹*La* $\neq$ *L*› **and**
‹$x \in\#$ *mset* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)›
for *La* :: ‹′*a literal*› **and** *x* :: ‹*nat* × ′*a literal* × *bool*›
using *that Hneq*[*of La*]
by *auto*
moreover have ‹$\{\#i \in\#$ *fst* ‘# *mset* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*). $i \in\#$ *dom-m N*#} =
clause-to-update La $(M, N, D, NE, UE, \{\#\}, \{\#\})$›
if
‹*La* $\in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))› **and**
‹*La* $\neq$ *L*›
for *La* :: ‹′*a literal*›
using *that Hneq*[*of La*]
by *simp*
moreover have ‹*distinct-watched* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*)›
if
‹*La* $\in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))› **and**
‹*La* $\neq$ *L*›
for *La* :: ‹′*a literal*›
using *that Hneq*[*of La*]
by *simp*
moreover have ‹*distinct-watched* (*take j* ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*) @
drop (*Suc w*) ((*W*(*L* := (*W L*)[*j* := *W L* ! *w*])) *La*))›
if
‹*La* $\in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))› **and**
‹*La* = *L*›
for *La* :: ‹′*a literal*›
using *that Heq*[*of L*] *w-le j-w* **apply** −
apply (*subst distinct-mset-mset-distinct*[*symmetric*])
apply (*subst mset-map*)
apply (*subst eq*)
apply (*solves simp*)
apply (*subst* (*asm*) *distinct-mset-mset-distinct*[*symmetric*])
apply (*subst* (*asm*) *mset-map*)
apply (*rule distinct-mset-mono*[*of* - ‹{#*i*. (*i*, *j*, *k*) $\in\#$ *mset* (*take j* (*W L*) @ *drop w* (*W L*))#}›])
by (*auto simp*: *image-mset-remove1-mset-if split*: *if-splits*)
ultimately show *?thesis*

unfolding *S keep-watch-def prod.simps correct-watching-except.simps*
by *fast*
qed

lemma *correct-watching-except-correct-watching-except-update-clause*:
assumes
corr: ⟨*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(M, N, D, NE, UE, Q, $W(L := (W\ L)[j := W\ L\ !\ w])$)⟩ **and**
j-w: ⟨$j \le w$⟩ **and**
w-le: ⟨$w <$ *length* ($W\ L$)⟩ **and**
L': ⟨$L' \in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf* N + (NE + UE))⟩
⟨$L' \in$ *set* ($N \propto x1$)⟩**and**
L-L: ⟨$L \in\#$ *all-lits-of-mm* ({#*mset* (*fst* x). $x \in\#$ *ran-m* N#} + (NE + UE))⟩ **and**
L: ⟨$L \neq N \propto x1\ !\ xa$⟩ **and**
dom: ⟨$x1 \in\#$ *dom-m* N⟩ **and**
i-xa: ⟨$i <$ *length* ($N \propto x1$)⟩ ⟨$xa <$ *length* ($N \propto x1$)⟩ **and**
[*simp*]: ⟨$W\ L\ !\ w = (x1, x2, b)$⟩ **and**
N-i: ⟨$N \propto x1\ !\ i = L$⟩ ⟨$N \propto x1\ !\ (1 - i) \neq L$⟩⟨$N \propto x1\ !\ xa \neq L$⟩ **and**
N-xa: ⟨$N \propto x1\ !\ xa \neq N \propto x1\ !\ i$⟩ ⟨$N \propto x1\ !\ xa \neq N \propto x1\ !\ (Suc\ 0 - i)$⟩**and**
i-2: ⟨$i < 2$⟩ **and** ⟨$xa \ge 2$⟩ **and**
L-neq: ⟨$L' \neq N \propto x1\ !\ xa$⟩ — The new blocking literal is not the new watched literal.
shows ⟨*correct-watching-except j* (*Suc w*) *L*
(M, $N(x1 \hookrightarrow$ *swap* ($N \propto x1$) *i xa*), D, NE, UE, Q, W
($L := (W\ L)[j := (x1, x2, b)]$,
$N \propto x1\ !\ xa := W\ (N \propto x1\ !\ xa)$ @ $[(x1, L', b)]$))⟩
proof −
define W' **where** ⟨$W' \equiv W(L := (W\ L)[j := W\ L\ !\ w])$⟩
have ⟨*length* ($N \propto x1$) > 2⟩
using *i-2 i-xa assms*
by (*auto simp*: *correctly-marked-as-binary.simps*)

have
Heq: ⟨⋀*La i K b*. *La*∈#*all-lits-of-mm* (*mset* ‘# *ran-mf* N + (NE + UE)) ⟹
$La = L$ ⟹
distinct-watched (*take* (*Suc j*) (W' *La*) @ *drop* (*Suc w*) (W' *La*)) ∧
((i, K, b)∈#*mset* (*take* (*Suc j*) (W' *La*) @ *drop* (*Suc w*) (W' *La*)) ⟶
$i \in\#$ *dom-m* N ⟶ $K \in$ *set* ($N \propto i$) ∧ $K \neq La$ ∧ *correctly-marked-as-binary* N (i, K, b)) ∧
((i, K, b)∈#*mset* (*take* (*Suc j*) (W' *La*) @ *drop* (*Suc w*) (W' *La*)) ⟶
b ⟶ $i \in\#$ *dom-m* N) ∧
{#$i \in\#$ *fst* ‘#
mset
(*take* (*Suc j*) (W' *La*) @ *drop* (*Suc w*) (W' *La*)).
$i \in\#$ *dom-m* N#} =
clause-to-update La (M, N, D, NE, UE, {#}, {#})⟩ **and**
Hneq: ⟨⋀*La i K b*. *La*∈#*all-lits-of-mm* (*mset* ‘# *ran-mf* N + (NE + UE)) ⟹
$La \neq L$ ⟹
distinct-watched (W' *La*) ∧
((i, K, b)∈#*mset* (W' *La*) ⟶ $i \in\#$ *dom-m* N ⟶ $K \in$ *set* ($N \propto i$) ∧ $K \neq La$ ∧
correctly-marked-as-binary N (i, K, b)) ∧
((i, K, b)∈#*mset* (W' *La*) ⟶ b ⟶ $i \in\#$ *dom-m* N) ∧
{#$i \in\#$ *fst* ‘# *mset* (W' *La*). $i \in\#$ *dom-m* N#} =
clause-to-update La (M, N, D, NE, UE, {#}, {#})⟩ **and**
Hneq2: ⟨⋀*La*. *La*∈#*all-lits-of-mm* (*mset* ‘# *ran-mf* N + (NE + UE)) ⟹
($La \neq L$ ⟶
distinct-watched (W' *La*) ∧
{#$i \in\#$ *fst* ‘# *mset* (W' *La*). $i \in\#$ *dom-m* N#} =

clause-to-update La $(M,\ N,\ D,\ NE,\ UE,\ \{\#\},\ \{\#\}))$›
using *corr* **unfolding** *correct-watching-except.simps W′-def*[*symmetric*]
by *fast+*
have *H1*: ‹*mset ‘# ran-mf* $(N(x1 \hookrightarrow swap\ (N \propto x1)\ i\ xa)) = mset$ *‘# ran-mf N*›
using *dom i-xa distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def dest*!: *multi-member-split intro*!: *image-mset-cong2*)
have *W-W′*: ‹*W*
$(L := (W\ L)[j := (x1,\ x2,\ b)],\ N \propto x1\ !\ xa := W\ (N \propto x1\ !\ xa)\ @\ [(x1,\ L',\ b)]) =$
$W'(N \propto x1\ !\ xa := W\ (N \propto x1\ !\ xa)\ @\ [(x1,\ L',\ b)])$›
unfolding *W′-def*
by *auto*
have *W-W2*: ‹$W\ (N \propto x1\ !\ xa) = W'\ (N \propto x1\ !\ xa)$›
using *L* **unfolding** *W′-def* **by** *auto*
have *H2*: ‹$set\ (swap\ (N \propto x1)\ i\ xa) = set\ (N \propto x1)$›
using *i-xa* **by** *auto*
have [*simp*]:
‹$set\ (fst\ (the\ (if\ x1 = ia\ then\ Some\ (swap\ (N \propto x1)\ i\ xa,\ irred\ N\ x1)\ else\ fmlookup\ N\ ia))) =$
$set\ (fst\ (the\ (fmlookup\ N\ ia)))$› **for** *ia*
using *H2*
by *auto*
have *H3*: ‹$i = x1 \lor i \in\# remove1\text{-}mset\ x1\ (dom\text{-}m\ N) \longleftrightarrow i \in\# dom\text{-}m\ N$› **for** *i*
using *dom* **by** (*auto dest*: *multi-member-split*)
have *set-N-swap-x1*: ‹$set\ (watched\text{-}l\ (swap\ (N \propto x1)\ i\ xa)) = \{N \propto x1\ !\ (1 - i),\ N \propto x1\ !\ xa\}$›
using *i-2 i-xa* ‹$xa \geq 2$› *N-i*
by (*cases* ‹$N \propto x1$›; *cases* ‹$tl\ (N \propto x1)$›; *cases i*; *cases* ‹$i-1$›; *cases xa*)
(*auto simp*: *swap-def split*: *nat.splits*)
have *set-N-x1*: ‹$set\ (watched\text{-}l\ (N \propto x1)) = \{N \propto x1\ !\ (1 - i),\ N \propto x1\ !\ i\}$›
using *i-2 i-xa* ‹$xa \geq 2$› *N-i*
by (*cases i*) (*auto simp*: *swap-def take-2-if*)

have *La-in-notin-swap*: ‹$La \in set\ (watched\text{-}l\ (N \propto x1)) \Longrightarrow$
$La \notin set\ (watched\text{-}l\ (swap\ (N \propto x1)\ i\ xa)) \Longrightarrow La = L$› **for** *La*
using *i-2 i-xa* ‹$xa \geq 2$› *N-i*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)

have *L-notin-swap*: ‹$L \notin set\ (watched\text{-}l\ (swap\ (N \propto x1)\ i\ xa))$›
using *i-2 i-xa* ‹$xa \geq 2$› *N-i*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)
have *N-xa-in-swap*: ‹$N \propto x1\ !\ xa \in set\ (watched\text{-}l\ (swap\ (N \propto x1)\ i\ xa))$›
using *i-2 i-xa* ‹$xa \geq 2$› *N-i*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)
have *H4*: ‹$(i = x1 \longrightarrow K \in set\ (N \propto x1) \land K \neq La) \land (i \in\# remove1\text{-}mset\ x1\ (dom\text{-}m\ N) \longrightarrow K$
$\in set\ (N \propto i) \land K \neq La) \longleftrightarrow$
$(i \in\# dom\text{-}m\ N \longrightarrow K \in set\ (N \propto i) \land K \neq La)$› **for** *i P K La*
using *dom* **by** (*auto dest*: *multi-member-split*)
have [*simp*]: ‹$x1 \notin\# Ab \Longrightarrow$
$\{\#C \in\# Ab.$
$(x1 = C \longrightarrow Q\ C) \land$
$(x1 \neq C \longrightarrow R\ C)\#\} =$
$\{\#C \in\# Ab.\ R\ C\#\}$› **for** *Ab Q R*
by (*auto intro*: *filter-mset-cong*)
have *bin*:
‹*correctly-marked-as-binary N* $(x1,\ x2,\ b)$›
using *Heq*[*of L* ‹$fst\ (W\ L\ !\ w)$› ‹$fst\ (snd\ (W\ L\ !\ w\ ))$› ‹$snd\ (snd\ (W\ L\ !\ w))$›] *j-w w-le dom L′*
by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append L-L*)
have *x1-new*: ‹$x1 \notin fst\ ‘\ set\ (W\ (N \propto x1\ !\ xa))$›

proof (*rule ccontr*)
  assume *H*: ¬ *?thesis*
  have ‹*N* ∝ *x1* ! *xa*
    ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*))›
   using *dom in-clause-in-all-lits-of-m*[*of* ‹*N* ∝ *x1* ! *xa*› ‹*mset* (*N* ∝ *x1*)›] *i-xa*
   by (*auto simp*: *all-lits-of-mm-union ran-m-def all-lits-of-mm-add-mset*
 dest!: *multi-member-split*)
  then have ‹{#*i* ∈# *fst* ‘# *mset* (*W* (*N* ∝ *x1* ! *xa*)). *i* ∈# *dom-m N*#} =
    clause-to-update (*N* ∝ *x1* ! *xa*) (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})›
   using *Hneq*[*of* ‹*N* ∝ *x1* ! *xa*›] *L* **unfolding** *W′-def*
   by *simp*
  then have ‹*x1* ∈# *clause-to-update* (*N* ∝ *x1* ! *xa*) (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})›
   using *H dom* **by** (*metis* (*no-types*, *lifting*) *mem-Collect-eq set-image-mset*
    set-mset-filter set-mset-mset)
  then show *False*
   using *N-xa i-2 i-xa*
   by (*cases i*; *cases* ‹*N* ∝ *x1* ! *xa*›)
    (*auto simp*: *clause-to-update-def take-2-if split*: *if-splits*)
 qed

 let *?N* = ‹*N*(*x1* ↪ *swap* (*N* ∝ *x1*) *i xa*)›
 have ‹*L* ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*)) ⟹ *La* = *L* ⟹
   x ∈ *set* (*take j* (*W L*)) ∨ *x* ∈ *set* (*drop* (*Suc w*) (*W L*)) ⟹
   case x of (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧
     correctly-marked-as-binary ?N (*i*, *K*, *b*)› **for** *La x*
  using *Heq*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le*
  by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append correctly-marked-as-binary.simps*
   split: *if-splits*)
 moreover have ‹*L* ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*)) ⟹ *La* =
L ⟹
   x ∈ *set* (*take j* (*W L*)) ∨ *x* ∈ *set* (*drop* (*Suc w*) (*W L*)) ⟹
   case x of (*i*, *K*, *b*) ⇒*b* ⟶ *i* ∈# *dom-m N*› **for** *La x*
  using *Heq*[*of L* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le*
  by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append correctly-marked-as-binary.simps*
split: *if-splits*)
 moreover have ‹*L* ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*)) ⟹
    La = *L* ⟹
 distinct-watched (*take j* (*W L*) @ *drop* (*Suc w*) (*W L*)) ∧
    {#*i* ∈# *fst* ‘# *mset* (*take j* (*W L*)). *i* ∈# *dom-m N*#} + {#*i* ∈# *fst* ‘# *mset* (*drop* (*Suc w*)
(*W L*)). *i* ∈# *dom-m N*#} =
    clause-to-update L (*M*, *N*(*x1* ↪ *swap* (*N* ∝ *x1*) *i xa*), *D*, *NE*, *UE*, {#}, {#})› **for** *La*
  using *Heq*[*of L x1 x2 b*] *j-w w-le dom L-notin-swap N-xa-in-swap distinct-mset-dom*[*of N*]
  i-xa i-2 assms(*12*)
  by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append set-N-x1 assms*(*11*)
    clause-to-update-def dest!: *multi-member-split split*: *if-splits*
    intro: *filter-mset-cong2*)

 moreover have ‹*La* ∈# *all-lits-of-mm*
      ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*)) ⟹
   La ≠ *L* ⟹
   x ∈ *set* (*if La* = *N* ∝ *x1* ! *xa*
      then W′ (*N* ∝ *x1* ! *xa*) @ [(*x1*, *L′*, *b*)]
      else (*W*(*L* := (*W L*)[*j* := (*x1*, *x2*, *b*)])) *La*) ⟹
   case x of
   (*i*, *K*, *b*) ⇒ *i* ∈# *dom-m ?N* ⟶ *K* ∈ *set* (*?N* ∝ *i*) ∧ *K* ≠ *La* ∧ *correctly-marked-as-binary ?N*
(*i*, *K*, *b*)› **for** *La x*

using *Hneq*[*of La* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le L′ L-neq bin dom*
by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append*
correctly-marked-as-binary.simps split: *if-splits*)
moreover have ‹$La \in\#$ *all-lits-of-mm*
({#*mset* (*fst x*). $x \in\#$ *ran-m N*#} + (*NE* + *UE*)) $\Longrightarrow$
$La \neq L \Longrightarrow$
$x \in$ *set* (*if* $La = N \propto x1 ! xa$
then W' ($N \propto x1 ! xa$) @ [(*x1*, *L′*, *b*)]
else (*W*(*L* := (*W L*)[*j* := (*x1*, *x2*, *b*)])) *La*) $\Longrightarrow$
case x of (*i*, *K*, *b*) $\Rightarrow$ *b* $\longrightarrow$ $i \in\#$ *dom-m N*› **for** *La x*
using *Hneq*[*of La* ‹*fst x*› ‹*fst* (*snd x*)› ‹*snd* (*snd x*)›] *j-w w-le L′ L-neq* ‹*length* ($N \propto x1$) > *2*›
dom
by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append correctly-marked-as-binary.simps*
split: *if-splits*)
moreover have ‹$La \in\#$ *all-lits-of-mm*
({#*mset* (*fst x*). $x \in\#$ *ran-m N*#} + (*NE* + *UE*)) $\Longrightarrow$
$La \neq L \Longrightarrow$ *distinct-watched* ((*W*
(*L* := (*W L*)[*j* := (*x1*, *x2*, *b*)],
$N \propto x1 ! xa$:= *W* ($N \propto x1 ! xa$) @ [(*x1*, *L′*, *b*)])) *La*)› **for** *La x*
using *Hneq*[*of La*] *j-w w-le L′ L-neq* ‹*length* ($N \propto x1$) > *2*›
dom x1-new
by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append correctly-marked-as-binary.simps*
split: *if-splits*)
moreover {
have ‹$N \propto x1 ! xa \notin$ *set* (*watched-l* ($N \propto x1$))›
using *N-xa*
by (*auto simp*: *set-N-x1 set-N-swap-x1*)

then have ‹ $\bigwedge$*Ab Ac La*.
all-lits-of-mm ({#*mset* (*fst x*). $x \in\#$ *ran-m N*#} + (*NE* + *UE*)) = *add-mset L′* (*add-mset* ($N \propto x1 ! xa$) *Ac*) $\Longrightarrow$
dom-m N = *add-mset x1 Ab* $\Longrightarrow$
$N \propto x1 ! xa \neq L \Longrightarrow$
{#$i \in\#$ *fst* ‘# *mset* (*W* ($N \propto x1 ! xa$)). $i = x1 \lor i \in\#$ *Ab*#} =
{#$C \in\#$ *Ab*. $N \propto x1 ! xa \in$ *set* (*watched-l* ($N \propto C$))#} ›
using *Hneq2*[*of* ‹$N \propto x1 ! xa$›] *L-neq* **unfolding** *W-W′ W-W2*
by (*auto simp*: *clause-to-update-def split*: *if-splits*)
then have ‹$La \in\#$ *all-lits-of-mm* ({#*mset* (*fst x*). $x \in\#$ *ran-m N*#} + (*NE* + *UE*)) $\Longrightarrow$
$La \neq L \Longrightarrow$
distinct-watched (W' *La*) $\land$
($x1 \in\#$ *dom-m N* $\longrightarrow$
($La = N \propto x1 ! xa \longrightarrow$
add-mset x1 {#$i \in\#$ *fst* ‘# *mset* (W' ($N \propto x1 ! xa$)). $i \in\#$ *dom-m N*#} =
clause-to-update ($N \propto x1 ! xa$) (*M*, *N*(*x1* $\hookrightarrow$ *swap* ($N \propto x1$) *i xa*), *D*, *NE*, *UE*, {#}, {#})) $\land$
($La \neq N \propto x1 ! xa \longrightarrow$
{#$i \in\#$ *fst* ‘# *mset* (*W La*). $i \in\#$ *dom-m N*#} =
clause-to-update La (*M*, *N*(*x1* $\hookrightarrow$ *swap* ($N \propto x1$) *i xa*), *D*, *NE*, *UE*, {#}, {#}))) $\land$
($x1 \notin\#$ *dom-m N* $\longrightarrow$
($La = N \propto x1 ! xa \longrightarrow$
{#$i \in\#$ *fst* ‘# *mset* (W' ($N \propto x1 ! xa$)). $i \in\#$ *dom-m N*#} =
clause-to-update ($N \propto x1 ! xa$) (*M*, *N*(*x1* $\hookrightarrow$ *swap* ($N \propto x1$) *i xa*), *D*, *NE*, *UE*, {#}, {#})) $\land$
($La \neq N \propto x1 ! xa \longrightarrow$
{#$i \in\#$ *fst* ‘# *mset* (*W La*). $i \in\#$ *dom-m N*#} =
clause-to-update La (*M*, *N*(*x1* $\hookrightarrow$ *swap* ($N \propto x1$) *i xa*), *D*, *NE*, *UE*, {#}, {#})))› **for** *La*
using *Hneq2*[*of La*] *j-w w-le L′ dom distinct-mset-dom*[*of N*] *L-notin-swap N-xa-in-swap L-neq*
by (*auto simp*: *take-Suc-conv-app-nth W′-def list-update-append clause-to-update-def*

add-mset-eq-add-mset set-N-x1 set-N-swap-x1 assms(*11*) *N-i*
dest!: *multi-member-split La-in-notin-swap*
split: *if-splits*
intro: *image-mset-cong2 intro*: *filter-mset-cong2*)
}
ultimately show *?thesis*
using *L j-w*
unfolding *correct-watching-except.simps H1 W′-def*[*symmetric*] *W-W′ H2 W-W2 H4 H3*
by (*intro conjI impI ballI*)
(*simp-all add*: *L′ W-W′ W-W2 H3 H4 drop-map*)
qed

definition *unit-propagation-inner-loop-wl-loop-pre* **where**
‹*unit-propagation-inner-loop-wl-loop-pre L* = (λ(*j*, *w*, *S*).
w < *length* (*watched-by S L*) ∧ *j* ≤ *w* ∧
unit-propagation-inner-loop-wl-loop-inv L (*j*, *w*, *S*))›

It was too hard to align the programi unto a refinable form directly.

definition *unit-propagation-inner-loop-body-wl-int* :: ‹*′v literal* ⇒ *nat* ⇒ *nat* ⇒ *′v twl-st-wl* ⇒ (*nat* × *nat* × *′v twl-st-wl*) *nres*› **where**
‹*unit-propagation-inner-loop-body-wl-int L j w S* = *do* {
  ASSERT(*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*));
  let (*C*, *K*, *b*) = (*watched-by S L*) ! *w*;
  let S = *keep-watch L j w S*;
  ASSERT(*unit-prop-body-wl-inv S j w L*);
  let val-K = *polarity* (*get-trail-wl S*) *K*;
  if val-K = *Some True*
  then RETURN (*j*+*1*, *w*+*1*, *S*)
  else do { — Now the costly operations:
    if C ∉# *dom-m* (*get-clauses-wl S*)
    then RETURN (*j*, *w*+*1*, *S*)
    else do {
      let i = (*if* ((*get-clauses-wl S*)∝*C*) ! *0* = *L then 0 else 1*);
      let L′ = ((*get-clauses-wl S*)∝*C*) ! (*1* − *i*);
      let val-L′ = *polarity* (*get-trail-wl S*) *L′*;
      if val-L′ = *Some True*
      then update-blit-wl L C b j w L′ S
      else do {
        f ← *find-unwatched-l* (*get-trail-wl S*) (*get-clauses-wl S* ∝*C*);
        ASSERT (*unit-prop-body-wl-find-unwatched-inv f C S*);
        case f of
          None ⇒ *do* {
            if val-L′ = *Some False*
            then do {*RETURN* (*j*+*1*, *w*+*1*, *set-conflict-wl* (*get-clauses-wl S* ∝ *C*) *S*)}
            else do {*RETURN* (*j*+*1*, *w*+*1*, *propagate-lit-wl-general L′ C i S*)}
          }
        | *Some f* ⇒ *do* {
            let K = *get-clauses-wl S* ∝ *C* ! *f*;
            let val-L′ = *polarity* (*get-trail-wl S*) *K*;
            if val-L′ = *Some True*
            then update-blit-wl L C b j w K S
            else update-clause-wl L C b j w i f S
          }
      }
    }
  }

}⟩

definition *propagate-proper-bin-case* **where**
⟨*propagate-proper-bin-case* L L' S C $\longleftrightarrow$
$C \in\#$ *dom-m* (*get-clauses-wl* S) $\wedge$ *length* ((*get-clauses-wl* S)$\propto C$) = *2* $\wedge$
set (*get-clauses-wl* $S \propto C$) = $\{L, L'\}$ $\wedge$ $L \neq L'$⟩

definition *unit-propagation-inner-loop-body-wl* :: ⟨$'v$ *literal* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'v$ *twl-st-wl* $\Rightarrow$
(*nat* $\times$ *nat* $\times$ $'v$ *twl-st-wl*) *nres*⟩ **where**
⟨*unit-propagation-inner-loop-body-wl* L j w S = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre* L (j, w, S));
let (C, K, b) = (*watched-by* S L) ! w;
let S = *keep-watch* L j w S;
ASSERT(*unit-prop-body-wl-inv* S j w L);
let val-K = *polarity* (*get-trail-wl* S) K;
if val-K = *Some True*
then RETURN (j+*1*, w+*1*, S)
else do {
if b *then do* {
ASSERT(*propagate-proper-bin-case* L K S C);
if val-K = *Some False*
then RETURN (j+*1*, w+*1*, *set-conflict-wl* (*get-clauses-wl* $S \propto C$) S)
else do { — This is non-optimal (memory access: relax invariant!):
let i = (*if* ((*get-clauses-wl* S)$\propto C$) ! *0* = L *then 0 else 1*);
RETURN (j+*1*, w+*1*, *propagate-lit-wl-bin* K C i S)}
} — Now the costly operations:
else if $C \notin\#$ *dom-m* (*get-clauses-wl* S)
then RETURN (j, w+*1*, S)
else do {
let i = (*if* ((*get-clauses-wl* S)$\propto C$) ! *0* = L *then 0 else 1*);
let L' = ((*get-clauses-wl* S)$\propto C$) ! (*1* − i);
let val-L' = *polarity* (*get-trail-wl* S) L';
if val-L' = *Some True*
then update-blit-wl L C b j w L' S
else do {
$f \leftarrow$ *find-unwatched-l* (*get-trail-wl* S) (*get-clauses-wl* S $\propto C$);
ASSERT (*unit-prop-body-wl-find-unwatched-inv* f C S);
case f *of*
None $\Rightarrow$ *do* {
if val-L' = *Some False*
then do {*RETURN* (j+*1*, w+*1*, *set-conflict-wl* (*get-clauses-wl* S $\propto$ C) S)}
else do {*RETURN* (j+*1*, w+*1*, *propagate-lit-wl* L' C i S)}
}
| *Some* f $\Rightarrow$ *do* {
let K = *get-clauses-wl* S $\propto$ C ! f;
let val-L' = *polarity* (*get-trail-wl* S) K;
if val-L' = *Some True*
then update-blit-wl L C b j w K S
else update-clause-wl L C b j w i f S
}
}
}
}
}⟩

lemma [*twl-st-wl*]: ‹*get-clauses-wl* (*keep-watch* L j w S) = *get-clauses-wl* S›
  by (*cases* S) (*auto simp*: *keep-watch-def*)

lemma *unit-propagation-inner-loop-body-wl-int-alt-def*:
 ‹*unit-propagation-inner-loop-body-wl-int* L j w S = *do* {
      ASSERT(*unit-propagation-inner-loop-wl-loop-pre* L (j, w, S));
      let (C, K, b) = (*watched-by* S L) ! w;
      let b' = (C $\notin\#$ *dom-m* (*get-clauses-wl* S));
      if b' *then do* {
        let S = *keep-watch* L j w S;
        ASSERT(*unit-prop-body-wl-inv* S j w L);
        let K = K;
        let val-K = *polarity* (*get-trail-wl* S) K *in*
        if val-K = *Some True*
        then RETURN (j+*1*, w+*1*, S)
        else — Now the costly operations:
          RETURN (j, w+*1*, S)
      }
      else do {
        let S' = *keep-watch* L j w S;
        ASSERT(*unit-prop-body-wl-inv* S' j w L);
        K ← *SPEC*((=) K);
        let val-K = *polarity* (*get-trail-wl* S') K *in*
        if val-K = *Some True*
        then RETURN (j+*1*, w+*1*, S')
        else do { — Now the costly operations:
          let i = (*if* ((*get-clauses-wl* S')$\propto C$) ! *0* = L *then 0 else 1*);
          let L' = ((*get-clauses-wl* S')$\propto C$) ! (*1* − i);
          let val-L' = *polarity* (*get-trail-wl* S') L';
          if val-L' = *Some True*
          then update-blit-wl L C b j w L' S'
          else do {
            f ← *find-unwatched-l* (*get-trail-wl* S') (*get-clauses-wl* $S'\propto C$);
            ASSERT (*unit-prop-body-wl-find-unwatched-inv* f C S');
            case f *of*
              None ⇒ *do* {
                if val-L' = *Some False*
                then do {*RETURN* (j+*1*, w+*1*, *set-conflict-wl* (*get-clauses-wl* S' $\propto$ C) S')}
                else do {*RETURN* (j+*1*, w+*1*, *propagate-lit-wl-general* L' C i S')}
              }
            | *Some* f ⇒ *do* {
                let K = *get-clauses-wl* S' $\propto$ C ! f;
                let val-L' = *polarity* (*get-trail-wl* S') K;
                if val-L' = *Some True*
                then update-blit-wl L C b j w K S'
                else update-clause-wl L C b j w i f S'
              }
          }
        }
      }
   }›
proof −

We first define an intermediate step where both then and else branches are the same.

  have E: ‹*unit-propagation-inner-loop-body-wl-int* L j w S = *do* {

```
ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
let (C, K, b) = (watched-by S L) ! w;
let b′ = (C ∉# dom-m (get-clauses-wl S));
if b′ then do {
  let S = keep-watch L j w S;
  ASSERT(unit-prop-body-wl-inv S j w L);
  let K = K;
  let val-K = polarity (get-trail-wl S) K in
  if val-K = Some True
  then RETURN (j+1, w+1, S)
  else do { — Now the costly operations:
    if b′
    then RETURN (j, w+1, S)
    else do {
      let i = (if ((get-clauses-wl S)∝C) ! 0 = L then 0 else 1);
      let L′ = ((get-clauses-wl S)∝C) ! (1 − i);
      let val-L′ = polarity (get-trail-wl S) L′;
      if val-L′ = Some True
      then update-blit-wl L C b j w L′ S
      else do {
        f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S ∝C);
        ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
        case f of
          None ⇒ do {
            if val-L′ = Some False
            then do {RETURN (j+1, w+1, set-conflict-wl (get-clauses-wl S ∝ C) S)}
            else do {RETURN (j+1, w+1, propagate-lit-wl-general L′ C i S)}
          }
        | Some f ⇒ do {
          let K = get-clauses-wl S ∝ C ! f;
          let val-L′ = polarity (get-trail-wl S) K;
          if val-L′ = Some True
          then update-blit-wl L C b j w K S
          else update-clause-wl L C b j w i f S
          }
      }
    }
  }
}
else do {
  let S′ = keep-watch L j w S;
  ASSERT(unit-prop-body-wl-inv S′ j w L);
  K ← SPEC((=) K);
  let val-K = polarity (get-trail-wl S′) K in
  if val-K = Some True
  then RETURN (j+1, w+1, S′)
  else do { — Now the costly operations:
    if b′
    then RETURN (j, w+1, S′)
    else do {
      let i = (if ((get-clauses-wl S′)∝C) ! 0 = L then 0 else 1);
      let L′ = ((get-clauses-wl S′)∝C) ! (1 − i);
      let val-L′ = polarity (get-trail-wl S′) L′;
      if val-L′ = Some True
      then update-blit-wl L C b j w L′ S′
      else do {
```

$f \leftarrow$ *find-unwatched-l* (*get-trail-wl* S') (*get-clauses-wl* $S' \propto C$);
ASSERT (*unit-prop-body-wl-find-unwatched-inv* f C S');
case f *of*
None $\Rightarrow$ *do* {
if *val-L'* = *Some False*
then do {*RETURN* (j+1, w+1, *set-conflict-wl* (*get-clauses-wl* $S' \propto C$) S')}
else do {*RETURN* (j+1, w+1, *propagate-lit-wl-general* L' C i S')}
}
| *Some* f $\Rightarrow$ *do* {
let K = *get-clauses-wl* $S' \propto C$! f;
let *val-L'* = *polarity* (*get-trail-wl* S') K;
if *val-L'* = *Some True*
then *update-blit-wl* L C b j w K S'
else *update-clause-wl* L C b j w i f S'
}
}
}
}
}
}›
(**is** ‹- = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre* L (j, w, S));
let (C, K, b) = (*watched-by* S L) ! w;
let b' = ($C \notin\#$ *dom-m* (*get-clauses-wl* S));
if b' *then do* {
?P C K b b'
}
else do {
?Q C K b b'
}
}›)
unfolding *unit-propagation-inner-loop-body-wl-int-def if-not-swap bind-to-let-conv SPEC-eq-is-RETURN twl-st-wl*
unfolding *Let-def if-not-swap bind-to-let-conv SPEC-eq-is-RETURN twl-st-wl*
apply (*subst if-cancel*)
apply (*intro bind-cong-nres case-prod-cong if-cong*[*OF refl*] *refl*)
done
show *?thesis*
unfolding *E*
apply (*subst if-replace-cond*[*of* - ‹*?P* - - -›])
unfolding *if-True if-False*
apply *auto*
done
qed

1.4.3 The Functions

Inner Loop

lemma *clause-to-update-mapsto-upd-If*:
assumes
i: ‹$i \in\#$ *dom-m* N›
shows
‹*clause-to-update* L (M, N($i \hookrightarrow C'$), C, NE, UE, WS, Q) =
(*if* $L \in$ *set* (*watched-l* C')

then add-mset i (remove1-mset i (clause-to-update L (M, N, C, NE, UE, WS, Q)))
else remove1-mset i (clause-to-update L (M, N, C, NE, UE, WS, Q)))⟩
proof −
define D' **where** ⟨$D' = dom\text{-}m\ N - \{\#i\#\}$⟩
then have [*simp*]: ⟨$dom\text{-}m\ N = add\text{-}mset\ i\ D'$⟩
using *assms* **by** (*simp add*: *mset-set.remove*)
have [*simp*]: ⟨$i \notin\# D'$⟩
using *assms distinct-mset-dom*[*of N*] **unfolding** D'*-def* **by** *auto*

have ⟨$\{\#C \in\# D'.$
$(i = C \longrightarrow L \in set\ (watched\text{-}l\ C')) \wedge$
$(i \neq C \longrightarrow L \in set\ (watched\text{-}l\ (N \propto C)))\#\} =$
$\{\#C \in\# D'.\ L \in set\ (watched\text{-}l\ (N \propto C))\#\}$⟩
by (*rule filter-mset-cong2*) *auto*
then show *?thesis*
unfolding *clause-to-update-def*
by *auto*
qed

lemma *unit-propagation-inner-loop-body-l-with-skip-alt-def*:
⟨*unit-propagation-inner-loop-body-l-with-skip L* $(S', n) = do$ {
ASSERT (*clauses-to-update-l* $S' \neq \{\#\} \vee 0 < n$);
ASSERT (*unit-propagation-inner-loop-l-inv L* (S', n));
$b \leftarrow SPEC\ (\lambda b.\ (b \longrightarrow 0 < n) \wedge (\neg\ b \longrightarrow clauses\text{-}to\text{-}update\text{-}l\ S' \neq \{\#\}))$;
if $\neg\ b$
then do {
ASSERT (*clauses-to-update-l* $S' \neq \{\#\}$);
$X2 \leftarrow select\text{-}from\text{-}clauses\text{-}to\text{-}update\ S'$;
ASSERT (*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*));
$x \leftarrow SPEC\ (\lambda K.\ K \in set\ (get\text{-}clauses\text{-}l\ (fst\ X2) \propto snd\ X2))$;
let v = polarity (*get-trail-l* (*fst X2*)) *x*;
if v = Some True then let T = fst X2 in RETURN (*T*, *if get-conflict-l T = None then n else 0*)
else let v = if get-clauses-l (*fst X2*) $\propto$ *snd X2 ! 0 = L then 0 else 1*;
va = get-clauses-l (*fst X2*) $\propto$ *snd X2 !* $(1 - v)$; *vaa = polarity* (*get-trail-l* (*fst X2*)) *va*
in
if vaa = Some True
then let T = fst X2 in RETURN (*T*, *if get-conflict-l T = None then n else 0*)
else do {
$x \leftarrow$ *find-unwatched-l* (*get-trail-l* (*fst X2*)) (*get-clauses-l* (*fst X2*) $\propto$ *snd X2*);
case x of
None ⇒
if vaa = Some False
then let T = set-conflict-l (*get-clauses-l* (*fst X2*) $\propto$ *snd X2*) (*fst X2*)
in RETURN (*T*, *if get-conflict-l T = None then n else 0*)
else let T = propagate-lit-l va (*snd X2*) *v* (*fst X2*)
in RETURN (*T*, *if get-conflict-l T = None then n else 0*)
| *Some a* ⇒ *do* {
$x \leftarrow$ *ASSERT* (*a < length* (*get-clauses-l* (*fst X2*) $\propto$ *snd X2*));
let K = (*get-clauses-l* (*fst X2*) $\propto$ (*snd X2*))!*a*;
let val-K = polarity (*get-trail-l* (*fst X2*)) *K*;
if val-K = Some True
then let T = fst X2 in RETURN (*T*, *if get-conflict-l T = None then n else 0*)
else do {
$T \leftarrow$ *update-clause-l* (*snd X2*) *v a* (*fst X2*);
RETURN (*T*, *if get-conflict-l T = None then n else 0*)
}

}
}
}
else RETURN $(S', n - 1)$
})

proof −
have *remove-pairs*: ⟨*do* {(*x2*, *x2'*) ← (*b0* :: - *nres*); *F x2 x2'*} =
do {*X2* ← *b0*; *F* (*fst X2*) (*snd X2*)}⟩ **for** *T a0 b0 a b c f t F*
by (*meson case-prod-unfold*)

have *H1*: ⟨*do* {*T* ← *do* {*x* ← *a* ; *b x*}; *RETURN* (*f T*)} =
do {*x* ← *a*; *T* ← *b x*; *RETURN* (*f T*)}⟩ **for** *T a0 b0 a b c f t*
by *auto*
have *H2*: ⟨*do*{*T* ← *let v* = *val in g v*; (*f T* :: - *nres*)} =
do{*let v* = *val*; *T* ← *g v*; *f T*} ⟩ **for** *g f T val*
by *auto*
have *H3*: ⟨*do*{*T* ← *if b then g else g'*; (*f T* :: - *nres*)} =
(*if b then do*{*T* ← *g*; *f T*} *else do*{*T* ← *g'*; *f T*}) ⟩ **for** *g g' f T b*
by *auto*
have *H4*: ⟨*do*{*T* ← *case x of None* ⇒ *g* | *Some a* ⇒ *g' a*; (*f T* :: - *nres*)} =
(*case x of None* ⇒ *do*{*T* ← *g*; *f T*} | *Some a* ⇒ *do*{*T* ← *g' a*; *f T*}) ⟩ **for** *g g' f T b x*
by (*cases x*) *auto*
show *?thesis*
unfolding *unit-propagation-inner-loop-body-l-with-skip-def prod.case*
unit-propagation-inner-loop-body-l-def remove-pairs
unfolding *H1 H2 H3 H4 bind-to-let-conv*
by *simp*
qed

lemma *keep-watch-st-wl*[*twl-st-wl*]:
⟨*get-unit-clauses-wl* (*keep-watch L j w S*) = *get-unit-clauses-wl S*⟩
⟨*get-conflict-wl* (*keep-watch L j w S*) = *get-conflict-wl S*⟩
⟨*get-trail-wl* (*keep-watch L j w S*) = *get-trail-wl S*⟩
by (*cases S*; *auto simp*: *keep-watch-def*; *fail*)+
declare *twl-st-wl*[*simp*]

lemma *correct-watching-except-correct-watching-except-propagate-lit-wl*:
assumes
corr: ⟨*correct-watching-except j w L S*⟩ **and**
i-le: ⟨*Suc 0* < *length* (*get-clauses-wl S* ∝ *C*)⟩ **and**
C: ⟨*C* ∈# *dom-m* (*get-clauses-wl S*)⟩
shows ⟨*correct-watching-except j w L* (*propagate-lit-wl-general L' C i S*)⟩
proof −
obtain *M N D NE UE Q W* **where** *S*: ⟨*S* = (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)⟩ **by** (*cases S*)
have
Hneq: ⟨⋀*La*. *La*∈#*all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*)) ⟹
La ≠ *L* ⟹
(∀(*i*, *K*, *b*)∈#*mset* (*W La*). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *La* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)) ∧
(∀(*i*, *K*, *b*)∈#*mset* (*W La*). *b* ⟶ *i* ∈# *dom-m N*) ∧
{#*i* ∈# *fst* '# *mset* (*W La*). *i* ∈# *dom-m N*#} = *clause-to-update La* (*M*, *N*, *D*, *NE*, *UE*,
{#}, {#})⟩ **and**
Heq: ⟨⋀*La*. *La*∈#*all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*)) ⟹
La = *L* ⟹
(∀(*i*, *K*, *b*)∈#*mset* (*take j* (*W La*) @ *drop w* (*W La*)). *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧
K ≠ *La* ∧

correctly-marked-as-binary N $(i,\ K,\ b)) \wedge$
$(\forall (i,\ K,\ b) \in\# mset$ $(take\ j\ (W\ La)\ @\ drop\ w\ (W\ La)).\ b \longrightarrow i \in\# \ dom\text{-}m\ N) \wedge$
$\{\#i \in\# \ fst\ `\#\ mset\ (take\ j\ (W\ La)\ @\ drop\ w\ (W\ La)).\ i \in\# \ dom\text{-}m\ N\#\} =$
clause-to-update La $(M,\ N,\ D,\ NE,\ UE,\ \{\#\},\ \{\#\})$›
using *corr* **unfolding** *S correct-watching-except.simps*
by *fast+*
let *?N* = ‹*if length* $(N \propto C) > 2$ *then* $N(C \hookrightarrow swap\ (N \propto C)\ 0\ (Suc\ 0\ -\ i))$ *else* N›

have ‹*Suc* $0 - i <$ *length* $(N \propto C)$› **and** ‹$0 <$ *length* $(N \propto C)$›
using *i-le*
by (*auto simp*: *S*)
then have [*simp*]: ‹*mset* (*swap* $(N \propto C)$ *0* (*Suc* $0 - i$)) = *mset* $(N \propto C)$›
by (*auto simp*: *S*)
have *H1*[*simp*]: ‹$\{\#mset\ (fst\ x).\ x \in\# \ ran\text{-}m\ ?N\#\} =$
$\{\#mset\ (fst\ x).\ x \in\# \ ran\text{-}m\ N\#\}$›
using *C*
by (*auto dest*!: *multi-member-split simp*: *ran-m-def S*
intro!: *image-mset-cong*)

have *H2*: ‹*mset* ‘# *ran-mf ?N* = *mset* ‘# *ran-mf N*›
using *H1* **by** *auto*
have *H3*: ‹*dom-m ?N* = *dom-m N*›
using *C* **by** (*auto simp*: *S*)
have *H4*: ‹*set* $(?N \propto ia)$ =
set $(N \propto ia)$› **for** *ia*
using *i-le*
by (*cases* ‹$C = ia$›) (*auto simp*: *S*)
have *H5*: ‹*set* (*watched-l* $(?N \propto ia)$) = *set* (*watched-l* $(N \propto ia)$)› **for** *ia*
using *i-le*
by (*cases* ‹$C = ia$›; *cases i*; *cases* ‹$N \propto ia$›; *cases* ‹*tl* $(N \propto ia)$›) (*auto simp*: *S swap-def*)
have [*iff*]: ‹*correctly-marked-as-binary N C'* $\longleftrightarrow$ *correctly-marked-as-binary ?N C'*› **for** *C' ia*
by (*cases C'*)
(*auto simp*: *correctly-marked-as-binary.simps*)
show *?thesis*
using *corr*
unfolding *S propagate-lit-wl-general-def prod.simps correct-watching-except.simps Let-def*
H1 H2 H3 H4 clause-to-update-def get-clauses-l.simps H5
by *fast*
qed

lemma *unit-propagation-inner-loop-body-wl-int-alt-def2*:
‹*unit-propagation-inner-loop-body-wl-int L j w S* = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*));
let (*C*, *K*, *b*) = (*watched-by S L*) ! *w*;
let S = *keep-watch L j w S*;
ASSERT(*unit-prop-body-wl-inv S j w L*);
let val-K = *polarity* (*get-trail-wl S*) *K*;
if val-K = *Some True*
then RETURN (*j+1*, *w+1*, *S*)
else do { — Now the costly operations:
if b then
if $C \notin\#$ *dom-m* (*get-clauses-wl S*)
then RETURN (*j*, *w+1*, *S*)
else do {
let i = (*if* ((*get-clauses-wl S*)$\propto$*C*) ! *0* = *L then 0 else 1*);

```
          let L' = ((get-clauses-wl S)∝C) ! (1 − i);
          let val-L' = polarity (get-trail-wl S) L';
          if val-L' = Some True
          then update-blit-wl L C b j w L' S
          else do {
            f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S ∝C);
            ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
            case f of
              None ⇒ do {
                if val-L' = Some False
                then do {RETURN (j+1, w+1, set-conflict-wl (get-clauses-wl S ∝ C) S)}
                else do {RETURN (j+1, w+1, propagate-lit-wl-general L' C i S)}
              }
            | Some f ⇒ do {
                let K = get-clauses-wl S ∝ C ! f;
                let val-L' = polarity (get-trail-wl S) K;
                if val-L' = Some True
                then update-blit-wl L C b j w K S
                else update-clause-wl L C b j w i f S
              }
          }
        }
      else
        if C ∉# dom-m (get-clauses-wl S)
        then RETURN (j, w+1, S)
        else do {
          let i = (if ((get-clauses-wl S)∝C) ! 0 = L then 0 else 1);
          let L' = ((get-clauses-wl S)∝C) ! (1 − i);
          let val-L' = polarity (get-trail-wl S) L';
          if val-L' = Some True
          then update-blit-wl L C b j w L' S
          else do {
            f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S ∝C);
            ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
            case f of
              None ⇒ do {
                if val-L' = Some False
                then do {RETURN (j+1, w+1, set-conflict-wl (get-clauses-wl S ∝ C) S)}
                else do {RETURN (j+1, w+1, propagate-lit-wl-general L' C i S)}
              }
            | Some f ⇒ do {
                let K = get-clauses-wl S ∝ C ! f;
                let val-L' = polarity (get-trail-wl S) K;
                if val-L' = Some True
                then update-blit-wl L C b j w K S
                else update-clause-wl L C b j w i f S
              }
          }
        }
    }
  }›
```
unfolding *unit-propagation-inner-loop-body-wl-int-def if-not-swap bind-to-let-conv SPEC-eq-is-RETURN twl-st-wl*
unfolding *Let-def if-not-swap bind-to-let-conv SPEC-eq-is-RETURN twl-st-wl*
apply (*subst if-cancel*)

apply (*intro bind-cong-nres case-prod-cong if-cong*[*OF refl*] *refl*)
done

lemma *unit-propagation-inner-loop-body-wl-alt-def*:

```
‹unit-propagation-inner-loop-body-wl L j w S = do {
    ASSERT(unit-propagation-inner-loop-wl-loop-pre L (j, w, S));
    let (C, K, b) = (watched-by S L) ! w;
    let S = keep-watch L j w S;
    ASSERT(unit-prop-body-wl-inv S j w L);
    let val-K = polarity (get-trail-wl S) K;
    if val-K = Some True
    then RETURN (j+1, w+1, S)
    else do {
      if b then do {
        if False
        then RETURN (j, w+1, S)
        else
          if False — val-L' = Some True
          then RETURN (j, w+1, S)
          else do {
            f ← RETURN (None :: nat option);
            case f of
             None ⇒ do {
               ASSERT(propagate-proper-bin-case L K S C);
               if val-K = Some False
               then RETURN (j+1, w+1, set-conflict-wl (get-clauses-wl S ∝ C) S)
               else do {
                 let i = (if ((get-clauses-wl S)∝C) ! 0 = L then 0 else 1);
                 RETURN (j+1, w+1, propagate-lit-wl-bin K C i S)}
             }
           | - ⇒ RETURN (j, w+1, S)
          }
      } — Now the costly operations:
      else if C ∉# dom-m (get-clauses-wl S)
      then RETURN (j, w+1, S)
      else do {
        let i = (if ((get-clauses-wl S)∝C) ! 0 = L then 0 else 1);
        let L' = ((get-clauses-wl S)∝C) ! (1 − i);
        let val-L' = polarity (get-trail-wl S) L';
        if val-L' = Some True
        then update-blit-wl L C b j w L' S
        else do {
          f ← find-unwatched-l (get-trail-wl S) (get-clauses-wl S ∝C);
          ASSERT (unit-prop-body-wl-find-unwatched-inv f C S);
          case f of
            None ⇒ do {
              if val-L' = Some False
              then do {RETURN (j+1, w+1, set-conflict-wl (get-clauses-wl S ∝ C) S)}
              else do {RETURN (j+1, w+1, propagate-lit-wl L' C i S)}
            }
          | Some f ⇒ do {
              let K = get-clauses-wl S ∝ C ! f;
              let val-L' = polarity (get-trail-wl S) K;
              if val-L' = Some True
              then update-blit-wl L C b j w K S
              else update-clause-wl L C b j w i f S
```

}
}
}
}
}⟩
unfolding *unit-propagation-inner-loop-body-wl-def if-not-swap bind-to-let-conv SPEC-eq-is-RETURN twl-st-wl*
unfolding *Let-def if-not-swap bind-to-let-conv SPEC-eq-is-RETURN twl-st-wl if-False*
apply (*intro bind-cong-nres case-prod-cong if-cong*[*OF refl*] *refl*)
apply *auto*
done

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

defines
[*simp*]: ⟨$C'' \equiv$ *get-clauses-l* $S' \propto C'$⟩
assumes
S-S': ⟨(S, S') $\in$ *state-wl-l* (*Some* (L, w))⟩ **and**
w-le: ⟨$w <$ *length* (*watched-by* S L)⟩ **and**
j-w: ⟨$j \leq w$⟩ **and**
corr-w: ⟨*correct-watching-except* j w L S⟩ **and**
inner-loop-inv: ⟨*unit-propagation-inner-loop-wl-loop-inv* L (j, w, S)⟩ **and**
n: ⟨$n =$ *size* (*filter-mset* (λ(i, -). $i \notin\#$ *dom-m* (*get-clauses-wl* S)) (*mset* (*drop* w (*watched-by* S L))))⟩
and
confl-S: ⟨*get-conflict-wl* S = *None*⟩
shows *unit-propagation-inner-loop-body-wl-wl-int*: ⟨*unit-propagation-inner-loop-body-wl* L j w S $\leq$ $\Downarrow$ *Id* (*unit-propagation-inner-loop-body-wl-int* L j w S)⟩
proof –
obtain *bL bin* **where** *SLw*: ⟨*watched-by* S L ! w = (C', *bL*, *bin*)⟩
using C'*-def* **by** (*cases* ⟨*watched-by* S L ! w⟩) *auto*

define i :: *nat* **where**
⟨$i \equiv$ (*if get-clauses-wl* $S \propto C'$! $0 = L$ *then* 0 *else* 1)⟩

have
l-wl-inv: ⟨*unit-prop-body-wl-inv* S j w L⟩ (**is** *?inv*) **and**
clause-ge-0: ⟨$0 <$ *length* (*get-clauses-l* $T \propto C'$)⟩ (**is** *?ge*) **and**
L-def: ⟨*defined-lit* (*get-trail-wl* S) L⟩ ⟨$-L \in$ *lits-of-l* (*get-trail-wl* S)⟩
⟨$L \notin$ *lits-of-l* (*get-trail-wl* S)⟩ (**is** *?L-def*) **and**
i-le: ⟨$i <$ *length* (*get-clauses-wl* $S \propto C'$)⟩ (**is** *?i-le*) **and**
i-le2: ⟨$1-i <$ *length* (*get-clauses-wl* $S \propto C'$)⟩ (**is** *?i-le2*) **and**
C'*-dom*: ⟨$C' \in\#$ *dom-m* (*get-clauses-l* T)⟩ (**is** *?C'-dom*) **and**
L-watched: ⟨$L \in$ *set* (*watched-l* (*get-clauses-l* $T \propto C'$))⟩ (**is** *?L-w*) **and**
dist-clss: ⟨*distinct-mset-mset* (*mset* ‘# *ran-mf* (*get-clauses-wl* S))⟩ **and**
confl: ⟨*get-conflict-l* T = *None*⟩ (**is** *?confl*) **and**
alien-L:
⟨$L \in\#$ *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl* S) + *get-unit-init-clss-wl* S)⟩
(**is** *?alien*) **and**
alien-L':
⟨$L \in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl* S) + *get-unit-clauses-wl* S)⟩
(**is** *?alien'*) **and**

alien-L'':
⟨*L* ∈# *all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*)⟩
(**is** *?alien''*) **and**
correctly-marked-as-binary: ⟨*correctly-marked-as-binary* (*get-clauses-wl S*) (*C'*, *bL*, *bin*)⟩
if
⟨*unit-propagation-inner-loop-body-l-inv L C' T*⟩
proof −
have ⟨*unit-propagation-inner-loop-body-l-inv L C' T*⟩
using *that* **unfolding** *unit-prop-body-wl-inv-def* **by** *fast+*
then obtain *T'* **where**
T-T': ⟨(*set-clauses-to-update-l* (*clauses-to-update-l T* + {#*C'*#}) *T*, *T'*) ∈ *twl-st-l* (*Some L*)⟩ **and**
struct-invs: ⟨*twl-struct-invs T'*⟩ **and**
⟨*twl-stgy-invs T'*⟩ **and**
C'-dom: ⟨*C'* ∈# *dom-m* (*get-clauses-l T*)⟩ **and**
⟨*0* < *C'*⟩ **and**
ge-0: ⟨*0* < *length* (*get-clauses-l T* ∝ *C'*)⟩ **and**
⟨*no-dup* (*get-trail-l T*)⟩ **and**
i-le: ⟨(*if get-clauses-l T* ∝ *C'* ! *0* = *L then 0 else 1*)
< *length* (*get-clauses-l T* ∝ *C'*)⟩ **and**
i-le2: ⟨*1* − (*if get-clauses-l T* ∝ *C'* ! *0* = *L then 0 else 1*)
< *length* (*get-clauses-l T* ∝ *C'*)⟩ **and**
L-watched: ⟨*L* ∈ *set* (*watched-l* (*get-clauses-l T* ∝ *C'*))⟩ **and**
confl: ⟨*get-conflict-l T* = *None*⟩
unfolding *unit-propagation-inner-loop-body-l-inv-def* **by** *blast*
show *?i-le* **and** *?C'-dom* **and** *?L-w* **and** *?i-le2*
using *S-S' i-le C'-dom L-watched i-le2* **unfolding** *i-def* **by** *auto*
have
alien: ⟨$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of T'*)⟩ **and**
dup: ⟨*no-duplicate-queued T'*⟩ **and**
lev: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* ($state_W$*-of T'*)⟩ **and**
dist: ⟨$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state* ($state_W$*-of T'*)⟩
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *blast+*
have *n-d*: ⟨*no-dup* (*trail* ($state_W$*-of T'*))⟩
using *lev* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** *auto*
have *1*: ⟨*C* ∈# *clauses-to-update T'* ⟹
add-mset (*fst C*) (*literals-to-update T'*) ⊆#
uminus '# *lit-of* '# *mset* (*get-trail T'*)⟩ **for** *C*
using *dup* **unfolding** *no-duplicate-queued-alt-def*
by *blast*
have *H*: ⟨(*L*, *twl-clause-of C''*) ∈# *clauses-to-update T'*⟩
using *twl-st-l*(*5*)[*OF T-T'*]
by (*auto simp*: *twl-st-l*)
have *uL-M*: ⟨−*L* ∈ *lits-of-l* (*get-trail T'*)⟩
using *mset-le-add-mset-decr-left2*[*OF 1*[*OF H*]]
by (*auto simp*: *lits-of-def*)
then show ⟨*defined-lit* (*get-trail-wl S*) *L*⟩ ⟨−*L* ∈ *lits-of-l* (*get-trail-wl S*)⟩
⟨*L* ∉ *lits-of-l* (*get-trail-wl S*)⟩
using *S-S' T-T' n-d* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l twl-st*
dest: *no-dup-consistentD*)
show *L*: *?alien*
using *alien uL-M twl-st-l*(*1*−*8*)[*OF T-T'*] *S-S'*
init-clss-state-to-l[*OF T-T'*]
unit-init-clauses-get-unit-init-clauses-l[*OF T-T'*]
unfolding $cdcl_W$*-restart-mset.no-strange-atm-def*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff twl-st-wl twl-st twl-st-l*)

then show *alien′*: *?alien′*
  apply (*rule set-rev-mp*)
  apply (*rule all-lits-of-mm-mono*)
  by (*cases S*) *auto*
show *?alien′′*
  using *L*
  apply (*rule set-rev-mp*)
  apply (*rule all-lits-of-mm-mono*)
  by (*cases S*) *auto*
then have *l-wl-inv*: ‹$(S, S') \in$ *state-wl-l* (*Some* (*L*, *w*)) ∧
    unit-propagation-inner-loop-body-l-inv L (*fst* (*watched-by S L* ! *w*))
    (*remove-one-lit-from-wq* (*fst* (*watched-by S L* ! *w*)) *S′*) ∧
    L ∈# *all-lits-of-mm*
        (*mset* ‘# *init-clss-lf* (*get-clauses-wl S*) +
        get-unit-clauses-wl S) ∧
    correct-watching-except j w L S ∧
    w < *length* (*watched-by S L*) ∧ *get-conflict-wl S* = *None*›
  using *that assms L* **unfolding** *unit-prop-body-wl-inv-def unit-propagation-inner-loop-body-l-inv-def*
  by (*auto simp*: *twl-st*)

then show *?inv*
  using *that assms* **unfolding** *unit-prop-body-wl-inv-def unit-propagation-inner-loop-body-l-inv-def*
  by *blast*
show *?ge*
  by (*rule ge-0*)
show ‹*distinct-mset-mset* (*mset* ‘# *ran-mf* (*get-clauses-wl S*))›
  using *dist S-S′ twl-st-l*(*1*−*8*)[*OF T-T′*] *T-T′* **unfolding** $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-alt-def*
  by (*auto simp*: *twl-st*)
show *?confl*
  using *confl* **.**
have ‹*watched-by S L* ! *w* ∈ *set* (*take j* (*watched-by S L*)) ∪ *set* (*drop w* (*watched-by S L*))›
  using *L alien′ C′-dom SLw w-le*
  by (*cases S*)
   (*auto simp*: *in-set-drop-conv-nth*)
then show ‹*correctly-marked-as-binary* (*get-clauses-wl S*) (*C′*, *bL*, *bin*)›
  using *corr-w alien′ C′-dom SLw S-S′*
  by (*cases S*; *cases* ‹*watched-by S L* ! *w*›)
   (*clarsimp simp*: *correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def*
    simp del: *Un-iff*
    dest!: *multi-member-split*[*of L*])
qed

have *f′*: ‹(*f*, *f′*) ∈ ⟨*Id*⟩*option-rel*›
  if ‹(*f*, *f′*) ∈ {(*f*, *f′*). *f* = *f′* ∧ *f′* = *None*}› **for** *f f′*
  using *that* **by** *auto*

have *f′′*: ‹(*f*, *f′*) ∈ ⟨*Id*⟩*option-rel*›
  if ‹(*f*, *f′*) ∈ *Id*› **for** *f f′*
  using *that* **by** *auto*
have *i-def′*: ‹*i* = (*if get-clauses-l T* ∝ *C′* ! *0* = *L then 0 else 1*)›
  using *S-S′* **unfolding** *i-def* **by** *auto*

have
  bin-dom: ‹*propagate-proper-bin-case L x1c* (*keep-watch L j w S*) *x1*› **and**
  bin-in-dom: ‹*False* = (*x1* ∉# *dom-m* (*get-clauses-wl* (*keep-watch L j w S*)))› **and**
  bin-pol-not-True:

‹*False* =
  (*polarity* (*get-trail-wl* (*keep-watch L j w S*))
    (*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* !
     (*1* − (*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* ! *0* = *L then 0 else 1*))) =
    Some True)› **and**
bin-cannot-find-new:
  ‹*RETURN None* $\leq \Downarrow$ {(*f*, *f'*). *f* = *f'* $\wedge$ *f'* = *None*}
  (*find-unwatched-l* (*get-trail-wl* (*keep-watch L j w S*)) (*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1*))›
  and
bin-pol-False:
 ‹(*polarity* (*get-trail-wl* (*keep-watch L j w S*)) *x1c* = *Some False*) =
  (*polarity* (*get-trail-wl* (*keep-watch L j w S*))
    (*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* !
     (*1* − (*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* ! *0* = *L then 0 else 1*))) =
   Some False)› **and**
bin-prop:
 ‹(*let i* = *if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1b* ! *0* = *L then 0 else 1*
 in RETURN (*j* + *1*, *w* + *1*, *propagate-lit-wl-bin x1c x1b i* (*keep-watch L j w S*)))
$\leq$ *SPEC* (λ*c*. (*c*, *j* + *1*, *w* + *1*,
       propagate-lit-wl-general
        (*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* !
         (*1* − (*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* ! *0* = *L then 0 else 1*)))
        x1 (*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* ! *0* = *L then 0 else 1*)
        (*keep-watch L j w S*))
       $\in$ *Id*)›
if
  pre: ‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
  st: ‹*x2* = (*x1a*, *x2a*)› ‹*x2b* = (*x1c*, *x2c*)› **and**
  SLw': ‹*watched-by S L* ! *w* = (*x1*, *x2*)› **and**
  SLw'': ‹*watched-by S L* ! *w* = (*x1b*, *x2b*)› **and**
  inv: ‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*› **and**
  ‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*› **and**
  ‹*polarity* (*get-trail-wl* (*keep-watch L j w S*)) *x1c* $\neq$ *Some True*› **and**
  bin: ‹*x2c*› ‹*x2a*›
for *x1 x2 x1a x2a x1b x2b x1c x2c*
proof −
 obtain *T* **where**
  S-T: ‹(*S*, *T*) $\in$ *state-wl-l* (*Some* (*L*, *w*))› **and**
  ‹*j* $\leq$ *w*› **and**
  w-le: ‹*w* < *length* (*watched-by S L*)›
  ‹*unit-propagation-inner-loop-l-inv L* (*T*, *remaining-nondom-wl w L S*)› **and**
  ‹*correct-watching-except j w L S* $\wedge$ *w* $\leq$ *length* (*watched-by S L*)›
  using *pre* **unfolding** *unit-propagation-inner-loop-wl-loop-pre-def prod.simps*
   unit-propagation-inner-loop-wl-loop-inv-def
  by *fast*+
 then obtain *T'* **where**
  S-T: ‹(*S*, *T*) $\in$ *state-wl-l* (*Some* (*L*, *w*))› **and**
  ‹*j* $\leq$ *w*› **and**
  ‹*correct-watching-except j w L S*› **and**
  ‹*w* $\leq$ *length* (*watched-by S L*)› **and**
  T-T': ‹(*T*, *T'*) $\in$ *twl-st-l* (*Some L*)› **and**
  struct-invs: ‹*twl-struct-invs T'*› **and**
  ‹*twl-stgy-invs T'*› **and**
  ‹*twl-list-invs T*› **and**
  uL: ‹− *L* $\in$ *lits-of-l* (*get-trail-l T*)› **and**
  confl: ‹*clauses-to-update T'* $\neq$ {#} $\vee$ *0* < *remaining-nondom-wl w L S* $\longrightarrow$ *get-conflict T'* = *None*›

unfolding *unit-propagation-inner-loop-l-inv-def prod.case*
by *metis*
have *confl*: ‹*get-conflict* T' = *None*›
using *S-T w-le T-T' confl-S*
by (*cases S*; *cases* T') (*auto simp*: *state-wl-l-def twl-st-l-def*)
have
alien: ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of* T')› **and**
dup: ‹*no-duplicate-queued* T'› **and**
lev: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* ($state_W$*-of* T')› **and**
dist: ‹$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state* ($state_W$*-of* T')›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *blast+*
have *n-d*: ‹*no-dup* (*trail* ($state_W$*-of* T'))›
using *lev* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** *auto*
have *1*: ‹C ∈# *clauses-to-update* T' ⟹
add-mset (*fst C*) (*literals-to-update* T') ⊆#
uminus '# *lit-of* '# *mset* (*get-trail* T')› **for** *C*
using *dup* **unfolding** *no-duplicate-queued-alt-def*
by *blast*
have *uL-M*: ‹$-L$ ∈ *lits-of-l* (*get-trail* T')›
using *uL T-T'*
by (*auto simp*: *lits-of-def*)
have *L*: ‹*L* ∈# *all-lits-of-mm*
(*mset* '# *init-clss-lf* (*get-clauses-wl S*) + *get-unit-init-clss-wl S*)›
using *alien uL-M twl-st-l*(*1−8*)[*OF T-T'*] *S-S' S-T*
init-clss-state-to-l[*OF T-T'*]
unit-init-clauses-get-unit-init-clauses-l[*OF T-T'*]
unfolding $cdcl_W$*-restart-mset.no-strange-atm-def*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff twl-st-wl twl-st twl-st-l*)
then have *alien'*:
‹*L* ∈# *all-lits-of-mm* (*mset* '# *ran-mf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*)›
apply (*rule set-rev-mp*)
apply (*rule all-lits-of-mm-mono*)
by (*cases S*) *auto*
have ‹*watched-by S L* ! *w* ∈ *set* (*drop w* (*watched-by S L*))›
using *corr-w alien' SLw S-S' inv pre*
by (*cases S*; *cases* ‹*watched-by S L* ! *w*›)
(*auto simp*: *correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def*
unit-propagation-inner-loop-wl-loop-pre-def in-set-drop-conv-nth
intro!: *bex-geI*[*of - w*]
simp del: *Un-iff*
dest!: *multi-member-split*[*of L*])
then have *H*: ‹*x1* ∈# *dom-m* (*get-clauses-wl S*) ∧ *bL* ∈ *set* (*get-clauses-wl S* ∝ *C'*) ∧
bL ≠ *L* ∧ *correctly-marked-as-binary* (*get-clauses-wl S*) (*C'*, *bL*, *bin*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m* (*get-clauses-wl S*))
(*fst* '# *mset* (*take j* (*watched-by S L*) @ *drop w* (*watched-by S L*))) =
clause-to-update L (*get-trail-wl S*, *get-clauses-wl S*, *get-conflict-wl S*,
get-unit-init-clss-wl S, *get-unit-learned-clss-wl S*, {#}, {#})›
using *corr-w alien' S-S' bin SLw'* **unfolding** *SLw st*
by (*cases S*)
(*auto simp*: *correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def*
simp del:
dest!: *multi-member-split*[*of L*])
then show ‹*False* = (*x1* ∉# *dom-m* (*get-clauses-wl* (*keep-watch L j w S*)))›
by *auto*
have *dom*: ‹*C'* ∈# *dom-m* (*get-clauses-wl S*)› **and**

filter: ‹*filter-mset* (*λi*. *i* ∈# *dom-m* (*get-clauses-wl S*))
(*fst* '# *mset* (*take j* (*watched-by S L*) @ *drop w* (*watched-by S L*))) =
clause-to-update L (*get-trail-wl S*, *get-clauses-wl S*, *get-conflict-wl S*,
get-unit-init-clss-wl S, *get-unit-learned-clss-wl S*, {#}, {#})›
using ‹*watched-by S L* ! *w* ∈ *set* (*drop w* (*watched-by S L*))› *H SLw′* **unfolding** *SLw st*
by *auto*

have *x1c*: ‹*x1c* = *bL*› **and** *x1*: ‹*x1* = *x1b*›
using *SLw′ SLw″* **unfolding** *st SLw*
by *auto*
have ‹*C′* ∈# *filter-mset* (*λi*. *i* ∈# *dom-m* (*get-clauses-wl S*))
(*fst* '# *mset* (*take j* (*watched-by S L*) @ *drop w* (*watched-by S L*)))›
using ‹*watched-by S L* ! *w* ∈ *set* (*drop w* (*watched-by S L*))› *dom*
by *auto*
then have *L-in*: ‹*L* ∈ *set* (*watched-l* (*get-clauses-wl S* ∝ *C′*))›
using *L-watched S-T SLw′ bin* **unfolding** *filter*
by (*auto simp*: *clause-to-update-def*)
moreover have *le2*: ‹*length* (*get-clauses-wl S* ∝ *C′*) = *2*›
using *H SLw′ bin* **unfolding** *SLw st*
by (*auto simp*: *correctly-marked-as-binary.simps*)
ultimately have *lit*: ‹(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* − (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*))) = *bL*› **and**
[*simp*]: ‹*unwatched-l* (*get-clauses-wl S* ∝ *x1*) = []› **and**
lit′: ‹(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* !
((*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* ! *0* = *L then 0 else 1*))) = *L*›
using *H SLw′ bin* **unfolding** *SLw st length-list-2 x1*
by (*auto simp del*: *simp del*: *C′-def*)
show ‹*False* =
(*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* − (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*))) =
Some True)›
using *that*(*8*)
unfolding *x1c lit*
by *auto*
show ‹*propagate-proper-bin-case L x1c* (*keep-watch L j w S*) *x1*›
using *H le2 SLw′ L-in* **unfolding** *propagate-proper-bin-case-def x1 SLw length-list-2 x1 x1c*
by *auto*

show ‹*RETURN None* ≤ ⇓ {(*f*, *f′*). *f* = *f′* ∧ *f′* = *None*}
(*find-unwatched-l* (*get-trail-wl* (*keep-watch L j w S*)) (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*))›
by (*auto simp*: *find-unwatched-l-def RETURN-RES-refine-iff*)
show
‹(*polarity* (*get-trail-wl* (*keep-watch L j w S*)) *x1c* = *Some False*) =
(*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* − (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*))) =
Some False)›
unfolding *x1c lit* **..**
show
bin-prop:
‹(*let i* = *if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* ! *0* = *L then 0 else 1*
in RETURN (*j* + *1*, *w* + *1*, *propagate-lit-wl-bin x1c x1b i* (*keep-watch L j w S*)))
≤ *SPEC* (*λc*. (*c*, *j* + *1*, *w* + *1*,
propagate-lit-wl-general
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !

$(1 - (if\ get\text{-}clauses\text{-}wl\ (keep\text{-}watch\ L\ j\ w\ S) \propto x1\ !\ 0 = L\ then\ 0\ else\ 1)))$
x1 (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*)
(*keep-watch L j w S*))
∈ *Id*)›
using *le2 SLw''*
unfolding *x1c lit Let-def* **unfolding** *x1 propagate-lit-wl-bin-def propagate-lit-wl-general-def*
by (*cases S*)
(*auto intro*!: *RETURN-RES-refine simp*: *keep-watch-def*)
qed
have *find-unwatched-l*:
‹*find-unwatched-l* (*get-trail-wl* (*keep-watch L j w S*)) (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b*)
≤ ⇓ *Id*
(*find-unwatched-l* (*get-trail-wl* (*keep-watch L j w S*)) (*get-clauses-wl* (*keep-watch L j w S*) ∝
x1))›
if
‹*x2* = (*x1a*, *x2a*)› **and**
‹*watched-by S L* ! *w* = (*x1*, *x2*)› **and**
‹*x2b* = (*x1c*, *x2c*)› **and**
‹*watched-by S L* ! *w* = (*x1b*, *x2b*)›
for *x1 x2 x1a x2a x1b x2b x1c x2c*
proof −
show *?thesis*
using *that*
by *auto*
qed
have *propagate-lit-wl*: ‹((*j* + *1*, *w* + *1*,
propagate-lit-wl
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* ! *0* = *L then 0*
else 1)))
x1b
(*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* ! *0* = *L then 0 else 1*)
(*keep-watch L j w S*)),
j + *1*, *w* + *1*,
propagate-lit-wl-general
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*)))
x1 (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*)
(*keep-watch L j w S*))
∈ *Id*›
if
pre: ‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
st: ‹*x2* = (*x1a*, *x2a*)›‹*x2b* = (*x1c*, *x2c*)› **and**
SLw: ‹*watched-by S L* ! *w* = (*x1*, *x2*)› **and**
SLw': ‹*watched-by S L* ! *w* = (*x1b*, *x2b*)› **and**
inv: ‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*› **and**
‹*polarity* (*get-trail-wl* (*keep-watch L j w S*)) *x1c* ≠ *Some True*› **and**
‹*polarity* (*get-trail-wl* (*keep-watch L j w S*)) *x1a* ≠ *Some True*› **and**
bin: ‹¬ *x2c*› ‹¬ *x2a*› **and**
dom: ‹¬ *x1b* ∉# *dom-m* (*get-clauses-wl* (*keep-watch L j w S*))›
‹¬ *x1* ∉# *dom-m* (*get-clauses-wl* (*keep-watch L j w S*))› **and**
‹*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1b* !
(*1* −

(*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1b* ! *0* = *L then 0 else 1*))) $\neq$
Some True⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* ! *0* = *L then 0 else 1*))) $\neq$
Some True⟩ **and**
⟨(*f*, *fa*) ∈ *Id*⟩ **and**
⟨*unit-prop-body-wl-find-unwatched-inv fa x1* (*keep-watch L j w S*)⟩ **and**
⟨*unit-prop-body-wl-find-unwatched-inv f x1b* (*keep-watch L j w S*)⟩ **and**
⟨*f* = *None*⟩ **and**
⟨*fa* = *None*⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1b* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1b* ! *0* = *L then 0 else 1*))) $\neq$
Some False⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) $\propto$ *x1* ! *0* = *L then 0 else 1*))) $\neq$
Some False⟩
for *x1 x2 x1a x2a x1b x2b x1c x2c f fa*
proof −
obtain *T* **where**
S-T: ⟨(*S*, *T*) ∈ *state-wl-l* (*Some* (*L*, *w*))⟩ **and**
⟨*j* ≤ *w*⟩ **and**
w-le: ⟨*w* < *length* (*watched-by S L*)⟩
⟨*unit-propagation-inner-loop-l-inv L* (*T*, *remaining-nondom-wl w L S*)⟩ **and**
⟨*correct-watching-except j w L S* ∧ *w* ≤ *length* (*watched-by S L*)⟩
using *pre* **unfolding** *unit-propagation-inner-loop-wl-loop-pre-def prod.simps*
unit-propagation-inner-loop-wl-loop-inv-def
by *fast+*
then obtain *T'* **where**
S-T: ⟨(*S*, *T*) ∈ *state-wl-l* (*Some* (*L*, *w*))⟩ **and**
⟨*j* ≤ *w*⟩ **and**
⟨*correct-watching-except j w L S*⟩ **and**
⟨*w* ≤ *length* (*watched-by S L*)⟩ **and**
T-T': ⟨(*T*, *T'*) ∈ *twl-st-l* (*Some L*)⟩ **and**
struct-invs: ⟨*twl-struct-invs T'*⟩ **and**
⟨*twl-stgy-invs T'*⟩ **and**
⟨*twl-list-invs T*⟩ **and**
uL: ⟨− *L* ∈ *lits-of-l* (*get-trail-l T*)⟩ **and**
confl: ⟨*clauses-to-update T'* ≠ {#} ∨ *0* < *remaining-nondom-wl w L S* ⟶ *get-conflict T'* = *None*⟩
unfolding *unit-propagation-inner-loop-l-inv-def prod.case*
by *metis*
have *confl*: ⟨*get-conflict T'* = *None*⟩
using *S-T w-le T-T' confl-S*
by (*cases S*; *cases T'*) (*auto simp*: *state-wl-l-def twl-st-l-def*)
have
alien: ⟨$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of T'*)⟩ **and**
dup: ⟨*no-duplicate-queued T'*⟩ **and**
lev: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* ($state_W$*-of T'*)⟩ **and**
dist: ⟨$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state* ($state_W$*-of T'*)⟩ **and**
twl-st-inv: ⟨*twl-st-inv T'*⟩
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*

```
      twl-st-inv.simps
    by blast+
  have n-d: ‹no-dup (trail (stateW-of T'))›
     using lev unfolding cdclW-restart-mset.cdclW-M-level-inv-def by auto
  have 1: ‹C ∈# clauses-to-update T' ⟹
       add-mset (fst C) (literals-to-update T') ⊆#
       uminus `# lit-of `# mset (get-trail T')› for C
    using dup unfolding no-duplicate-queued-alt-def
    by blast
  have uL-M: ‹−L ∈ lits-of-l (get-trail T')›
    using uL T-T'
    by (auto simp: lits-of-def)
  have L: ‹L ∈# all-lits-of-mm
        (mset `# init-clss-lf (get-clauses-wl S) + get-unit-init-clss-wl S)›
    using alien uL-M twl-st-l(1−8)[OF T-T'] S-S' S-T
      init-clss-state-to-l[OF T-T']
      unit-init-clauses-get-unit-init-clauses-l[OF T-T']
    unfolding cdclW-restart-mset.no-strange-atm-def
    by (auto simp: in-all-lits-of-mm-ain-atms-of-iff twl-st-wl twl-st twl-st-l)
  then have alien':
    ‹L ∈# all-lits-of-mm (mset `# ran-mf (get-clauses-wl S) + get-unit-clauses-wl S)›
    apply (rule set-rev-mp)
    apply (rule all-lits-of-mm-mono)
    by (cases S) auto
  have ‹watched-by S L ! w  ∈ set (drop w (watched-by S L))›
    using corr-w alien' SLw S-S' inv pre
    by (cases S; cases ‹watched-by S L ! w›)
      (auto simp: correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def
        unit-propagation-inner-loop-wl-loop-pre-def in-set-drop-conv-nth
        intro!: bex-geI[of -  w]
        simp del: Un-iff
        dest!: multi-member-split[of L])
  then have H: ‹correctly-marked-as-binary (get-clauses-wl S) (x1b, x1c, False)›
    using corr-w alien' S-S' SLw'[unfolded SLw] SLw bin dom unfolding st
    by (cases S)
      (auto simp: correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def
        simp del:
        dest!: multi-member-split[of L])
  have ‹∀ C∈# (dom-m (get-clauses-wl S)). length (get-clauses-wl S ∝ C) ≥ 2›
    using twl-st-inv S-T T-T'
    by (cases T; cases T'; cases S)
      (auto simp: state-wl-l-def twl-st-l-def twl-st-inv.simps
      image-Un[symmetric])
  then have le2: ‹length (get-clauses-wl S ∝ C') > 2›
    using H SLw' bin dom unfolding SLw st
    by (auto simp: correctly-marked-as-binary.simps SLw)
  then show ?thesis
    using that
    by (cases S)
      (auto simp: propagate-lit-wl-def
      propagate-lit-wl-general-def keep-watch-def)
qed
show ?thesis
  unfolding unit-propagation-inner-loop-body-wl-int-alt-def2
     unit-propagation-inner-loop-body-wl-alt-def
  apply refine-rcg
```

subgoal by *auto*
subgoal by *auto*
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c*
  by (*rule bin-in-dom*)
subgoal by (*rule bin-pol-not-True*)
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c*
  by *fast* — impossible case
      apply (*rule bin-cannot-find-new*; *assumption*)
apply (*rule* f'; *assumption*)
subgoal
  by (*rule bin-dom*)
subgoal
  by (*rule bin-pol-False*)
subgoal by *auto*
subgoal
  by (*rule bin-prop*)
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c*
  by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
    apply (*rule find-unwatched-l*; *assumption*)
subgoal by *auto*
apply (*rule* f''; *assumption*)
subgoal by *auto*
subgoal by *auto*
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c f fa*
  by (*rule propagate-lit-wl*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

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

  defines
    [*simp*]: ‹$C'' \equiv$ *get-clauses-l* $S' \propto C'$›
  assumes
    S-S': ‹$(S, S') \in$ *state-wl-l* (*Some* (L, w))› **and**
    w-le: ‹$w <$ *length* (*watched-by* S L)› **and**
    j-w: ‹$j \leq w$› **and**
    corr-w: ‹*correct-watching-except* j w L S› **and**
    inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv* L (j, w, S)› **and**
    n: ‹$n =$ *size* (*filter-mset* ($\lambda(i, -)$. $i \notin\#$ *dom-m* (*get-clauses-wl* S)) (*mset* (*drop* w (*watched-by* S L))))›
and
    confl-S: ‹*get-conflict-wl* S = *None*›
  shows *unit-propagation-inner-loop-body-wl-int-spec*: ‹*unit-propagation-inner-loop-body-wl-int* L j w S
$\leq$
    $\Downarrow\{((i, j, T'), (T, n))$.
      $(T', T) \in$ *state-wl-l* (*Some* (L, j)) $\wedge$
      correct-watching-except i j L T' $\wedge$

$j \leq$ *length* (*watched-by* T' L) $\wedge$
length (*watched-by* S L) $=$ *length* (*watched-by* T' L) $\wedge$
$i \leq j \wedge$
(*get-conflict-wl* $T' =$ *None* $\longrightarrow$
$n =$ *size* (*filter-mset* (λ(i, -). $i \notin\#$ *dom-m* (*get-clauses-wl* T')) (*mset* (*drop* j (*watched-by* T' L))))) $\wedge$
(*get-conflict-wl* $T' \neq$ *None* $\longrightarrow n = 0$)}
(*unit-propagation-inner-loop-body-l-with-skip* L (S', n))› (**is** ‹*?propa*› **is** ‹- $\leq \Downarrow$ *?unit* -›)**and**
unit-propagation-inner-loop-body-wl-update:
‹*unit-propagation-inner-loop-body-l-inv* L C' $T \Longrightarrow$
mset ‘# (*ran-mf* ((*get-clauses-wl* S) ($C' \hookrightarrow$ (*swap* (*get-clauses-wl* $S \propto C'$) *0*
(*1* $-$ (*if* (*get-clauses-wl* S)$\propto C'$! *0* $= L$ *then* *0* *else* *1*)))))) $=$
mset ‘# (*ran-mf* (*get-clauses-wl* S))› (**is** ‹- $\Longrightarrow$ *?eq*›)
proof $-$
obtain *bL* **where** *SLw*: ‹*watched-by* S L ! w = (C', *bL*)›
using C'*-def* **by** (*cases* ‹*watched-by* S L ! w›) *auto*
have *val*: ‹(*polarity* a b, *polarity* a' b') $\in$ *Id*›
if ‹$a = a'$› **and** ‹$b = b'$› **for** a a' :: ‹($'a$, $'b$) *ann-lits*› **and** b b' :: ‹$'a$ *literal*›
by (*auto simp*: *that*)
let *?M* = ‹*get-trail-wl* S›
have f: ‹*find-unwatched-l* (*get-trail-wl* S) (*get-clauses-wl* $S \propto C'$)
$\leq \Downarrow$ {(*found*, *found'*). *found* = *found'* $\wedge$
(*found* = *None* $\longleftrightarrow$ ($\forall$ $L$$\in$*set* (*unwatched-l* C''). $-L \in$ *lits-of-l* *?M*)) $\wedge$
($\forall$ j. *found* = *Some* $j \longrightarrow$ ($j <$ *length* C'' $\wedge$ (*undefined-lit* *?M* (C''!j) $\vee$ C''!$j \in$ *lits-of-l* *?M*) $\wedge$ $j \geq 2$))
}
(*find-unwatched-l* (*get-trail-l* T) (*get-clauses-l* $T \propto C'$))›
(**is** ‹- $\leq \Downarrow$ *?find* -›)
using *S-S'* **by** (*auto simp*: *find-unwatched-l-def intro*!: *RES-refine*)

define i :: *nat* **where**
‹$i \equiv$ (*if* *get-clauses-wl* $S \propto C'$! *0* $= L$ *then* *0* *else* *1*)›
have
l-wl-inv: ‹*unit-prop-body-wl-inv* S j w L› (**is** *?inv*) **and**
clause-ge-0: ‹*0* $<$ *length* (*get-clauses-l* $T \propto C'$)› (**is** *?ge*) **and**
L-def: ‹*defined-lit* (*get-trail-wl* S) L› ‹$-L \in$ *lits-of-l* (*get-trail-wl* S)›
‹$L \notin$ *lits-of-l* (*get-trail-wl* S)› (**is** *?L-def*) **and**
i-le: ‹$i <$ *length* (*get-clauses-wl* $S \propto C'$)› (**is** *?i-le*) **and**
i-le2: ‹*1*$-i <$ *length* (*get-clauses-wl* $S \propto C'$)› (**is** *?i-le2*) **and**
C'*-dom*: ‹$C' \in\#$ *dom-m* (*get-clauses-l* T)› (**is** *?*C'*-dom*) **and**
L-watched: ‹$L \in$ *set* (*watched-l* (*get-clauses-l* $T \propto C'$))› (**is** *?L-w*) **and**
dist-clss: ‹*distinct-mset-mset* (*mset* ‘# *ran-mf* (*get-clauses-wl* S))› **and**
confl: ‹*get-conflict-l* T = *None*› (**is** *?confl*) **and**
alien-L:
‹$L \in\#$ *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl* S) + *get-unit-init-clss-wl* S)›
(**is** *?alien*) **and**
alien-L':
‹$L \in\#$ *all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl* S) + *get-unit-clauses-wl* S)›
(**is** *?alien'*) **and**
alien-L'':
‹$L \in\#$ *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl* S) + *get-unit-clauses-wl* S)›
(**is** *?alien''*) **and**
correctly-marked-as-binary: ‹*correctly-marked-as-binary* (*get-clauses-wl* S) (C', *bL*)›
if
‹*unit-propagation-inner-loop-body-l-inv* L C' T›
proof $-$

have ‹*unit-propagation-inner-loop-body-l-inv* L C' T›
  using *that* **unfolding** *unit-prop-body-wl-inv-def* **by** *fast+*
then obtain T' **where**
  T-T': ‹(*set-clauses-to-update-l* (*clauses-to-update-l* T + {#C'#}) T, T') ∈ *twl-st-l* (*Some* L)› **and**
  struct-invs: ‹*twl-struct-invs* T'› **and**
  ‹*twl-stgy-invs* T'› **and**
  C'-dom: ‹C' ∈# *dom-m* (*get-clauses-l* T)› **and**
  ‹*0* < C'› **and**
  ge-0: ‹*0* < *length* (*get-clauses-l* T ∝ C')› **and**
  ‹*no-dup* (*get-trail-l* T)› **and**
  i-le: ‹(*if get-clauses-l* T ∝ C' ! *0* = L *then 0 else 1*)
    < *length* (*get-clauses-l* T ∝ C')› **and**
  i-le2: ‹*1* − (*if get-clauses-l* T ∝ C' ! *0* = L *then 0 else 1*)
    < *length* (*get-clauses-l* T ∝ C')› **and**
  L-watched: ‹L ∈ *set* (*watched-l* (*get-clauses-l* T ∝ C'))› **and**
  confl: ‹*get-conflict-l* T = *None*›
  unfolding *unit-propagation-inner-loop-body-l-inv-def* **by** *blast*
show *?i-le* **and** *?C'-dom* **and** *?L-w* **and** *?i-le2*
  using *S-S'* *i-le* *C'-dom* *L-watched* *i-le2* **unfolding** *i-def* **by** *auto*
have
    alien: ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of* T')› **and**
    dup: ‹*no-duplicate-queued* T'› **and**
    lev: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv* ($state_W$*-of* T')› **and**
    dist: ‹$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state* ($state_W$*-of* T')›
  using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
  by *blast+*
have *n-d*: ‹*no-dup* (*trail* ($state_W$*-of* T'))›
   using *lev* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** *auto*
have *1*: ‹C ∈# *clauses-to-update* T' ⟹
     add-mset (*fst* C) (*literals-to-update* T') ⊆#
     uminus ‘# *lit-of* ‘# *mset* (*get-trail* T')› **for** C
  using *dup* **unfolding** *no-duplicate-queued-alt-def*
  by *blast*
have *H*: ‹(L, *twl-clause-of* C'') ∈# *clauses-to-update* T'›
  using *twl-st-l*(*5*)[*OF T-T'*]
  by (*auto simp*: *twl-st-l*)
have *uL-M*: ‹−L ∈ *lits-of-l* (*get-trail* T')›
  using *mset-le-add-mset-decr-left2*[*OF 1*[*OF H*]]
  by (*auto simp*: *lits-of-def*)
then show ‹*defined-lit* (*get-trail-wl* S) L› ‹−L ∈ *lits-of-l* (*get-trail-wl* S)›
  ‹L ∉ *lits-of-l* (*get-trail-wl* S)›
  using *S-S'* *T-T'* *n-d* **by** (*auto simp*: *Decided-Propagated-in-iff-in-lits-of-l twl-st*
    dest: *no-dup-consistentD*)
show *L*: *?alien*
  using *alien uL-M twl-st-l*(*1*−*8*)[*OF T-T'*] *S-S'*
    init-clss-state-to-l[*OF T-T'*]
    unit-init-clauses-get-unit-init-clauses-l[*OF T-T'*]
  unfolding $cdcl_W$*-restart-mset.no-strange-atm-def*
  by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff twl-st-wl twl-st twl-st-l*)
then show *alien'*: *?alien'*
  apply (*rule set-rev-mp*)
  apply (*rule all-lits-of-mm-mono*)
  by (*cases S*) *auto*
show *?alien''*
  using *L*
  apply (*rule set-rev-mp*)

apply (*rule all-lits-of-mm-mono*)
by (*cases S*) *auto*
then have *l-wl-inv*: ‹(*S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *w*)) ∧
unit-propagation-inner-loop-body-l-inv L (*fst* (*watched-by S L* ! *w*))
(*remove-one-lit-from-wq* (*fst* (*watched-by S L* ! *w*)) *S'*) ∧
L ∈# *all-lits-of-mm*
(*mset* ‘# *init-clss-lf* (*get-clauses-wl S*) +
get-unit-clauses-wl S) ∧
correct-watching-except j w L S ∧
w < *length* (*watched-by S L*) ∧ *get-conflict-wl S* = *None*›
using *that assms L* **unfolding** *unit-prop-body-wl-inv-def unit-propagation-inner-loop-body-l-inv-def*
by (*auto simp*: *twl-st*)

then show *?inv*
using *that assms* **unfolding** *unit-prop-body-wl-inv-def unit-propagation-inner-loop-body-l-inv-def*
by *blast*
show *?ge*
by (*rule ge-0*)
show ‹*distinct-mset-mset* (*mset* ‘# *ran-mf* (*get-clauses-wl S*))›
using *dist S-S' twl-st-l*(*1*−*8*)[*OF T-T'*] *T-T'* **unfolding** $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-alt-def*
by (*auto simp*: *twl-st*)
show *?confl*
using *confl* **.**
have ‹*watched-by S L* ! *w* ∈ *set* (*take j* (*watched-by S L*)) ∪ *set* (*drop w* (*watched-by S L*))›
using *L alien' C'-dom SLw w-le*
by (*cases S*)
(*auto simp*: *in-set-drop-conv-nth*)
then show ‹*correctly-marked-as-binary* (*get-clauses-wl S*) (*C'*, *bL*)›
using *corr-w alien' C'-dom SLw S-S'*
by (*cases S*; *cases* ‹*watched-by S L* ! *w*›)
(*clarsimp simp*: *correct-watching-except.simps Ball-def all-conj-distrib state-wl-l-def*
simp del: *Un-iff*
dest!: *multi-member-split*[*of L*])
qed

have *f'*: ‹(*f*, *f'*) ∈ ⟨*Id*⟩*option-rel*›
if ‹*f* = *f'*› **for** *f f'*
using *that* **by** *auto*

have *i-def'*: ‹*i* = (*if get-clauses-l T* ∝ *C'* ! *0* = *L then 0 else 1*)›
using *S-S'* **unfolding** *i-def* **by** *auto*
have [*refine0*]: ‹*RETURN* (*C'*, *bL*) ≤ ⇓ {((*C'*, *bL*), *b*). (*b* ⟷ *C'*∉# *dom-m* (*get-clauses-wl S*)) ∧
(*b* ⟶ *0* < *n*) ∧ (¬*b* ⟶ *clauses-to-update-l S'* ≠ {#})}
(*SPEC* (λ*b*. (*b* ⟶ *0* < *n*) ∧ (¬*b* ⟶ *clauses-to-update-l S'* ≠ {#})))›
(**is** ‹- ≤ ⇓ *?blit* -›)
if ‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)› **and**
‹*clauses-to-update-l S'* ≠ {#} ∨ *0* < *n* › ‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)›
‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)›
proof −
have *1*: ‹(*C'*, *bL*) ∈# {#(*i*, -) ∈# *mset* (*drop w* (*watched-by S L*)). *i* ∉# *dom-m* (*get-clauses-wl S*)#}›
if ‹*fst* (*watched-by S L* ! *w*) ∉# *dom-m* (*get-clauses-wl S*) ›
using *that w-le* **unfolding** *SLw* **apply** −
apply (*auto simp add*: *in-set-drop-conv-nth intro*!: *ex-geI*[*of - w*])
unfolding *SLw*
apply *auto*

done
have ‹*fst* (*watched-by S L* ! *w*) ∈# *dom-m* (*get-clauses-wl S*) ⟹
clauses-to-update-l S' = {#} ⟹ *False*›
using *S-S' w-le that n 1* **unfolding** *SLw unit-propagation-inner-loop-l-inv-def* **apply** −
by (*cases S*; *cases S'*)
(*auto simp add*: *state-wl-l-def in-set-drop-conv-nth twl-st-l-def*
Cons-nth-drop-Suc[*symmetric*]
intro: *ex-geI*[*of* - *w*]
split: *if-splits*)
with *multi-member-split*[*OF 1*] **show** *?thesis*
apply (*intro RETURN-SPEC-refine*)
apply (*rule exI*[*of* - ‹*C'* ∉# *dom-m* (*get-clauses-wl S*)›])
using *n*
by *auto*
qed
have [*simp*]: ‹*length* (*watched-by* (*keep-watch L j w S*) *L*) = *length* (*watched-by S L*)› **for** *S j w L*
by (*cases S*) (*auto simp*: *keep-watch-def*)
have *S-removal*: ‹(*S*, *set-clauses-to-update-l*
(*remove1-mset* (*fst* (*watched-by S L* ! *w*)) (*clauses-to-update-l S'*)) *S'*)
∈ *state-wl-l* (*Some* (*L*, *Suc w*))›
using *S-S' w-le* **by** (*cases S*; *cases S'*)
(*auto simp*: *state-wl-l-def Cons-nth-drop-Suc*[*symmetric*])

have *K*:
‹*RETURN* (*get-clauses-wl* (*keep-watch L j w S*) ∝ *C'*)
≤ ⇓ {(-, (*U*, *C*)). *C* = *C'* ∧ (*S*, *U*) ∈ *state-wl-l* (*Some* (*L*, *Suc w*))} (*select-from-clauses-to-update*
S')›
if ‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)› **and**
‹*fst* (*watched-by S L* ! *w*) ∈# *clauses-to-update-l S'*›
unfolding *select-from-clauses-to-update-def*
apply (*rule RETURN-RES-refine*)
apply (*rule exI*[*of* - ‹(*T*, *C'*)›])
by (*auto simp*: *remove-one-lit-from-wq-def S-removal that*)
have *keep-watch-state-wl*: ‹*fst* (*watched-by S L* ! *w*) ∉# *dom-m* (*get-clauses-wl S*) ⟹
(*keep-watch L j w S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *Suc w*))›
using *S-S' w-le j-w* **by** (*cases S*; *cases S'*)
(*auto simp*: *state-wl-l-def keep-watch-def Cons-nth-drop-Suc*[*symmetric*]
drop-map)
have [*simp*]: ‹*drop* (*Suc w*) (*watched-by* (*keep-watch L j w S*) *L*) = *drop* (*Suc w*) (*watched-by S L*)›
using *j-w w-le* **by** (*cases S*) (*auto simp*: *keep-watch-def*)
have [*simp*]: ‹*get-clauses-wl* (*keep-watch L j w S*) = *get-clauses-wl S*› **for** *L j w S*
by (*cases S*) (*auto simp*: *keep-watch-def*)
have *keep-watch*:
‹*RETURN* (*keep-watch L j w S*) ≤ ⇓ {(*T*, (*T'*, *C*)). (*T*, *T'*) ∈ *state-wl-l* (*Some* (*L*, *Suc w*)) ∧
C = *C'* ∧ *T'* = *set-clauses-to-update-l* (*clauses-to-update-l S'* − {#*C*#}) *S'*}
(*select-from-clauses-to-update S'*)›
(**is** ‹- ≤ ⇓ *?keep-watch* -›)
if
cond: ‹*clauses-to-update-l S'* ≠ {#} ∨ *0* < *n*› **and**
inv: ‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)› **and**
‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)› **and**
‹¬ *C'* ∉# *dom-m* (*get-clauses-wl S*)› **and**
clss: ‹*clauses-to-update-l S'* ≠ {#}›
proof −
have ‹*get-conflict-l S'* = *None*›
using *clss inv* **unfolding** *unit-propagation-inner-loop-l-inv-def twl-struct-invs-def prod.case*

apply −
apply *normalize-goal+*
by *auto*
then show *?thesis*
using *S-S′ that w-le j-w*
unfolding *select-from-clauses-to-update-def keep-watch-def*
by (*cases S*)
(*auto intro*!: *RETURN-RES-refine simp*: *state-wl-l-def drop-map*
Cons-nth-drop-Suc[*symmetric*])
qed
have *trail-keep-w*: ‹*get-trail-wl* (*keep-watch L j w S*) = *get-trail-wl S*› **for** *L j w S*
by (*cases S*) (*auto simp*: *keep-watch-def*)
have *unit-prop-body-wl-inv*: ‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*›
if
‹*clauses-to-update-l S′* ≠ {#} ∨ *0* < *n*› **and**
loop-l: ‹*unit-propagation-inner-loop-l-inv L* (*S′*, *n*)› **and**
loop-wl: ‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
‹((*C′*, *bL*), *b*) ∈ *?blit*› **and**
‹(*C′*, *bL*) = (*x1*, *x2*)› **and**
‹¬ *x1* ∉# *dom-m* (*get-clauses-wl S*)› **and**
‹¬ *b*› **and**
‹*clauses-to-update-l S′* ≠ {#}› **and**
X2: ‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*› **and**
inv: ‹*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)›
for *x1 b X2 x2*
proof −
have *corr-w′*:
‹*correct-watching-except j w L S* ⟹ *correct-watching-except j w L* (*keep-watch L j w S*)›
using *j-w w-le*
apply (*cases S*)
apply (*simp only*: *correct-watching-except.simps keep-watch-def prod.case*)
apply (*cases* ‹*j* = *w*›)
by *simp-all*
have [*simp*]:
‹(*keep-watch L j w S*, *S′*) ∈ *state-wl-l* (*Some* (*L*, *w*)) ⟷ (*S*, *S′*) ∈ *state-wl-l* (*Some* (*L*, *w*))›
using *j-w*
by (*cases S* ; *cases* ‹*j*=*w*›)
(*auto simp*: *state-wl-l-def keep-watch-def drop-map*)
have [*simp*]: ‹*watched-by* (*keep-watch L j w S*) *L* ! *w* = *watched-by S L* ! *w*›
using *j-w*
by (*cases S* ; *cases* ‹*j*=*w*›)
(*auto simp*: *state-wl-l-def keep-watch-def drop-map*)
have [*simp*]: ‹*get-conflict-wl S* = *None*›
using *S-S′ inv X2* **unfolding** *unit-propagation-inner-loop-body-l-inv-def* **apply** −
apply *normalize-goal+*
by *auto*
have ‹*unit-propagation-inner-loop-body-l-inv L C′ T*›
using *that* **by** (*auto simp*: *remove-one-lit-from-wq-def*)
then have ‹*L* ∈# *all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*)›
using *alien-L′′* **by** *fast*
then show *?thesis*
using *j-w w-le*
unfolding *unit-prop-body-wl-inv-def*
apply (*intro impI conjI*)
subgoal using *w-le* **by** *auto*
subgoal using *j-w* **by** *auto*

subgoal
  apply (*rule exI*[*of* - *S'*])
  using *inv X2 w-le S-S'*
  by (*auto simp*: *corr-w' corr-w remove-one-lit-from-wq-def*)
done
qed
have [*refine0*]: ‹*SPEC* ((=) *x2*) ≤ *SPEC* (λ*K*. *K* ∈ *set* (*get-clauses-l* (*fst X2*) ∝ *snd X2*))›
if
‹*clauses-to-update-l S'* ≠ {#} ∨ *0* < *n*› **and**
‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
bL: ‹((*C'*, *bL*), *b*) ∈ *?blit*› **and**
x: ‹(*C'*, *bL*) = (*x1*, *x2'*)› **and**
x2': ‹*x2'* =(*x2*, *x3*)› **and**
x1: ‹¬ *x1* ∉# *dom-m* (*get-clauses-wl S*)› **and**
‹¬ *b*› **and**
‹*clauses-to-update-l S'* ≠ {#}› **and**
X2: ‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*› **and**
‹*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)› **and**
‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*›
for *x1 x2 X2 b x3 x2'*
proof −
have [*simp*]: ‹*x2'* = *bL*› ‹*x1* = *C'*›
  using *x* **by** *simp-all*
have ‹*unit-propagation-inner-loop-body-l-inv L C' T*›
  using *that* **by** (*auto simp*: *remove-one-lit-from-wq-def*)
from *alien-L'*[*OF this*]
have ‹*L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*)›
  .
from *correct-watching-exceptD*[*OF corr-w this w-le*]
have ‹*fst bL* ∈ *set* (*get-clauses-wl S* ∝ *fst* (*watched-by S L* ! *w*))›
  using *x1 SLw*
  by (*cases S*; *cases* ‹*watched-by S L* ! *w*›) (*auto simp add*:)
then show *?thesis*
  using *bL X2 S-S' x1 x2'*
  by *auto*
qed
have *find-unwatched-l*: ‹*find-unwatched-l* (*get-trail-wl* (*keep-watch L j w S*))
  (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*)
  ≤ ⇓ {(*k*, *k'*). *k* = *k'* ∧ *get-clauses-wl S* ∝ *x1* ≠ [] ∧
    (*k* ≠ *None* ⟶ (*the k* ≥ *2* ∧ *the k* < *length* (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*) ∧
      (*undefined-lit* (*get-trail-wl S*) (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*!(*the k*))
        ∨ *get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*!(*the k*) ∈ *lits-of-l* (*get-trail-wl S*)))) ∧
    ((*k* = *None*) ⟷
      (∀ *La*∈#*mset* (*unwatched-l* (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*)).
      − *La* ∈ *lits-of-l* (*get-trail-wl* (*keep-watch L j w S*))))}
  (*find-unwatched-l* (*get-trail-l* (*fst X2*))
    (*get-clauses-l* (*fst X2*) ∝ *snd X2*))›
(**is** ‹- ≤ ⇓ *?find-unw* -›)
if
C': ‹(*C'*, *bL*) = (*x1*, *x2*)› **and**
X2: ‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*› **and**
x: ‹*x* ∈ {*K*. *K* ∈ *set* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)}› **and**
‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*›
for *x1 x2 X2 x*
proof −

show *?thesis*
using *S-S′ X2 SLw that* **unfolding** *C′*
by (*auto simp*: *twl-st-wl find-unwatched-l-def intro*!: *SPEC-refine*)
qed

have *blit-final*:
‹(*if polarity* (*get-trail-wl* (*keep-watch L j w S*)) *x2* = *Some True*
then RETURN (*j* + *1*, *w* + *1*, *keep-watch L j w S*)
else RETURN (*j*, *w* + *1*, *keep-watch L j w S*))
≤ ⇓ *?unit*
(*RETURN* (*S′*, *n* − *1*))›
if
‹((*C′*, *bL*), *b*) ∈ *?blit*› **and**
‹(*C′*, *bL*) = (*x1*, *x2′*)› **and**
x2′: ‹*x2′* =(*x2*, *x3*)› **and**
‹*x1* ∉# *dom-m* (*get-clauses-wl S*)› **and**
‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*›
for *b x1 x2 x2′ x3*
using *S-S′ w-le j-w n that confl-S*
by (*auto simp*: *keep-watch-state-wl assert-bind-spec-conv Let-def twl-st-wl*
Cons-nth-drop-Suc[*symmetric*] *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
corr-w correct-watching-except-correct-watching-except-Suc-notin
split: *if-splits*)

have *conflict-final*: ‹((*j* + *1*, *w* + *1*,
set-conflict-wl (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1*)
(*keep-watch L j w S*)),
set-conflict-l (*get-clauses-l* (*fst X2*) ∝ *snd X2*) (*fst X2*),
if get-conflict-l
(*set-conflict-l* (*get-clauses-l* (*fst X2*) ∝ *snd X2*) (*fst X2*)) =
None
then n else 0)
∈ *?unit*›
if
C′-bl: ‹(*C′*, *bL*) = (*x1*, *x2′*)› **and**
x2′: ‹*x2′* =(*x2*, *x3*)› **and**
X2: ‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*›
for *b x1 x2 X2 K x f x′ x2′ x3*
proof −
have [*simp*]: ‹*get-conflict-l* (*set-conflict-l C S*) ≠ *None*›
‹*get-conflict-wl* (*set-conflict-wl C S′*) = *Some* (*mset C*)›
‹*watched-by* (*set-conflict-wl C S′*) *L* = *watched-by S′ L*› **for** *C S S′ L*
apply (*cases S*; *auto simp*: *set-conflict-l-def*; *fail*)
apply (*cases S′*; *auto simp*: *set-conflict-wl-def*; *fail*)
apply (*cases S′*; *auto simp*: *set-conflict-wl-def*; *fail*)
done
have [*simp*]: ‹*correct-watching-except j w L* (*set-conflict-wl C S*) ⟷
correct-watching-except j w L S› **for** *j w L C S*
apply (*cases S*)
by (*simp only*: *correct-watching-except.simps*
set-conflict-wl-def prod.case clause-to-update-def get-clauses-l.simps)
have ‹(*set-conflict-wl* (*get-clauses-wl S* ∝ *x1*) (*keep-watch L j w S*),
set-conflict-l (*get-clauses-l* (*fst X2*) ∝ *snd X2*) (*fst X2*))
∈ *state-wl-l* (*Some* (*L*, *Suc w*))›
using *S-S′ X2 SLw C′-bl* **by** (*cases S*; *cases S′*) (*auto simp*: *state-wl-l-def*
set-conflict-wl-def set-conflict-l-def keep-watch-def

clauses-to-update-wl.simps)
then show *?thesis*
using *S-S′ w-le j-w n*
by (*auto simp*: *keep-watch-state-wl*
correct-watching-except-correct-watching-except-Suc-Suc-keep-watch
corr-w correct-watching-except-correct-watching-except-Suc-notin
split: *if-splits*)
qed
have *propa-final*: ‹((*j* + *1*, *w* + *1*,
propagate-lit-wl-general
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*)))
x1 (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*)
(*keep-watch L j w S*)),
propagate-lit-l
(*get-clauses-l* (*fst X2*) ∝ *snd X2* !
(*1* − (*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*)))
(*snd X2*) (*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*)
(*fst X2*),
if get-conflict-l
(*propagate-lit-l*
(*get-clauses-l* (*fst X2*) ∝ *snd X2* !
(*1* − (*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*)))
(*snd X2*) (*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*)
(*fst X2*)) =
None
then n else 0)
∈ *?unit*›
if
C′: ‹(*C′*, *bL*) = (*x1*, *x2*)› **and**
x1-dom: ‹¬ *x1* ∉# *dom-m* (*get-clauses-wl S*)› **and**
X2: ‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*› **and**
l-inv: ‹*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)›

for *b x1 x2 X2 K x f x′*
proof −
have [*simp*]: ‹*get-conflict-l* (*propagate-lit-l C L w S*) = *get-conflict-l S*›
‹*watched-by* (*propagate-lit-wl-general C L w S′*) *L′* = *watched-by S′ L′*›
‹*get-conflict-wl* (*propagate-lit-wl-general C L w S′*) = *get-conflict-wl S′*›
‹*L* ∈# *dom-m* (*get-clauses-wl S′*) ⟹
dom-m (*get-clauses-wl* (*propagate-lit-wl-general C L w S′*)) = *dom-m* (*get-clauses-wl S′*)›
‹*dom-m* (*get-clauses-wl* (*keep-watch L′ i j S′*)) = *dom-m* (*get-clauses-wl S′*)›
for *C L w S S′ L′ i j*
apply (*cases S*; *auto simp*: *propagate-lit-l-def*; *fail*)
apply (*cases S′*; *auto simp*: *propagate-lit-wl-general-def*; *fail*)
apply (*cases S′*; *auto simp*: *propagate-lit-wl-general-def*; *fail*)
apply (*cases S′*; *auto simp*: *propagate-lit-wl-general-def*; *fail*)
apply (*cases S′*; *auto simp*: *propagate-lit-wl-general-def*; *fail*)
done
define *i* :: *nat* **where** ‹*i* ≡ *if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*›
have *i-alt-def*: ‹*i* = (*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*)›
using *X2 S-S′ SLw* **unfolding** *i-def C′* **by** *auto*
have *x1-dom*[*simp*]: ‹*x1* ∈# *dom-m* (*get-clauses-wl S*)›
using *x1-dom* **by** *fast*
have [*simp*]: ‹*get-clauses-wl S* ∝ *x1* ! *0* ≠ *L* ⟹ *get-clauses-wl S* ∝ *x1* ! *Suc 0* = *L*› **and**

⟨Suc 0 < length (get-clauses-wl S ∝ x1)⟩
using *l-inv X2 S-S′ SLw* **unfolding** *unit-propagation-inner-loop-body-l-inv-def C′*
apply − **apply** *normalize-goal+*
by (*cases* ⟨*get-clauses-wl S ∝ x1*⟩; *cases* ⟨*tl* (*get-clauses-wl S ∝ x1*)⟩)
auto

have *n*: ⟨*n = size* {#(*i*, -) ∈# *mset* (*drop* (*Suc w*) (*watched-by S L*)).
i ∉# *dom-m* (*get-clauses-wl S*)#}⟩
using *n*
apply (*subst* (*asm*) *Cons-nth-drop-Suc*[*symmetric*])
subgoal using *w-le* **by** *simp*
subgoal using *n SLw X2 S-S′* **unfolding** *i-def C′* **by** *auto*
done
have [*simp*]: ⟨*get-conflict-l* (*fst X2*) = *get-conflict-wl S*⟩
using *X2 S-S′* **by** *auto*

have
⟨(*propagate-lit-wl-general* (*get-clauses-wl S ∝ x1* ! (*Suc 0* − *i*)) *x1 i* (*keep-watch L j w S*),
propagate-lit-l (*get-clauses-l* (*fst X2*) ∝ *snd X2* ! (*Suc 0* − *i*)) (*snd X2*) *i* (*fst X2*))
∈ *state-wl-l* (*Some* (*L*, *Suc w*))⟩
using *X2 S-S′ SLw j-w w-le multi-member-split*[*OF x1-dom*] **unfolding** *C′*
by (*cases S*; *cases S′*)
(*auto simp*: *state-wl-l-def propagate-lit-wl-general-def keep-watch-def*
propagate-lit-l-def drop-map)
moreover have ⟨*correct-watching-except* (*Suc j*) (*Suc w*) *L* (*keep-watch L j w S*) ⟹
correct-watching-except (*Suc j*) (*Suc w*) *L*
(*propagate-lit-wl-general* (*get-clauses-wl S ∝ x1* ! (*Suc 0* − *i*)) *x1 i* (*keep-watch L j w S*))⟩
apply (*rule correct-watching-except-correct-watching-except-propagate-lit-wl*)
using *w-le j-w* ⟨*Suc 0 < length* (*get-clauses-wl S ∝ x1*)⟩ **by** *auto*
moreover have ⟨*correct-watching-except* (*Suc j*) (*Suc w*) *L* (*keep-watch L j w S*)⟩
by (*simp add*: *corr-w correct-watching-except-correct-watching-except-Suc-Suc-keep-watch j-w w-le*)
ultimately show *?thesis*
using *w-le* **unfolding** *i-def*[*symmetric*] *i-alt-def*[*symmetric*]
by (*auto simp*: *twl-st-wl j-w n*)
qed

have *update-blit-wl-final*:
⟨*update-blit-wl L x1 x3 j w* (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *xa*) (*keep-watch L j w S*)
≤ ⇓ *?unit*
(*RETURN* (*fst X2*, *if get-conflict-l* (*fst X2*) = *None then n else 0*))⟩
if
cond: ⟨*clauses-to-update-l S′* ≠ {#} ∨ *0 < n*⟩ **and**
loop-inv: ⟨*unit-propagation-inner-loop-l-inv L* (*S′*, *n*)⟩ **and**
⟨*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)⟩ **and**
C′bl: ⟨((*C′*, *bL*), *b*) ∈ *?blit*⟩ **and**
C′-bl: ⟨(*C′*, *bL*) = (*x1*, *x2′*)⟩ **and**
x2′: ⟨*x2′* =(*x2*, *x3*)⟩ **and**
dom: ⟨¬ *x1* ∉# *dom-m* (*get-clauses-wl S*)⟩ **and**
⟨¬ *b*⟩ **and**
⟨*clauses-to-update-l S′* ≠ {#}⟩ **and**
X2: ⟨(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*⟩ **and**
pre: ⟨*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)⟩ **and**
⟨*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*⟩ **and**
⟨(*K*, *x*) ∈ *Id*⟩ **and**
⟨*K* ∈ *Collect* ((=) *x2*)⟩ **and**
⟨*x* ∈ {*K*. *K* ∈ *set* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)}⟩ **and**

fx′: ⟨(*f*, *x′*) ∈ *?find-unw x1*⟩ **and**
⟨*unit-prop-body-wl-find-unwatched-inv f x1* (*keep-watch L j w S*)⟩ **and**
f: ⟨*f* = *Some xa*⟩ **and**
x′: ⟨*x′* = *Some x′a*⟩ **and**
xa: ⟨(*xa*, *x′a*) ∈ *nat-rel*⟩ **and**
⟨*x′a* < *length* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch L j w S*)) (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *xa*) = *Some True*⟩ **and**
pol: ⟨*polarity* (*get-trail-l* (*fst X2*)) (*get-clauses-l* (*fst X2*) ∝ *snd X2* ! *x′a*) = *Some True*⟩
for *b x1 x2 X2 K x f x′ xa x′a x2′ x3*
proof −
have *confl*: ⟨*get-conflict-wl S* = *None*⟩
using *S-S′ loop-inv cond* **unfolding** *unit-propagation-inner-loop-l-inv-def prod.case* **apply** −
by *normalize-goal*+ *auto*

have *unit-T*: ⟨*unit-propagation-inner-loop-body-l-inv L C′ T*⟩
using *that* **by** (*auto simp*: *remove-one-lit-from-wq-def*)

have ⟨*correct-watching-except* (*Suc j*) (*Suc w*) *L* (*keep-watch L j w S*)⟩
by (*simp add*: *corr-w correct-watching-except-correct-watching-except-Suc-Suc-keep-watch j-w w-le*)
moreover have ⟨*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*a*, *b*, *None*, *d*, *e*, *f*, *ga*(*L* := (*ga L*)[*j* := (*x1*, *b* ∝ *x1* ! *xa*, *x3*)]))⟩
if
corr: ⟨*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*a*, *b*, *None*, *d*, *e*, *f*, *ga*(*L* := (*ga L*)[*j* := (*x1*, *x2*, *x3*)]))⟩ **and**
⟨*ga L* ! *w* = (*x1*, *x2*, *x3*)⟩ **and**
S[*simp*]: ⟨*S* = (*a*, *b*, *None*, *d*, *e*, *f*, *ga*)⟩ **and**
⟨*X2* = (*set-clauses-to-update-l* (*remove1-mset x1* (*clauses-to-update-l S′*)) *S′*, *x1*)⟩ **and**
⟨(*a*, *b*, *None*, *d*, *e*,
{#*i* ∈# *mset* (*drop* (*Suc w*) (*map fst* ((*ga L*)[*j* := (*x1*, *x2*, *x3*)]))). *i* ∈# *dom-m b*#}, *f*) =
set-clauses-to-update-l (*remove1-mset x1* (*clauses-to-update-l S′*)) *S′*⟩
for *a* :: ⟨(′*v literal*, ′*v literal*,*nat*) *annotated-lit list*⟩ **and**
b :: ⟨(*nat*, ′*v literal list* × *bool*) *fmap*⟩ **and**
d :: ⟨′*v literal multiset multiset*⟩ **and**
e :: ⟨′*v literal multiset multiset*⟩ **and**
f :: ⟨′*v literal multiset*⟩ **and**
ga :: ⟨′*v literal* ⇒ (*nat* × ′*v literal* × *bool*) *list*⟩
proof −
have ⟨*b* ∝ *x1* ! *xa* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf b* + (*d* + *e*))⟩
using *dom fx′* **by** (*auto simp*: *ran-m-def all-lits-of-mm-add-mset x′ f twl-st-wl*
dest!: *multi-member-split*
intro!: *in-clause-in-all-lits-of-m*)
moreover have ⟨*b* ∝ *x1* ! *xa* ∈ *set* (*b* ∝ *x1*)⟩
using *dom fx′* **by** (*auto simp*: *ran-m-def all-lits-of-mm-add-mset x′ f twl-st-wl*
dest!: *multi-member-split*
intro!: *in-clause-in-all-lits-of-m*)

moreover have ⟨*b* ∝ *x1* ! *xa* ≠ *L*⟩
using *pol X2 L-def*[*OF unit-T*] *S-S′ SLw fx′ x′ f′ xa* **unfolding** *C′-bl*
by (*auto simp*: *polarity-def split*: *if-splits*)
moreover have ⟨*correctly-marked-as-binary b* (*x1*, *b* ∝ *x1* ! *xa*, *x3*)⟩
using *correctly-marked-as-binary unit-T C′-bl x2′ C′bl dom SLw* **by** (*auto simp*: *correctly-marked-as-binary.simps*)
ultimately show *?thesis*
by (*rule correct-watching-except-update-blit*[*OF corr*])
qed

ultimately have ‹*update-blit-wl L x1 x3 j w* (*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *xa*) (*keep-watch L j w S*)
≤ *SPEC*(λ(*i*, *j*, *T'*). *correct-watching-except i j L T'*)›
using *X2 confl SLw* **unfolding** *C'-bl*
apply (*cases S*)
by (*auto simp*: *keep-watch-def state-wl-l-def x2'*
update-blit-wl-def)
moreover have ‹*get-conflict-wl S* = *None*›
using *S-S' loop-inv cond* **unfolding** *unit-propagation-inner-loop-l-inv-def prod.case* **apply** −
by *normalize-goal+ auto*
moreover have ‹*n* = *size* {#(*i*, -) ∈# *mset* (*drop* (*Suc w*) (*watched-by S L*)). *i* ∉# *dom-m* (*get-clauses-wl S*)#}›
using *n dom X2 w-le S-S' SLw* **unfolding** *C'-bl*
by (*auto simp*: *Cons-nth-drop-Suc*[*symmetric*])
ultimately show *?thesis*
using *j-w w-le S-S' X2*
by (*cases S*)
(*auto simp*: *update-blit-wl-def keep-watch-def state-wl-l-def drop-map*)
qed
have *update-clss-final*: ‹*update-clause-wl L x1 x3 j w*
(*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*) *xa*
(*keep-watch L j w S*)
≤ ⇓ *?unit*
(*update-clause-l* (*snd X2*)
(*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*) *x'a* (*fst X2*) ≫=
(λ*T*. *RETURN* (*T*, *if get-conflict-l T* = *None then n else 0*)))›
if
cond: ‹*clauses-to-update-l S'* ≠ {#} ∨ *0* < *n*› **and**
loop-inv: ‹*unit-propagation-inner-loop-l-inv L* (*S'*, *n*)› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*)› **and**
‹((*C'*, *bL*), *b*) ∈ *?blit*› **and**
C'-bl: ‹(*C'*, *bL*) = (*x1*, *x2'*)› **and**
x2': ‹*x2'* =(*x2*, *x3*)› **and**
dom: ‹¬ *x1* ∉# *dom-m* (*get-clauses-wl S*)› **and**
‹¬ *b*› **and**
‹*clauses-to-update-l S'* ≠ {#}› **and**
X2: ‹(*keep-watch L j w S*, *X2*) ∈ *?keep-watch*› **and**
wl-inv: ‹*unit-prop-body-wl-inv* (*keep-watch L j w S*) *j w L*› **and**
‹(*K*, *x*) ∈ *Id*› **and**
‹*K* ∈ *Collect* ((=) *x2*)› **and**
‹*x* ∈ {*K*. *K* ∈ *set* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)}› **and**
‹*polarity* (*get-trail-wl* (*keep-watch L j w S*)) *K* ≠ *Some True*› **and**
‹*polarity* (*get-trail-l* (*fst X2*)) *x* ≠ *Some True*› **and**
‹*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* − (*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*))) ≠
Some True› **and**
‹*polarity* (*get-trail-l* (*fst X2*))
(*get-clauses-l* (*fst X2*) ∝ *snd X2* !
(*1* − (*if get-clauses-l* (*fst X2*) ∝ *snd X2* ! *0* = *L then 0 else 1*))) ≠
Some True› **and**
fx': ‹(*f*, *x'*) ∈ *?find-unw x1*› **and**
‹*unit-prop-body-wl-find-unwatched-inv f x1* (*keep-watch L j w S*)› **and**
f: ‹*f* = *Some xa*› **and**
x': ‹*x'* = *Some x'a*› **and**
xa: ‹(*xa*, *x'a*) ∈ *nat-rel*› **and**

‹$x'a$ < *length* (*get-clauses-l* (*fst X2*) ∝ *snd X2*)› **and**
‹*polarity* (*get-trail-wl* (*keep-watch L j w S*))
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *xa*) ≠
Some True› **and**
pol: ‹*polarity* (*get-trail-l* (*fst X2*)) (*get-clauses-l* (*fst X2*) ∝ *snd X2* ! $x'a$) ≠ *Some True*› **and**
‹*unit-propagation-inner-loop-body-l-inv L* (*snd X2*) (*fst X2*)›
for *b x1 x2 X2 K x f* x' *xa* $x'a$ $x2'$ *x3*
proof −
have *confl*: ‹*get-conflict-wl S* = *None*›
using *S-S'* *loop-inv cond* **unfolding** *unit-propagation-inner-loop-l-inv-def prod.case* **apply** −
by *normalize-goal+ auto*

then obtain *M N NE UE Q W* **where**
S: ‹*S* = (*M*, *N*, *None*, *NE*, *UE*, *Q*, *W*)›
by (*cases S*) (*auto simp*: *twl-st-l*)
have *dom'*: ‹*x1* ∈# *dom-m* (*get-clauses-wl* (*keep-watch L j w S*)) ⟷ *True*›
using *dom* **by** *auto*
moreover have *watch-by-S-w*: ‹*watched-by* (*keep-watch L j w S*) *L* ! *w* = (*x1*, *x2*, *x3*)›
using *j-w w-le SLw x2'* **unfolding** *i-def C'-bl*
by (*cases S*) (*auto simp*: *keep-watch-def*)
ultimately have *C'-dom*: ‹*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*) ∈# *dom-m* (*get-clauses-wl* (*keep-watch L j w S*)) ⟷ *True*›
using *SLw* **unfolding** *C'-bl* **by** (*auto simp*: *twl-st-wl*)
obtain *x* **where**
S-x: ‹(*keep-watch L j w S*, *x*) ∈ *state-wl-l* (*Some* (*L*, *w*))› **and**
unit-loop-inv:
‹*unit-propagation-inner-loop-body-l-inv L* (*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*))
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*)) *x*)› **and**
L: ‹*L* ∈# *all-lits-of-mm*
(*mset* '# *init-clss-lf* (*get-clauses-wl* (*keep-watch L j w S*)) +
get-unit-clauses-wl (*keep-watch L j w S*))› **and**
‹*correct-watching-except j w L* (*keep-watch L j w S*)› **and**
‹*w* < *length* (*watched-by* (*keep-watch L j w S*) *L*)› **and**
‹*get-conflict-wl* (*keep-watch L j w S*) = *None*›
using *wl-inv* **unfolding** *unit-prop-body-wl-inv-alt-def C'-dom simp-thms* **apply** −
by *blast*
obtain x' **where**
x-x': ‹(*set-clauses-to-update-l*
(*clauses-to-update-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*))
x) +
{#*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*)#})
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*)) *x*),
x') ∈ *twl-st-l* (*Some L*)› **and**
‹*twl-struct-invs* x'› **and**
‹*twl-stgy-invs* x'› **and**
‹*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*)
∈# *dom-m*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*))
x))› **and**
‹*0* < *fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*)› **and**
‹*0* < *length*
(*get-clauses-l*
(*remove-one-lit-from-wq*
(*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*)) *x*) ∝

fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))› **and**
‹*no-dup*
(*get-trail-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))
x))› **and**
ge0: ‹(*if* *get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))
x) ∝
fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*) !
0 =
L
then *0* *else* *1*)
< *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))
x) ∝
fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))› **and**
ge1i: ‹*1* −
(*if* *get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))
x) ∝
fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*) !
0 =
L
then *0* *else* *1*)
< *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))
x) ∝
fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))› **and**
L-watched: ‹*L* ∈ *set* (*watched-l*
(*get-clauses-l*
(*remove-one-lit-from-wq*
(*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*)) *x*) ∝
fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*)))› **and**
‹*get-conflict-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*)) *x*) =
None›
using *unit-loop-inv*
unfolding *unit-propagation-inner-loop-body-l-inv-def*
by *blast*

have [*simp*]: ‹*x′a* = *xa*›
using *xa* **by** *auto*
have *unit-T*: ‹*unit-propagation-inner-loop-body-l-inv* *L* *C′* *T*›
using *that*
by (*auto* *simp*: *remove-one-lit-from-wq-def*)

have *corr*: ‹*correct-watching-except* (*Suc* *j*) (*Suc* *w*) *L* (*keep-watch* *L* *j* *w* *S*)›
by (*simp add*: *corr-w* *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
j-w *w-le*)
have *i*:
‹*i* = (*if* *get-clauses-wl* (*keep-watch* *L* *j* *w* *S*) ∝ *x1* ! *0* = *L* *then* *0* *else* *1*)›
‹*i* = (*if* *get-clauses-l* (*fst* *X2*) ∝ *snd* *X2* ! *0* = *L* *then* *0* *else* *1*)›
using *SLw* *X2* *S-S′* **unfolding** *i-def* *C′-bl* **apply** (*cases* *X2*; *auto* *simp* *add*: *twl-st-wl*; *fail*)
using *SLw* *X2* *S-S′* **unfolding** *i-def* *C′-bl* **apply** (*cases* *X2*; *auto* *simp* *add*: *twl-st-wl*; *fail*)

done
have *i'*: ‹*i* = (*if get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch L j w S*) *L* ! *w*))
x) ∝
fst (*watched-by* (*keep-watch L j w S*) *L* ! *w*) !
0 =
L
then 0 else 1)›
using *j-w w-le S-x* **unfolding** *i-def*
by (*cases S*) (*auto simp*: *keep-watch-def*)
have ‹*twl-st-inv x'*›
using ‹*twl-struct-invs x'*› **unfolding** *twl-struct-invs-def* **by** *fast*
then have ‹∃ *x*. *twl-st-inv*
(*x*, {#*TWL-Clause* (*mset* (*watched-l* (*fst x*)))
(*mset* (*unwatched-l* (*fst x*)))
. *x* ∈# *init-clss-l N*#},
{#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* ∈# *learned-clss-l N*#},
None, *NE*, *UE*,
add-mset
(*L*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *fst* ((*W L*)[*j* := *W L* ! *w*] ! *w*))))
(*mset* (*unwatched-l* (*N* ∝ *fst* ((*W L*)[*j* := *W L* ! *w*] ! *w*)))))
{#(*L*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *x*)))
(*mset* (*unwatched-l* (*N* ∝ *x*))))
. *x* ∈# *remove1-mset* (*fst* ((*W L*)[*j* := *W L* ! *w*] ! *w*))
{#*i* ∈# *mset* (*drop w* (*map fst* ((*W L*)[*j* := *W L* ! *w*]))).
i ∈# *dom-m N*#}#},
Q)›
using *x-x' S-x*
apply (*cases x*)
apply (*auto simp*: *S twl-st-l-def state-wl-l-def keep-watch-def*
simp del: *struct-wf-twl-cls.simps*)
done
then have ‹*Multiset.Ball*
({#*TWL-Clause* (*mset* (*watched-l* (*fst x*))) (*mset* (*unwatched-l* (*fst x*)))
. *x* ∈# *ran-m N*#})
struct-wf-twl-cls›
unfolding *twl-st-inv.simps image-mset-union*[*symmetric*] *all-clss-l-ran-m*
by *blast*
then have *distinct-N-x1*: ‹*distinct* (*N* ∝ *x1*)›
using *dom*
by (*auto simp*: *S ran-m-def mset-take-mset-drop-mset' dest*!: *multi-member-split*)

then have *L-i*: ‹*L* = *N* ∝ *x1* ! *i*›
using *watch-by-S-w L-watched ge0 ge1i SLw S-x* **unfolding** *i-def C'-bl*
by (*auto simp*: *take-2-if twl-st-wl S split*: *if-splits*)
have *i-le*: ‹*i* < *length* (*N* ∝ *x1*)› ‹*1*−*i* < *length* (*N* ∝ *x1*)›
using *watch-by-S-w ge0 ge1i S-x* **unfolding** *i'*[*symmetric*]
by (*auto simp*: *S*)
have *X2*: ‹*X2* = (*set-clauses-to-update-l* (*remove1-mset x1* (*clauses-to-update-l S'*)) *S'*, *x1*)›
using *SLw X2 S-S'* **unfolding** *i-def C'-bl* **by** (*cases X2*; *auto simp add*: *twl-st-wl*)
have ‹*n* = *size* {#(*i*, -) ∈# *mset* (*drop* (*Suc w*) (*watched-by S L*)).
i ≠ *x1* ∧ *i* ∉# *remove1-mset x1* (*dom-m* (*get-clauses-wl S*))#}›
using *dom n w-le SLw* **unfolding** *C'-bl*
by (*auto simp*: *Cons-nth-drop-Suc*[*symmetric*] *dest*!: *multi-member-split*)
moreover have ‹*L* ≠ *get-clauses-wl S* ∝ *x1* ! *xa*›

using *pol X2 L-def*[*OF unit-T*] *S-S′ SLw xa fx′* **unfolding** *C′-bl f x′*
by (*auto simp*: *polarity-def twl-st-wl split*: *if-splits*)
moreover have ‹*remove1-mset x1* {#*i* ∈# *mset* (*drop w* (*map fst* (*watched-by S L*))). *i* ∈# *dom-m* (*get-clauses-wl S*)#} =
{#*i* ∈# *mset* (*drop* (*Suc w*) (*map fst* ((*watched-by S L*)[*j* := (*x1*, *x2*, *x3*)]))). *i* = *x1* ∨ *i* ∈# *remove1-mset x1* (*dom-m* (*get-clauses-wl S*))#}›
using *dom n w-le SLw j-w* **unfolding** *C′-bl*
by (*auto simp*: *Cons-nth-drop-Suc*[*symmetric*] *drop-map dest*!: *multi-member-split*)
moreover have ‹*correct-watching-except j* (*Suc w*) *L*
(*M*, *N*(*x1* ↪ *swap* (*N* ∝ *x1*) *i xa*), *None*, *NE*, *UE*, *Q*, *W*
(*L* := (*W L*)[*j* := (*x1*, *x2*, *x3*)],
N ∝ *x1* ! *xa* := *W* (*N* ∝ *x1* ! *xa*) @ [(*x1*, *L*, *x3*)]))›
apply (*rule correct-watching-except-correct-watching-except-update-clause*)
subgoal
using *corr j-w w-le* **unfolding** *S*
by (*auto simp*: *keep-watch-def*)
subgoal using *j-w* **.**
subgoal using *w-le* **by** (*auto simp*: *S*)
subgoal using *alien-L′*[*OF unit-T*] **by** (*auto simp*: *S twl-st-wl*)
subgoal using *i-le* **unfolding** *L-i* **by** *auto*
subgoal using *L* **by** (*subst all-clss-l-ran-m*[*symmetric*], *subst image-mset-union*)
(*auto simp*: *S all-lits-of-mm-union*)
subgoal using *distinct-N-x1 i-le fx′ xa i-le* **unfolding** *L-i x′*
by (*auto simp*: *S nth-eq-iff-index-eq i-def*)
subgoal using *dom* **by** (*simp add*: *S*)
subgoal using *i-le* **by** *simp*
subgoal using *xa fx′* **unfolding** *f xa* **by** (*auto simp*: *S*)
subgoal using *SLw* **unfolding** *C′-bl* **by** (*auto simp*: *S x2′*)
subgoal unfolding *L-i* **..**
subgoal using *distinct-N-x1 i-le* **unfolding** *L-i*
by (*auto simp*: *nth-eq-iff-index-eq i-def*)
subgoal using *distinct-N-x1 i-le fx′ xa i-le* **unfolding** *L-i x′*
by (*auto simp*: *S nth-eq-iff-index-eq i-def*)
subgoal using *distinct-N-x1 i-le fx′ xa i-le* **unfolding** *L-i x′*
by (*auto simp*: *S nth-eq-iff-index-eq i-def*)
subgoal using *distinct-N-x1 i-le fx′ xa i-le* **unfolding** *L-i x′*
by (*auto simp*: *S nth-eq-iff-index-eq i-def*)
subgoal using *i-def* **by** (*auto simp*: *S split*: *if-splits*)
subgoal using *xa fx′* **unfolding** *f xa* **by** (*auto simp*: *S*)
subgoal using *distinct-N-x1 i-le fx′ xa i-le* **unfolding** *L-i x′*
by (*auto simp*: *S nth-eq-iff-index-eq i-def*)
done
ultimately show *?thesis*
using *S-S′ w-le j-w SLw confl*
unfolding *update-clause-wl-def update-clause-l-def i*[*symmetric*] *C′-bl*
by (*cases S′*)
(*auto simp*: *Let-def X2 keep-watch-def state-wl-l-def S x2′*)
qed
have *blit-final-in-dom*: ‹*update-blit-wl L x1 x3 j w*
(*get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* !
(*1* −
(*if get-clauses-wl* (*keep-watch L j w S*) ∝ *x1* ! *0* = *L then 0 else 1*)))
(*keep-watch L j w S*)
≤ ⇓ *?unit*
(*RETURN* (*fst X2*, *if get-conflict-l* (*fst X2*) = *None then n else 0*))›
if

cond: ‹*clauses-to-update-l S′ ≠ {#} ∨ 0 < n*› **and**
loop-inv: ‹*unit-propagation-inner-loop-l-inv L (S′, n)*› **and**
‹*unit-propagation-inner-loop-wl-loop-pre L (j, w, S)*› **and**
‹*((C′, bL), b) ∈ ?blit*› **and**
C′-bl: ‹*(C′, bL) = (x1, x2′)*› **and**
x2′: ‹*x2′ =(x2, x3)*› **and**
dom: ‹*¬ x1 ∉# dom-m (get-clauses-wl S)*› **and**
‹*¬ b*› **and**
‹*clauses-to-update-l S′ ≠ {#}*› **and**
X2: ‹*(keep-watch L j w S, X2) ∈ ?keep-watch*› **and**
l-inv: ‹*unit-propagation-inner-loop-body-l-inv L (snd X2) (fst X2)*› **and**
wl-inv: ‹*unit-prop-body-wl-inv (keep-watch L j w S) j w L*› **and**
‹*(K, x) ∈ Id*› **and**
‹*K ∈ Collect ((=) x2)*› **and**
‹*x ∈ {K. K ∈ set (get-clauses-l (fst X2) ∝ snd X2)}*› **and**
‹*polarity (get-trail-wl (keep-watch L j w S)) K ≠ Some True*› **and**
‹*polarity (get-trail-l (fst X2)) x ≠ Some True*› **and**
‹*polarity (get-trail-wl (keep-watch L j w S))*
(get-clauses-wl (keep-watch L j w S) ∝ x1 !
(1 −
(if get-clauses-wl (keep-watch L j w S) ∝ x1 ! 0 = L then 0 else 1))) =
Some True› **and**
‹*polarity (get-trail-l (fst X2))*
(get-clauses-l (fst X2) ∝ snd X2 !
(1 − (if get-clauses-l (fst X2) ∝ snd X2 ! 0 = L then 0 else 1))) =
Some True›
for *b x1 x2 X2 K x x2′ x3*
proof −
have *confl*: ‹*get-conflict-wl S = None*›
using *S-S′ loop-inv cond* **unfolding** *unit-propagation-inner-loop-l-inv-def prod.case* **apply** −
by *normalize-goal+ auto*

then obtain *M N NE UE Q W* **where**
S: ‹*S = (M, N, None, NE, UE, Q, W)*›
by *(cases S) (auto simp: twl-st-l)*
have *dom′*: ‹*x1 ∈# dom-m (get-clauses-wl (keep-watch L j w S)) ⟷ True*›
using *dom* **by** *auto*
then have *SLW-dom′*: ‹*fst (watched-by (keep-watch L j w S) L ! w)*
∈# dom-m (get-clauses-wl (keep-watch L j w S))›
using *SLw w-le* **unfolding** *C′-bl* **by** *auto*
have *bin*: ‹*correctly-marked-as-binary N (x1, N ∝ x1 ! (Suc 0 − i), x3)*›
using *X2 correctly-marked-as-binary l-inv x2′ C′-bl*
by *(cases bL)*
(auto simp: S remove-one-lit-from-wq-def correctly-marked-as-binary.simps)

obtain *x* **where**
S-x: ‹*(keep-watch L j w S, x) ∈ state-wl-l (Some (L, w))*› **and**
unit-loop-inv:
‹*unit-propagation-inner-loop-body-l-inv L (fst (watched-by (keep-watch L j w S) L ! w))*
(remove-one-lit-from-wq (fst (watched-by (keep-watch L j w S) L ! w)) x)› **and**
L: ‹*L ∈# all-lits-of-mm*
(mset ‘# init-clss-lf (get-clauses-wl (keep-watch L j w S)) +
get-unit-clauses-wl (keep-watch L j w S))› **and**
‹*correct-watching-except j w L (keep-watch L j w S)*› **and**
‹*w < length (watched-by (keep-watch L j w S) L)*› **and**
‹*get-conflict-wl (keep-watch L j w S) = None*›

using *wl-inv SLW-dom′* **unfolding** *unit-prop-body-wl-inv-alt-def*
by *blast*
obtain x' **where**
x-x′: ‹(*set-clauses-to-update-l*
(*clauses-to-update-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x) +
{#*fst* (*watched-by* (*keep-watch* L j w S) L ! w)#})
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w)) x),
x') ∈ *twl-st-l* (*Some* L)› **and**
‹*twl-struct-invs* x'› **and**
‹*twl-stgy-invs* x'› **and**
‹*fst* (*watched-by* (*keep-watch* L j w S) L ! w)
∈# *dom-m*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x))› **and**
‹*0* < *fst* (*watched-by* (*keep-watch* L j w S) L ! w)› **and**
‹*0* < *length*
(*get-clauses-l*
(*remove-one-lit-from-wq*
(*fst* (*watched-by* (*keep-watch* L j w S) L ! w)) x) ∝
fst (*watched-by* (*keep-watch* L j w S) L ! w))› **and**
‹*no-dup*
(*get-trail-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x))› **and**
ge0: ‹(*if* *get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x) ∝
fst (*watched-by* (*keep-watch* L j w S) L ! w) !
0 =
L
then 0 else 1)
< *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x) ∝
fst (*watched-by* (*keep-watch* L j w S) L ! w))› **and**
ge1i: ‹*1* −
(*if* *get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x) ∝
fst (*watched-by* (*keep-watch* L j w S) L ! w) !
0 =
L
then 0 else 1)
< *length*
(*get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* L j w S) L ! w))
x) ∝
fst (*watched-by* (*keep-watch* L j w S) L ! w))› **and**
L-watched: ‹L ∈ *set* (*watched-l*
(*get-clauses-l*
(*remove-one-lit-from-wq*
(*fst* (*watched-by* (*keep-watch* L j w S) L ! w)) x) ∝

fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*)))⟩ **and**
⟨*get-conflict-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*)) *x*) =
None⟩
using *unit-loop-inv*
unfolding *unit-propagation-inner-loop-body-l-inv-def*
by *blast*

have *unit-T*: ⟨*unit-propagation-inner-loop-body-l-inv* *L* *C′* *T*⟩
using *that*
by (*auto simp*: *remove-one-lit-from-wq-def*)

have *corr*: ⟨*correct-watching-except* (*Suc* *j*) (*Suc* *w*) *L* (*keep-watch* *L* *j* *w* *S*)⟩
by (*simp add*: *corr-w* *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
j-w *w-le*)
have *i*:
⟨*i* = (*if* *get-clauses-wl* (*keep-watch* *L* *j* *w* *S*) ∝ *x1* ! *0* = *L* *then* *0* *else* *1*)⟩
⟨*i* = (*if* *get-clauses-l* (*fst* *X2*) ∝ *snd* *X2* ! *0* = *L* *then* *0* *else* *1*)⟩
using *SLw* *X2* *S-S′* **unfolding** *i-def* *C′-bl* **apply** (*cases* *X2*; *auto simp add*: *twl-st-wl*; *fail*)
using *SLw* *X2* *S-S′* **unfolding** *i-def* *C′-bl* **apply** (*cases* *X2*; *auto simp add*: *twl-st-wl*; *fail*)
done
have *i′*: ⟨*i* = (*if* *get-clauses-l*
(*remove-one-lit-from-wq* (*fst* (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*))
x) ∝
fst (*watched-by* (*keep-watch* *L* *j* *w* *S*) *L* ! *w*) !
0 =
L
then *0* *else* *1*)⟩
using *j-w* *w-le* *S-x* **unfolding** *i-def*
by (*cases* *S*) (*auto simp*: *keep-watch-def*)
have ⟨*twl-st-inv* *x′*⟩
using ⟨*twl-struct-invs* *x′*⟩ **unfolding** *twl-struct-invs-def* **by** *fast*
then have ⟨∃ *x*. *twl-st-inv*
(*x*, {#*TWL-Clause* (*mset* (*watched-l* (*fst* *x*)))
(*mset* (*unwatched-l* (*fst* *x*)))
. *x* ∈# *init-clss-l* *N*#},
{#*TWL-Clause* (*mset* (*watched-l* (*fst* *x*))) (*mset* (*unwatched-l* (*fst* *x*)))
. *x* ∈# *learned-clss-l* *N*#},
None, *NE*, *UE*,
add-mset
(*L*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *fst* ((*W* *L*)[*j* := *W* *L* ! *w*] ! *w*))))
(*mset* (*unwatched-l* (*N* ∝ *fst* ((*W* *L*)[*j* := *W* *L* ! *w*] ! *w*)))))
{#(*L*, *TWL-Clause* (*mset* (*watched-l* (*N* ∝ *x*)))
(*mset* (*unwatched-l* (*N* ∝ *x*))))
. *x* ∈# *remove1-mset* (*fst* ((*W* *L*)[*j* := *W* *L* ! *w*] ! *w*))
{#*i* ∈# *mset* (*drop* *w* (*map* *fst* ((*W* *L*)[*j* := *W* *L* ! *w*]))).
i ∈# *dom-m* *N*#}#},
Q)⟩
using *x-x′* *S-x*
apply (*cases* *x*)
apply (*auto simp*: *S* *twl-st-l-def* *state-wl-l-def* *keep-watch-def*
simp del: *struct-wf-twl-cls.simps*)
done
have ⟨*twl-st-inv* *x′*⟩
using ⟨*twl-struct-invs* *x′*⟩ **unfolding** *twl-struct-invs-def* **by** *fast*
then have ⟨∃ *x*. *twl-st-inv*

(x, {#TWL-Clause (mset (watched-l (fst x)))
      (mset (unwatched-l (fst x)))
    . x ∈# init-clss-l N#},
  {#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
  . x ∈# learned-clss-l N#},
  None, NE, UE,
  add-mset
   (L, TWL-Clause (mset (watched-l (N ∝ fst ((W L)[j := W L ! w] ! w))))
      (mset (unwatched-l (N ∝ fst ((W L)[j := W L ! w] ! w)))))
   {#(L, TWL-Clause (mset (watched-l (N ∝ x)))
        (mset (unwatched-l (N ∝ x))))
   . x ∈# remove1-mset (fst ((W L)[j := W L ! w] ! w))
        {#i ∈# mset (drop w (map fst ((W L)[j := W L ! w]))).
         i ∈# dom-m N#}#},
  Q)›
using *x-x′ S-x*
apply (*cases x*)
apply (*auto simp*: *S twl-st-l-def state-wl-l-def keep-watch-def*
  simp del: *struct-wf-twl-cls.simps*)
done
then have ‹*Multiset.Ball*
  ({#TWL-Clause (mset (watched-l (fst x))) (mset (unwatched-l (fst x)))
  . x ∈# ran-m N#})
  struct-wf-twl-cls›
unfolding *twl-st-inv.simps image-mset-union*[*symmetric*] *all-clss-l-ran-m*
by *blast*
then have *distinct-N-x1*: ‹*distinct* (N ∝ *x1*)›
using *dom*
by (*auto simp*: *S ran-m-def mset-take-mset-drop-mset′ dest*!: *multi-member-split*)

have *watch-by-S-w*: ‹*watched-by* (*keep-watch L j w S*) L ! w = (*x1*, *x2*, *x3*)›
using *j-w w-le SLw* **unfolding** *i-def C′-bl x2′*
by (*cases S*)
  (*auto simp*: *keep-watch-def split*: *if-splits*)
then have *L-i*: ‹L = N ∝ *x1* ! i›
using *L-watched ge0 ge1i SLw S-x* **unfolding** *i-def C′-bl*
by (*auto simp*: *take-2-if twl-st-wl S split*: *if-splits*)
have *i-le*: ‹i < *length* (N ∝ *x1*)› ‹1−i < *length* (N ∝ *x1*)›
using *watch-by-S-w ge0 ge1i S-x* **unfolding** *i′*[*symmetric*]
by (*auto simp*: *S*)
have *X2*: ‹*X2* = (*set-clauses-to-update-l* (*remove1-mset x1* (*clauses-to-update-l S′*)) S′, *x1*)›
using *SLw X2 S-S′* **unfolding** *i-def C′-bl* **by** (*cases X2*; *auto simp add*: *twl-st-wl*)
have *N-x1-in-L*: ‹N ∝ *x1* ! (*Suc 0* − i)
∈# *all-lits-of-mm* ({#*mset* (*fst x*). x ∈# *ran-m* N#} + (NE + UE))›
using *dom i-le* **by** (*auto simp*: *ran-m-def S all-lits-of-mm-add-mset*
  intro!: *in-clause-in-all-lits-of-m*
  dest!: *multi-member-split*)
have ‹((M, N, None, NE, UE, Q, W (L := (W L)[j := (*x1*, N ∝ *x1* ! (*Suc 0* − i), *x3*)])),
  fst X2) ∈ *state-wl-l* (*Some* (L, *Suc w*))›
using *S-S′ X2 j-w w-le SLw* **unfolding** *C′-bl*
apply (*auto simp*: *state-wl-l-def S keep-watch-def drop-map*)
apply (*subst Cons-nth-drop-Suc*[*symmetric*])
apply *auto*[]
apply (*subst* (*asm*) *Cons-nth-drop-Suc*[*symmetric*])
apply *auto*[]
unfolding *mset.simps image-mset-add-mset filter-mset-add-mset*

subgoal premises *p*
using *p*(*1−5*)
by (*auto simp*: *L-i*)
done
moreover have ‹*n* = *size* {#(*i*, -) ∈# *mset* (*drop* (*Suc w*) (*watched-by S L*)).
i ∉# *dom-m* (*get-clauses-wl S*)#}›
using *dom n w-le SLw* **unfolding** *C′-bl*
by (*auto simp*: *Cons-nth-drop-Suc*[*symmetric*] *dest*!: *multi-member-split*)
moreover {
have ‹*Suc 0* − *i* ≠ *i*›
by (*auto simp*: *i-def split*: *if-splits*)
then have ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*M*, *N*, *None*, *NE*, *UE*, *Q*, *W*(*L* := (*W L*)[*j* := (*x1*, *N* ∝ *x1* ! (*Suc 0* − *i*), *x3*)]))›
using *SLw* **unfolding** *C′-bl* **apply** −
apply (*rule correct-watching-except-update-blit*)
using *N-x1-in-L corr i-le distinct-N-x1 i-le bin x2′* **unfolding** *S*
by (*auto simp*: *keep-watch-def L-i nth-eq-iff-index-eq*)
}
ultimately show *?thesis*
using *j-w w-le*
unfolding *i*[*symmetric*]
by (*auto simp*: *S update-blit-wl-def keep-watch-def*)
qed

show *1*: *?propa*
(**is** ‹- ≤ ⇓ *?unit* -›)
supply *trail-keep-w*[*simp*]
unfolding *unit-propagation-inner-loop-body-wl-int-alt-def*
i-def[*symmetric*] *i-def′*[*symmetric*] *unit-propagation-inner-loop-body-l-with-skip-alt-def*
unit-propagation-inner-loop-body-l-def
apply (*rewrite at let* - = *keep-watch* - - - - *in* - *Let-def*)
unfolding *i-def*[*symmetric*] *SLw prod.case*
apply (*rewrite at let* - = - *in let* - = *get-clauses-l* - ∝ - ! - *in* - *Let-def*)
apply (*rewrite* **in** ‹*if* (¬-) *then ASSERT* - >>= - *else* -› *if-not-swap*)
supply *RETURN-as-SPEC-refine*[*refine2 del*]
supply [[*goals-limit=50*]]
apply (*refine-rcg val f f′ keep-watch find-unwatched-l*)
subgoal using *inner-loop-inv w-le j-w*
unfolding *unit-propagation-inner-loop-wl-loop-pre-def* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *w-le* **unfolding** *unit-prop-body-wl-inv-def* **by** *auto*
subgoal using *w-le j-w* **unfolding** *unit-prop-body-wl-inv-def* **by** *auto*
subgoal by (*rule blit-final*)
subgoal unfolding *unit-propagation-inner-loop-wl-loop-pre-def* **by** *fast*
subgoal by *auto*
subgoal by (*rule unit-prop-body-wl-inv*)
apply *assumption+*
subgoal
using *S-S′* **by** *auto*
subgoal
using *S-S′ w-le j-w n confl-S*
by (*auto simp*: *correct-watching-except-correct-watching-except-Suc-Suc-keep-watch*
Cons-nth-drop-Suc[*symmetric*] *corr-w twl-st-wl*)
subgoal
using *S-S′* **by** *auto*
subgoal for *b x1 x2 X2 K x*

by (*rule blit-final-in-dom*)
apply *assumption+*
subgoal for *b x1 x2 X2 K x*
unfolding *unit-prop-body-wl-find-unwatched-inv-def*
by *auto*
subgoal by *auto*
subgoal using *S-S'* **by** (*auto simp*: *twl-st-wl*)
subgoal for *b x1 x2 X2 K x f x'*
by (*rule conflict-final*)
subgoal for *b x1 x2 X2 K x*
by (*rule propa-final*)
subgoal
using *S-S'* **by** *auto*
subgoal for *b x1 x2 X2 K x f x' xa x'a*
by (*rule update-blit-wl-final*)
subgoal for *b x1 x2 X2 K x f x' xa x'a*
by (*rule update-clss-final*)
done

have [*simp*]: ‹*add-mset a* (*remove1-mset a M*) = *M* ⟷ *a* ∈# *M*› **for** *a M*
by (*metis ab-semigroup-add-class.add.commute add.left-neutral multi-self-add-other-not-self remove1-mset-eqE union-mset-add-mset-left*)

show *?eq* **if** *inv*: ‹*unit-propagation-inner-loop-body-l-inv L C' T*›
using *i-le*[*OF inv*] *i-le2*[*OF inv*] *C'-dom*[*OF inv*] *S-S'*
unfolding *i-def*[*symmetric*]
by (*auto simp*: *ran-m-clause-upd image-mset-remove1-mset-if*)
qed

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

defines
[*simp*]: ‹*C''* ≡ *get-clauses-l S'* ∝ *C'*›
assumes
S-S': ‹(*S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *w*))› **and**
w-le: ‹*w* < *length* (*watched-by S L*)› **and**
j-w: ‹*j* ≤ *w*› **and**
corr-w: ‹*correct-watching-except j w L S*› **and**
inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)› **and**
n: ‹*n* = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl S*)) (*mset* (*drop w* (*watched-by S L*))))›
and
confl-S: ‹*get-conflict-wl S* = *None*›
shows *unit-propagation-inner-loop-body-wl-spec*: ‹*unit-propagation-inner-loop-body-wl L j w S* ≤
⇓{((*i*, *j*, *T'*), (*T*, *n*)).
(*T'*, *T*) ∈ *state-wl-l* (*Some* (*L*, *j*)) ∧
correct-watching-except i j L T' ∧
j ≤ *length* (*watched-by T' L*) ∧
length (*watched-by S L*) = *length* (*watched-by T' L*) ∧
i ≤ *j* ∧
(*get-conflict-wl T'* = *None* ⟶
n = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl T'*)) (*mset* (*drop j* (*watched-by T' L*))))) ∧

$(get\text{-}conflict\text{-}wl\ T' \neq None \longrightarrow n = 0)\}$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L\ (S',\ n))$›
apply (*rule order-trans*)
apply (*rule unit-propagation-inner-loop-body-wl-wl-int*[*OF S-S' w-le j-w corr-w inner-loop-inv n confl-S*])
apply (*subst Down-id-eq*)
apply (*rule unit-propagation-inner-loop-body-wl-int-spec*[*OF S-S' w-le j-w corr-w inner-loop-inv n confl-S*])
done

definition *unit-propagation-inner-loop-wl-loop*
:: ‹$'v\ literal \Rightarrow 'v\ twl\text{-}st\text{-}wl \Rightarrow (nat \times nat \times 'v\ twl\text{-}st\text{-}wl)\ nres$› **where**
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\ L\ S_0 = do\ \{$
$let\ n = length\ (watched\text{-}by\ S_0\ L);$
$WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L}$
$(\lambda(j,\ w,\ S).\ w < n \wedge get\text{-}conflict\text{-}wl\ S = None)$
$(\lambda(j,\ w,\ S).\ do\ \{$
$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}wl\ L\ j\ w\ S$
$\})$
$(0,\ 0,\ S_0)$
$\}$›

lemma *correct-watching-except-correct-watching-cut-watch*:
assumes *corr*: ‹$correct\text{-}watching\text{-}except\ j\ w\ L\ (a,\ b,\ c,\ d,\ e,\ f,\ g)$›
shows ‹$correct\text{-}watching\ (a,\ b,\ c,\ d,\ e,\ f,\ g(L := take\ j\ (g\ L)\ @\ drop\ w\ (g\ L)))$›
proof −
have
Heq:
‹$\bigwedge La\ i\ K\ b'.\ La \in\#all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ b + (d + e)) \Longrightarrow$
$(La = L \longrightarrow$
$distinct\text{-}watched\ (take\ j\ (g\ La)\ @\ drop\ w\ (g\ La)) \wedge$
$((i,\ K,\ b')\in\#mset\ (take\ j\ (g\ La)\ @\ drop\ w\ (g\ La)) \longrightarrow$
$i \in\#\ dom\text{-}m\ b \longrightarrow K \in set\ (b \propto i) \wedge K \neq La \wedge correctly\text{-}marked\text{-}as\text{-}binary\ b\ (i,\ K,\ b')) \wedge$
$((i,\ K,\ b')\in\#mset\ (take\ j\ (g\ La)\ @\ drop\ w\ (g\ La)) \longrightarrow$
$b' \longrightarrow i \in\#\ dom\text{-}m\ b) \wedge$
$\{\#i \in\#\ fst\ `\#\ mset\ (take\ j\ (g\ La)\ @\ drop\ w\ (g\ La)).\ i \in\#\ dom\text{-}m\ b\#\} =$
$clause\text{-}to\text{-}update\ La\ (a,\ b,\ c,\ d,\ e,\ \{\#\},\ \{\#\}))$› **and**
Hneq:
‹$\bigwedge La\ i\ K\ b'.\ La\in\#all\text{-}lits\text{-}of\text{-}mm\ (mset\ `\#\ ran\text{-}mf\ b + (d + e)) \Longrightarrow$
$(La \neq L \longrightarrow$
$distinct\text{-}watched\ (g\ La) \wedge$
$((i,\ K,\ b')\in\#mset\ (g\ La) \longrightarrow i \in\#\ dom\text{-}m\ b \longrightarrow K \in set\ (b \propto i) \wedge K \neq La$
$\wedge\ correctly\text{-}marked\text{-}as\text{-}binary\ b\ (i,\ K,\ b')) \wedge$
$((i,\ K,\ b')\in\#mset\ (g\ La) \longrightarrow b' \longrightarrow i \in\#\ dom\text{-}m\ b) \wedge$
$\{\#i \in\#\ fst\ `\#\ mset\ (g\ La).\ i \in\#\ dom\text{-}m\ b\#\} =$
$clause\text{-}to\text{-}update\ La\ (a,\ b,\ c,\ d,\ e,\ \{\#\},\ \{\#\}))$›
using *corr*
unfolding *correct-watching.simps correct-watching-except.simps*
by *fast+*
have
‹$((i,\ K,\ b')\in\#mset\ ((g(L := take\ j\ (g\ L)\ @\ drop\ w\ (g\ L)))\ La) \Longrightarrow$
$i \in\#\ dom\text{-}m\ b \longrightarrow K \in set\ (b \propto i) \wedge K \neq La \wedge correctly\text{-}marked\text{-}as\text{-}binary\ b\ (i,\ K,\ b'))$› **and**
‹$(i,\ K,\ b')\in\#mset\ ((g(L := take\ j\ (g\ L)\ @\ drop\ w\ (g\ L)))\ La) \Longrightarrow$

$b' \longrightarrow i \in\# \ dom\text{-}m\ b$› **and**
‹$\{\#i \in\# \ fst$ ‘# $mset\ ((g(L := take\ j\ (g\ L)\ @\ drop\ w\ (g\ L)))\ La).$
$i \in\# \ dom\text{-}m\ b\#\} =$
$clause\text{-}to\text{-}update\ La\ (a,\ b,\ c,\ d,\ e,\ \{\#\},\ \{\#\})$›**and**
‹$distinct\text{-}watched\ ((g(L := take\ j\ (g\ L)\ @\ drop\ w\ (g\ L)))\ La)$›
if ‹$La{\in}\#all\text{-}lits\text{-}of\text{-}mm\ (mset$ ‘# $ran\text{-}mf\ b + (d + e))$›
for $La\ i\ K\ b'$
apply (*cases* ‹$La = L$›)
subgoal
using *Heq*[*of La i K*] *that* **by** *auto*
subgoal
using *Hneq*[*of La i K*] *that* **by** *auto*
apply (*cases* ‹$La = L$›)
subgoal
using *Heq*[*of La i K*] *that* **by** *auto*
subgoal
using *Hneq*[*of La i K*] *that* **by** *auto*
apply (*cases* ‹$La = L$›)
subgoal
using *Heq*[*of La i K*] *that* **by** *auto*
subgoal
using *Hneq*[*of La i K*] *that* **by** *auto*
apply (*cases* ‹$La = L$›)
subgoal
using *Heq*[*of La i K*] *that* **by** *auto*
subgoal
using *Hneq*[*of La i K*] *that* **by** *auto*
done
then show *?thesis*
unfolding *correct-watching.simps*
by *blast*
qed

lemma *unit-propagation-inner-loop-wl-loop-alt-def*:
‹*unit-propagation-inner-loop-wl-loop* $L\ S_0 =$ *do* {
let (- :: *nat*) = (*if get-conflict-wl* S_0 = *None then remaining-nondom-wl 0* $L\ S_0$ *else 0*);
let n = *length* (*watched-by* $S_0\ L$);
$WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L}$
$(\lambda(j,\ w,\ S).\ w < n \wedge get\text{-}conflict\text{-}wl\ S = None)$
$(\lambda(j,\ w,\ S).$ *do* {
unit-propagation-inner-loop-body-wl L j w S
})
$(0,\ 0,\ S_0)$
}
›
unfolding *unit-propagation-inner-loop-wl-loop-def Let-def* **by** *auto*

definition *cut-watch-list* :: ‹*nat* ⇒ *nat* ⇒ *'v literal* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*cut-watch-list j w L* =($\lambda(M,\ N,\ D,\ NE,\ UE,\ Q,\ W)$. *do* {
$ASSERT(j \le w \wedge j \le length\ (W\ L) \wedge w \le length\ (W\ L))$;
$RETURN\ (M,\ N,\ D,\ NE,\ UE,\ Q,\ W(L := take\ j\ (W\ L)\ @\ drop\ w\ (W\ L)))$
})›

definition *unit-propagation-inner-loop-wl* :: ‹*'v literal* ⇒ *'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*unit-propagation-inner-loop-wl* $L\ S_0 =$ *do* {
$(j,\ w,\ S) \leftarrow$ *unit-propagation-inner-loop-wl-loop* $L\ S_0$;

ASSERT($j \leq w \wedge w \leq$ *length* (*watched-by S L*));
cut-watch-list j w L S
})

lemma *correct-watching-correct-watching-except00*:
‹*correct-watching S* $\Longrightarrow$ *correct-watching-except 0 0 L S*›
apply (*cases S*)
apply (*simp only*: *correct-watching.simps correct-watching-except.simps take0 drop0 append.left-neutral*)
by *fast*

lemma *unit-propagation-inner-loop-wl-spec*:
shows ‹(*uncurry unit-propagation-inner-loop-wl*, *uncurry unit-propagation-inner-loop-l*) $\in$
$\{((L', T'::'v\ twl\text{-}st\text{-}wl), (L, T::'v\ twl\text{-}st\text{-}l)).\ L = L' \wedge (T', T) \in state\text{-}wl\text{-}l\ (Some\ (L, 0)) \wedge$
$correct\text{-}watching\ T'\} \rightarrow$
$\langle\{(T', T).\ (T', T) \in state\text{-}wl\text{-}l\ None \wedge correct\text{-}watching\ T'\}\rangle\ nres\text{-}rel$
› (**is** ‹$?fg \in\ ?A \rightarrow \langle ?B\rangle nres\text{-}rel$› **is** ‹$?fg \in\ ?A \rightarrow \langle\{(T', T).\ \text{-} \wedge\ ?P\ T\ T'\}\rangle nres\text{-}rel$›)
proof −
{
fix L :: ‹'v literal› **and** S :: ‹'v twl-st-wl› **and** S' :: ‹'v twl-st-l›
assume
corr-w: ‹*correct-watching S*› **and**
SS': ‹$(S, S') \in state\text{-}wl\text{-}l\ (Some\ (L, 0))$›

To ease the finding the correspondence between the body of the loops, we introduce following function:

let $?R'$ = ‹$\{((i, j, T'), (T, n)).$
$(T', T) \in state\text{-}wl\text{-}l\ (Some\ (L, j)) \wedge$
$correct\text{-}watching\text{-}except\ i\ j\ L\ T' \wedge$
$j \leq length\ (watched\text{-}by\ T'\ L) \wedge$
$length\ (watched\text{-}by\ S\ L) = length\ (watched\text{-}by\ T'\ L) \wedge$
$i \leq j \wedge$
$(get\text{-}conflict\text{-}wl\ T' = None \longrightarrow$
$n = size\ (filter\text{-}mset\ (\lambda(i, \text{-}).\ i \notin\#\ dom\text{-}m\ (get\text{-}clauses\text{-}wl\ T'))\ (mset\ (drop\ j\ (watched\text{-}by\ T'\ L))))) \wedge$
$(get\text{-}conflict\text{-}wl\ T' \neq None \longrightarrow n = 0)\}$›
have *inv*: ‹*unit-propagation-inner-loop-wl-loop-inv L iT'*›
if
iT'-Tn: ‹$(iT', Tn) \in\ ?R'$› **and**
‹*unit-propagation-inner-loop-l-inv L Tn*›
for *Tn iT'*
proof −
obtain *i j* :: *nat* **and** T' **where** *iT'*: ‹$iT' = (i, j, T')$› **by** (*cases iT'*)
obtain *T n* **where** *Tn*[*simp*]: ‹$Tn = (T, n)$› **by** (*cases Tn*)
have ‹*unit-propagation-inner-loop-l-inv L* (*T*, *0*::*nat*)›
if ‹*unit-propagation-inner-loop-l-inv L* (*T*, *n*)› **and** ‹*get-conflict-l T* $\neq$ *None*›
using *that iT'-Tn*
unfolding *unit-propagation-inner-loop-l-inv-def iT' prod.case*
apply − **apply** *normalize-goal+*
apply (*rule-tac x=x* **in** *exI*)
by *auto*
then show *?thesis*
unfolding *unit-propagation-inner-loop-wl-loop-inv-def iT' prod.simps* **apply** −
apply (*rule exI*[*of - T*])
using *that* **by** (*auto simp*: *iT'*)
qed

have *cond*: ‹$(j < length\ (watched\text{-}by\ S\ L) \wedge get\text{-}conflict\text{-}wl\ T' = None) =$
$(clauses\text{-}to\text{-}update\text{-}l\ T \neq \{\#\} \vee n > 0)$›
if
iT′-T: ‹$((ijT', Tn) \in ?R$› **and**
[*simp*]: ‹$ijT' = (i, jT')$› ‹$jT' = (j, T')$› ‹$Tn = (T, n)$›
for *ijT′ Tn i j T′ n T jT′*
proof −
have [*simp*]: ‹$size\ \{\#(i, \text{-}) \in\# mset\ (drop\ j\ xs).\ i \notin\# dom\text{-}m\ b\#\} =$
$size\ \{\#i \in\# fst\ `\#\ mset\ (drop\ j\ xs).\ i \notin\# dom\text{-}m\ b\#\}$› **for** *xs b*
apply (*induction* ‹*xs*› *arbitrary*: *j*)
subgoal by *auto*
subgoal premises *p* **for** *a xs j*
using *p*[*of 0*] *p*
by (*cases j*) *auto*
done
have [*simp*]: ‹$size\ (filter\text{-}mset\ (\lambda i.\ (i \in\# (dom\text{-}m\ b)))\ (fst\ `\#\ (mset\ (drop\ j\ (g\ L))))) +$
$size\ \{\#i \in\# fst\ `\#\ mset\ (drop\ j\ (g\ L)).\ i \notin\# dom\text{-}m\ b\#\} =$
$length\ (g\ L) - j$› **for** *g j b*
apply (*subst size-union*[*symmetric*])
apply (*subst multiset-partition*[*symmetric*])
by *auto*
have [*simp*]: ‹$A \neq \{\#\} \Longrightarrow size\ A > 0$› **for** *A*
by (*auto dest*!: *multi-member-split*)
have ‹$length\ (watched\text{-}by\ T'\ L) = size\ (clauses\text{-}to\text{-}update\text{-}wl\ T'\ L\ j) + n + j$›
if ‹$get\text{-}conflict\text{-}wl\ T' = None$›
using *that iT′-T*
by (*cases* ‹*get-conflict-wl T′*›; *cases T′*)
(*auto simp add*: *state-wl-l-def drop-map*)
then show *?thesis*
using *iT′-T*
by (*cases* ‹$get\text{-}conflict\text{-}wl\ T' = None$›) *auto*
qed
have *remaining*: ‹$RETURN\ (if\ get\text{-}conflict\text{-}wl\ S = None\ then\ remaining\text{-}nondom\text{-}wl\ 0\ L\ S\ else\ 0)$
$\leq SPEC\ (\lambda\text{-}.\ True)$›
by *auto*

have *unit-propagation-inner-loop-l-alt-def*: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\ L\ S' = do\ \{$
$n \leftarrow SPEC\ (\lambda\text{-}::nat.\ True);$
$(S, n) \leftarrow WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L}$
$(\lambda(S, n).\ clauses\text{-}to\text{-}update\text{-}l\ S \neq \{\#\} \vee 0 < n)$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\text{-}with\text{-}skip\ L)\ (S', n);$
$RETURN\ S\}$› **for** *L S′*
unfolding *unit-propagation-inner-loop-l-def* **by** *auto*
have *unit-propagation-inner-loop-wl-alt-def*: ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\ L\ S = do\ \{$
$let\ (n::nat) = (if\ get\text{-}conflict\text{-}wl\ S = None\ then\ remaining\text{-}nondom\text{-}wl\ 0\ L\ S\ else\ 0);$
$(j, w, S) \leftarrow WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}inv\ L}$
$(\lambda(j, w, T).\ w < length\ (watched\text{-}by\ S\ L) \wedge get\text{-}conflict\text{-}wl\ T = None)$
$(\lambda(j, x, y).\ unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}wl\ L\ j\ x\ y)\ (0, 0, S);$
$ASSERT\ (j \leq w \wedge w \leq length\ (watched\text{-}by\ S\ L));$
$cut\text{-}watch\text{-}list\ j\ w\ L\ S\}$›
unfolding *unit-propagation-inner-loop-wl-loop-alt-def unit-propagation-inner-loop-wl-def*
by *auto*
have ‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\ L\ S \leq$
$\Downarrow \{((T'), T).\ (T', T) \in state\text{-}wl\text{-}l\ None \wedge ?P\ T\ T'\}$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\ L\ S')$›

(**is** ‹- ≤ ⇓ *?R* -›)
unfolding *unit-propagation-inner-loop-l-alt-def uncurry-def*
unit-propagation-inner-loop-wl-alt-def
apply (*refine-vcg WHILEIT-refine-genR*[**where**
$R' = ‹?R'›$ **and**
$R = ‹\{((i, j, T'), (T, n)). ((i, j, T'), (T, n)) \in ?R' \wedge i \leq j \wedge$
$length\ (watched\text{-}by\ S\ L) = \ length\ (watched\text{-}by\ T'\ L) \wedge$
$(j \geq length\ (watched\text{-}by\ T'\ L) \vee get\text{-}conflict\text{-}wl\ T' \neq None)\}›]$
remaining)
subgoal using *corr-w SS'* **by** (*auto simp*: *correct-watching-correct-watching-except00*)
subgoal by (*rule inv*)
subgoal by (*rule cond*)
subgoal for $n\ i'w'T'\ Tn\ i'\ w'T'\ w'\ T'$
apply (*cases Tn*)
apply (*rule order-trans*)
apply (*rule unit-propagation-inner-loop-body-wl-spec*[*of* - ‹*fst Tn*›])
apply (*simp only*: *prod.case in-pair-collect-simp*)
apply *normalize-goal*+
by (*auto simp del*: *twl-st-of-wl.simps*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for $n\ i'w'T'\ Tn\ i'\ w'T'\ j\ L'\ w'\ T'$
apply (*cases* T')
by (*auto simp*: *state-wl-l-def cut-watch-list-def*
dest!: *correct-watching-except-correct-watching-cut-watch*)
done
}
note $H = this$

show *?thesis*
unfolding *fref-param1*
apply (*intro frefI nres-relI*)
by (*auto simp*: *intro*!: *H*)
qed

Outer loop

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

definition *unit-propagation-outer-loop-wl-inv* **where**
‹*unit-propagation-outer-loop-wl-inv* $S \longleftrightarrow$
$(\exists S'. (S, S') \in state\text{-}wl\text{-}l\ None \wedge$
$correct\text{-}watching\ S \wedge$
$unit\text{-}propagation\text{-}outer\text{-}loop\text{-}l\text{-}inv\ S')$›

definition *unit-propagation-outer-loop-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*unit-propagation-outer-loop-wl* S_0 =
$WHILE_T{}^{unit\text{-}propagation\text{-}outer\text{-}loop\text{-}wl\text{-}inv}$
$(\lambda S.\ literals\text{-}to\text{-}update\text{-}wl\ S \neq \{\#\})$
$(\lambda S.\ do\ \{$
$ASSERT(literals\text{-}to\text{-}update\text{-}wl\ S \neq \{\#\});$
$(S', L) \leftarrow select\text{-}and\text{-}remove\text{-}from\text{-}literals\text{-}to\text{-}update\text{-}wl\ S;$

$ASSERT(L \in\# \textit{all-lits-of-mm } (\textit{mset } `\# \textit{ ran-mf } (\textit{get-clauses-wl } S') + \textit{get-unit-clauses-wl } S'));$
unit-propagation-inner-loop-wl L S'
})
(S_0 :: $'v$ *twl-st-wl*)
⟩

lemma *unit-propagation-outer-loop-wl-spec*:
⟨(*unit-propagation-outer-loop-wl*, *unit-propagation-outer-loop-l*)
∈ {(T'::$'v$ *twl-st-wl*, T).
(T', T) ∈ *state-wl-l None* ∧
correct-watching T'} $\rightarrow_f$
⟨{(T', T).
(T', T) ∈ *state-wl-l None* ∧
correct-watching T'}⟩*nres-rel*⟩
(**is** ⟨*?u* ∈ *?A* $\rightarrow_f$ ⟨*?B*⟩ *nres-rel*⟩)
proof −
have *inv*: ⟨*unit-propagation-outer-loop-wl-inv* T'⟩
if
⟨(T', T) ∈ {(T', T). (T', T) ∈ *state-wl-l None* ∧ *correct-watching* T'}⟩ **and**
⟨*unit-propagation-outer-loop-l-inv* T⟩
for T T'
unfolding *unit-propagation-outer-loop-wl-inv-def*
apply (*rule exI*[*of* - T])
using *that* **by** *auto*

have *select-and-remove-from-literals-to-update-wl*:
⟨*select-and-remove-from-literals-to-update-wl* S' ≤
⇓ {((T', L'), (T, L)). $L = L'$ ∧ (T', T) ∈ *state-wl-l* (*Some* (L, *0*)) ∧
T' = *set-literals-to-update-wl* (*literals-to-update-wl* S' − {#L#}) S' ∧ L ∈# *literals-to-update-wl*
S' ∧
L ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl* S') + *get-unit-clauses-wl* S')
}
(*select-and-remove-from-literals-to-update* S)⟩
if S: ⟨(S', S) ∈ *state-wl-l None*⟩ **and** ⟨*get-conflict-wl* S' = *None*⟩ **and**
corr-w: ⟨*correct-watching* S'⟩ **and**
inv-l: ⟨*unit-propagation-outer-loop-l-inv* S⟩
for S :: ⟨$'v$ *twl-st-l*⟩ **and** S' :: ⟨$'v$ *twl-st-wl*⟩
proof −
obtain M N D NE UE W Q **where**
S': ⟨S' = (M, N, D, NE, UE, Q, W)⟩
by (*cases* S') *auto*
obtain R **where**
S-R: ⟨(S, R) ∈ *twl-st-l None*⟩ **and**
struct-invs: ⟨*twl-struct-invs* R⟩
using *inv-l* **unfolding** *unit-propagation-outer-loop-l-inv-def* **by** *blast*
have [*simp*]:
⟨*init-clss* ($state_W$*-of* R) = *mset* ‘# (*init-clss-lf* N) + NE⟩
using *S-R* S **by** (*auto simp*: *twl-st* S' *twl-st-wl*)
have
no-dup-q: ⟨*no-duplicate-queued* R⟩ **and**
alien: ⟨$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of* R)⟩
using *struct-invs that* **by** (*auto simp*: *twl-struct-invs-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*)
then have *H1*: ⟨L ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf* N + NE + UE)⟩ **if** *LQ*: ⟨L ∈# Q⟩ **for** L
proof −

have [*simp*]: ‹(*f o g*) ‘ *I* = *f* ‘ *g* ‘ *I*› **for** *f g I*
by *auto*
obtain *K* **where** ‹*L* = − *lit-of K*› **and** ‹*K* ∈# *mset* (*trail* ($state_W$*-of R*))›
using *that no-dup-q LQ S-R S*
mset-le-add-mset-decr-left2[*of L* ‹*remove1-mset L Q*› *Q*]
by (*fastforce simp*: *S′* $cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
all-lits-of-mm-def atms-of-ms-def twl-st-l-def state-wl-l-def uminus-lit-swap
convert-lit.simps
dest!: *multi-member-split*[*of L Q*] *mset-subset-eq-insertD in-convert-lits-lD2*)
from *imageI*[*OF this*(*2*), *of* ‹*atm-of o lit-of*›]
have ‹*atm-of L* ∈ *atm-of* ‘ *lits-of-l* (*get-trail-wl S′*)› **and**
[*simp*]: ‹*atm-of* ‘ *lits-of-l* (*trail* ($state_W$*-of R*)) = *atm-of* ‘ *lits-of-l* (*get-trail-wl S′*)›
using *S-R S S* ‹*L* = − *lit-of K*›
by (*simp-all add*: *twl-st image-image*[*symmetric*]
lits-of-def[*symmetric*])
then have ‹*atm-of L* ∈ *atm-of* ‘ *lits-of-l M*›
using *S′* **by** *auto*
moreover {
have ‹*atm-of* ‘ *lits-of-l M*
⊆ (⋃*x*∈*set-mset* (*init-clss-lf N*). *atm-of* ‘ *set x*) ∪
(⋃*x*∈*set-mset NE*. *atms-of x*) ›
using *that alien* **unfolding** $cdcl_W$*-restart-mset.no-strange-atm-def*
by (*auto simp*: *S′* $cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
all-lits-of-mm-def atms-of-ms-def)
then have ‹*atm-of* ‘ *lits-of-l M* ⊆ (⋃*x*∈*set-mset* (*init-clss-lf N*). *atm-of* ‘ *set x*) ∪
(⋃*x*∈*set-mset NE*. *atms-of x*)›
unfolding *image-Un*[*symmetric*]
set-append[*symmetric*]
append-take-drop-id
.
then have ‹*atm-of* ‘ *lits-of-l M* ⊆ *atms-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*)›
by (*smt UN-Un Un-iff append-take-drop-id atms-of-ms-def atms-of-ms-mset-unfold set-append*
set-image-mset set-mset-mset set-mset-union subset-eq)
}
ultimately have ‹*atm-of L* ∈ *atms-of-mm* (*mset* ‘# *ran-mf N* + *NE*)›
using *that*
unfolding *all-lits-of-mm-union atms-of-ms-union all-clss-lf-ran-m*[*symmetric*]
image-mset-union set-mset-union
by *auto*
then show *?thesis*
using *that* **by** (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff*)
qed
have *H*: ‹*clause-to-update L S* = {#*i* ∈# *fst* ‘# *mset* (*W L*). *i* ∈# *dom-m N*#}› **and**
‹*L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + *NE* + *UE*)›
if ‹*L* ∈# *Q*› **for** *L*
using *corr-w that S H1*[*OF that*] **by** (*auto simp*: *correct-watching.simps S′ clause-to-update-def*
Ball-def ac-simps all-conj-distrib
dest!: *multi-member-split*)
show *?thesis*
unfolding *select-and-remove-from-literals-to-update-wl-def select-and-remove-from-literals-to-update-def*
apply (*rule RES-refine*)
unfolding *Bex-def*
apply (*rule-tac x*=‹(*set-clauses-to-update-l* (*clause-to-update* (*snd s*) *S*)
(*set-literals-to-update-l*
(*remove1-mset* (*snd s*) (*literals-to-update-l S*)) *S*), *snd s*)› **in** *exI*)
using *that S′ S* **by** (*auto 5 5 simp*: *correct-watching.simps clauses-def state-wl-l-def*

mset-take-mset-drop-mset′ cdcl$_W$-restart-mset-state all-lits-of-mm-union
dest: *H H1*)
qed
have *conflict-None*: ‹*get-conflict-wl T* = *None*›
if
‹*literals-to-update-wl T* ≠ {#}› **and**
inv1: ‹*unit-propagation-outer-loop-wl-inv T*›
for *T*
proof −
obtain *T′* **where**
2: ‹(*T*, *T′*) ∈ *state-wl-l None*› **and**
inv2: ‹*unit-propagation-outer-loop-l-inv T′*›
using *inv1* **unfolding** *unit-propagation-outer-loop-wl-inv-def* **by** *blast*
obtain *T′′* **where**
3: ‹(*T′*, *T′′*) ∈ *twl-st-l None*› **and**
‹*twl-struct-invs T′′*›
using *inv2* **unfolding** *unit-propagation-outer-loop-l-inv-def* **by** *blast*
then have ‹*get-conflict T′′* ≠ *None* ⟶
clauses-to-update T′′ = {#} ∧ *literals-to-update T′′* = {#}›
unfolding *twl-struct-invs-def* **by** *fast*
then show *?thesis*
using *that 2 3* **by** (*auto simp*: *twl-st-wl twl-st twl-st-l*)
qed
show *?thesis*
unfolding *unit-propagation-outer-loop-wl-def unit-propagation-outer-loop-l-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg select-and-remove-from-literals-to-update-wl*
unit-propagation-inner-loop-wl-spec[*unfolded fref-param1*, *THEN fref-to-Down-curry*])
subgoal by (*rule inv*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule conflict-None*)
subgoal for *T′ T* **by** (*auto simp*:)
subgoal by (*auto simp*: *twl-st-wl*)
subgoal by *auto*
done
qed

Decide or Skip

definition *find-unassigned-lit-wl* :: ‹*′v twl-st-wl* ⇒ *′v literal option nres*› **where**
‹*find-unassigned-lit-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
SPEC (λ*L*.
(*L* ≠ *None* ⟶
undefined-lit M (*the L*) ∧
atm-of (*the L*) ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf N* + *NE*)) ∧
(*L* = *None* ⟶ (∄ *L′*. *undefined-lit M L′* ∧
atm-of L′ ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf N* + *NE*))))
)›

definition *decide-wl-or-skip-pre* **where**
‹*decide-wl-or-skip-pre S* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *state-wl-l None* ∧
decide-l-or-skip-pre S′
)›

definition *decide-lit-wl* :: ‹'*v literal* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl*› **where**
‹*decide-lit-wl* = (λ*L'* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
(*Decided L'* # *M*, *N*, *D*, *NE*, *UE*, {#− *L'*#}, *W*))›

definition *decide-wl-or-skip* :: ‹'*v twl-st-wl* ⇒ (*bool* × '*v twl-st-wl*) *nres*› **where**
‹*decide-wl-or-skip S* = (*do* {
ASSERT(*decide-wl-or-skip-pre S*);
L ← *find-unassigned-lit-wl S*;
case L of
None ⇒ *RETURN* (*True*, *S*)
| *Some L* ⇒ *RETURN* (*False*, *decide-lit-wl L S*)
})
›

lemma *decide-wl-or-skip-spec*:
‹(*decide-wl-or-skip*, *decide-l-or-skip*)
∈ {(*T'*:: '*v twl-st-wl*, *T*).
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T' ∧
get-conflict-wl T' = *None*} →
⟨{((*b'*, *T'*), (*b*, *T*)). *b'* = *b* ∧
(*T'*, *T*) ∈ *state-wl-l None* ∧
correct-watching T'}⟩*nres-rel*›
proof −
have *find-unassigned-lit-wl*: ‹*find-unassigned-lit-wl S'*
≤ ⇓ *Id*
(*find-unassigned-lit-l S*)›
if ‹(*S'*, *S*) ∈ *state-wl-l None*›
for *S* :: ‹'*v twl-st-l*› **and** *S'* :: ‹'*v twl-st-wl*›
using *that*
by (*cases S'*) (*auto simp*: *find-unassigned-lit-wl-def find-unassigned-lit-l-def*
mset-take-mset-drop-mset' state-wl-l-def)
have *option*: ‹(*x*, *x'*) ∈ ⟨*Id*⟩*option-rel*› **if** ‹*x* = *x'*› **for** *x x'*
using *that* **by** (*auto*)
show *?thesis*
unfolding *decide-wl-or-skip-def decide-l-or-skip-def*
apply (*refine-vcg find-unassigned-lit-wl option*)
subgoal unfolding *decide-wl-or-skip-pre-def* **by** *fast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *S S'*
by (*cases S*) (*auto simp*: *correct-watching.simps clause-to-update-def*
decide-lit-l-def decide-lit-wl-def state-wl-l-def)
done
qed

Skip or Resolve

definition *tl-state-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl*› **where**
‹*tl-state-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). (*tl M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*))›

definition *resolve-cls-wl'* :: ‹'*v twl-st-wl* ⇒ *nat* ⇒ '*v literal* ⇒ '*v clause*› **where**
‹*resolve-cls-wl' S C L* =
remove1-mset L (*remove1-mset* (−*L*) (*the* (*get-conflict-wl S*) ∪# (*mset* (*get-clauses-wl S* ∝ *C*))))›

definition *update-confl-tl-wl* :: ‹*nat* ⇒ *'v literal* ⇒ *'v twl-st-wl* ⇒ *bool* × *'v twl-st-wl*› **where**
‹*update-confl-tl-wl* = (*λC L* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
  let D = *resolve-cls-wl'* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) *C L in*
    (*False*, (*tl M*, *N*, *Some D*, *NE*, *UE*, *WS*, *Q*)))›

definition *skip-and-resolve-loop-wl-inv* :: ‹*'v twl-st-wl* ⇒ *bool* ⇒ *'v twl-st-wl* ⇒ *bool*› **where**
‹*skip-and-resolve-loop-wl-inv* S_0 *brk S* ⟷
 (∃ *S' S'*$_0$. (*S*, *S'*) ∈ *state-wl-l None* ∧
  (S_0, *S'*$_0$) ∈ *state-wl-l None* ∧
  skip-and-resolve-loop-inv-l S'$_0$ *brk S'* ∧
    correct-watching S)›

definition *skip-and-resolve-loop-wl* :: ‹*'v twl-st-wl* ⇒ *'v twl-st-wl nres*› **where**
‹*skip-and-resolve-loop-wl* S_0 =
  do {
    ASSERT(*get-conflict-wl* S_0 ≠ *None*);
    (-, *S*) ←
      $WHILE_T{}^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl\text{-}inv\ S_0\ brk\ S}$
      (λ(*brk*, *S*). ¬*brk* ∧ ¬*is-decided* (*hd* (*get-trail-wl S*)))
      (λ(-, *S*).
        do {
          let D' = *the* (*get-conflict-wl S*);
          let (*L*, *C*) = *lit-and-ann-of-propagated* (*hd* (*get-trail-wl S*));
          if −*L* ∉# *D'* *then*
            do {*RETURN* (*False*, *tl-state-wl S*)}
          else
            if get-maximum-level (*get-trail-wl S*) (*remove1-mset* (−*L*) *D'*) = *count-decided* (*get-trail-wl*
S)
            then
              do {*RETURN* (*update-confl-tl-wl C L S*)}
            else
              do {*RETURN* (*True*, *S*)}
        }
)
      (*False*, S_0);
    RETURN S
  }
›

lemma *tl-state-wl-tl-state-l*:
‹(*S*, *S'*) ∈ *state-wl-l None* ⟹ (*tl-state-wl S*, *tl-state-l S'*) ∈ *state-wl-l None*›
by (*cases S*) (*auto simp*: *state-wl-l-def tl-state-wl-def tl-state-l-def*)

lemma *skip-and-resolve-loop-wl-spec*:
‹(*skip-and-resolve-loop-wl*, *skip-and-resolve-loop-l*)
  ∈ {(*T'*::*'v twl-st-wl*, *T*).
      (*T'*, *T*) ∈ *state-wl-l None* ∧
       correct-watching T' ∧
       0 < *count-decided* (*get-trail-wl T'*)} →
    ⟨{(*T'*, *T*).
      (*T'*, *T*) ∈ *state-wl-l None* ∧
       correct-watching T'}⟩*nres-rel*›
(**is** ‹*?s* ∈ *?A* → ⟨*?B*⟩*nres-rel*›)
proof −
have *get-conflict-wl*: ‹((*False*, *S'*), *False*, *S*)

∈ *Id* $\times_r$ {(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'*}›
(**is** ‹- ∈ *?B*›)
if ‹(*S'*, *S*) ∈ *state-wl-l None*› **and** ‹*correct-watching S'*›
for *S* :: ‹*'v twl-st-l*› **and** *S'* :: ‹*'v twl-st-wl*›
using *that* **by** (*cases S'*) (*auto simp*: *state-wl-l-def*)
have [*simp*]: ‹*correct-watching* (*tl-state-wl S*) = *correct-watching S*› **for** *S*
by (*cases S*) (*auto simp*: *correct-watching.simps tl-state-wl-def clause-to-update-def*)
have [*simp*]: ‹*correct-watching* (*tl aa*, *ca*, *da*, *ea*, *fa*, *ha*, *h*) ⟷
correct-watching (*aa*, *ca*, *None*, *ea*, *fa*, *ha*, *h*)›
for *aa ba ca L da ea fa ha h*
by (*auto simp*: *correct-watching.simps tl-state-wl-def clause-to-update-def*)
have [*simp*]: ‹*NO-MATCH None da* ⟹ *correct-watching* (*aa*, *ca*, *da*, *ea*, *fa*, *ha*, *h*) ⟷
correct-watching (*aa*, *ca*, *None*, *ea*, *fa*, *ha*, *h*)›
for *aa ba ca L da ea fa ha h*
by (*auto simp*: *correct-watching.simps tl-state-wl-def clause-to-update-def*)
have *update-confl-tl-wl*: ‹
(*brkT*, *brkT'*) ∈ *bool-rel* $\times_f$ {(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'*} ⟹
case brkT' of (*brk*, *S*) ⇒ *skip-and-resolve-loop-inv-l S' brk S* ⟹
brkT' = (*brk'*, *T'*) ⟹
brkT = (*brk*, *T*) ⟹
lit-and-ann-of-propagated (*hd* (*get-trail-l T'*)) = (*L'*, *C'*) ⟹
lit-and-ann-of-propagated (*hd* (*get-trail-wl T*)) = (*L*, *C*) ⟹
(*update-confl-tl-wl C L T*, *update-confl-tl-l C' L' T'*) ∈ *bool-rel* $\times_f$ {(*T'*, *T*).
(*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'*}›
for *T' brkT brk brkT' brk' T C C' L L' S'*
unfolding *update-confl-tl-wl-def update-confl-tl-l-def resolve-cls-wl'-def resolve-cls-l'-def*
by (*cases T*; *cases T'*)
(*auto simp*: *Let-def state-wl-l-def*)
have *inv*: ‹*skip-and-resolve-loop-wl-inv S' b' T'*›
if
‹(*S'*, *S*) ∈ *?A*› **and**
‹*get-conflict-wl S'* ≠ *None*› **and**
bt-inv: ‹*case bT of* (*x*, *xa*) ⇒ *skip-and-resolve-loop-inv-l S x xa*› **and**
‹(*b'T'*, *bT*) ∈ *?B*› **and**
b'T': ‹*b'T'* = (*b'*, *T'*)›
for *S' S b'T' bT b' T'*
proof −
obtain *b T* **where** *bT*: ‹*bT* = (*b*, *T*)› **by** (*cases bT*)
show *?thesis*
unfolding *skip-and-resolve-loop-wl-inv-def*
apply (*rule exI*[*of - T*])
apply (*rule exI*[*of - S*])
using *that* **by** (*auto simp*: *bT b'T'*)
qed

show *H*: ‹*?s* ∈ *?A* → ⟨{(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'*}⟩*nres-rel*›
unfolding *skip-and-resolve-loop-wl-def skip-and-resolve-loop-l-def*
apply (*refine-rcg get-conflict-wl*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule inv*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto intro*!: *tl-state-wl-tl-state-l*)
subgoal for *S' S b'T' bT b' T'* **by** (*cases T'*) (*auto simp*: *correct-watching.simps*)

subgoal by *auto*
subgoal by (*rule update-confl-tl-wl*) *assumption+*
subgoal by *auto*
subgoal by (*auto simp*: *correct-watching.simps clause-to-update-def*)
done
qed

Backtrack

definition *find-decomp-wl* :: ‹$'v$ *literal* ⇒ $'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl nres*› **where**
‹*find-decomp-wl* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
SPEC(λ*S*. ∃ *K M2 M1*. *S* = (*M1*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ∧ (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
get-level M K = *get-maximum-level M* (*the D* − {#−*L*#}) + *1*))›

definition *find-lit-of-max-level-wl* :: ‹$'v$ *twl-st-wl* ⇒ $'v$ *literal* ⇒ $'v$ *literal nres*› **where**
‹*find-lit-of-max-level-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) *L*.
SPEC(λ*L'*. *L'* ∈# *remove1-mset* (−*L*) (*the D*) ∧ *get-level M L'* = *get-maximum-level M* (*the D* − {#−*L*#}))))›

fun *extract-shorter-conflict-wl* :: ‹$'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl nres*› **where**
‹*extract-shorter-conflict-wl* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = *SPEC*(λ*S*.
∃ *D'*. *D'* ⊆# *the D* ∧ *S* = (*M*, *N*, *Some D'*, *NE*, *UE*, *Q*, *W*) ∧
clause '# *twl-clause-of* '# *ran-mf N* + *NE* + *UE* ⊨*pm D'* ∧ −(*lit-of* (*hd M*)) ∈# *D'*)›

declare *extract-shorter-conflict-wl.simps*[*simp del*]
lemmas *extract-shorter-conflict-wl-def* = *extract-shorter-conflict-wl.simps*

definition *backtrack-wl-inv* **where**
‹*backtrack-wl-inv S* ⟷ (∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ *backtrack-l-inv S'* ∧ *correct-watching S*)
›

Rougly: we get a fresh index that has not yet been used.

definition *get-fresh-index-wl* :: ‹$'v$ *clauses-l* ⇒ - ⇒ - ⇒ *nat nres*› **where**
‹*get-fresh-index-wl N NUE W* = *SPEC*(λ*i*. *i* > *0* ∧ *i* ∉# *dom-m N* ∧
(∀ *L* ∈# *all-lits-of-mm* (*mset* '# *ran-mf N* + *NUE*) . *i* ∉ *fst* ' *set* (*W L*)))›

definition *propagate-bt-wl* :: ‹$'v$ *literal* ⇒ $'v$ *literal* ⇒ $'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl nres*› **where**
‹*propagate-bt-wl* = (λ*L L'* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
D'' ← *list-of-mset* (*the D*);
i ← *get-fresh-index-wl N* (*NE* + *UE*) *W*;
let b = (*length* ([−*L*, *L'*] @ (*remove1* (−*L*) (*remove1 L' D''*))) = *2*);
RETURN (*Propagated* (−*L*) *i* # *M*,
fmupd i ([−*L*, *L'*] @ (*remove1* (−*L*) (*remove1 L' D''*)), *False*) *N*,
None, *NE*, *UE*, {#*L*#}, *W*(−*L*:= *W* (−*L*) @ [(*i*, *L'*, *b*)], *L'*:= *W L'* @ [(*i*, −*L*, *b*)]))
})›

definition *propagate-unit-bt-wl* :: ‹$'v$ *literal* ⇒ $'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl*› **where**
‹*propagate-unit-bt-wl* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*).
(*Propagated* (−*L*) *0* # *M*, *N*, *None*, *NE*, *add-mset* (*the D*) *UE*, {#*L*#}, *W*))›

definition *backtrack-wl* :: ‹$'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl nres*› **where**
‹*backtrack-wl S* =
do {

ASSERT(*backtrack-wl-inv S*);
let L = *lit-of* (*hd* (*get-trail-wl S*));
S ← *extract-shorter-conflict-wl S*;
S ← *find-decomp-wl L S*;

if size (*the* (*get-conflict-wl S*)) > *1*
then do {
  L' ← *find-lit-of-max-level-wl S L*;
  propagate-bt-wl L L' S
}
else do {
  RETURN (*propagate-unit-bt-wl L S*)
}
}⟩

lemma *correct-watching-learn*:
assumes
L1: ⟨*atm-of L1* ∈ *atms-of-mm* (*mset* '# *ran-mf N* + *NE*)⟩ **and**
L2: ⟨*atm-of L2* ∈ *atms-of-mm* (*mset* '# *ran-mf N* + *NE*)⟩ **and**
UW: ⟨*atms-of* (*mset UW*) ⊆ *atms-of-mm* (*mset* '# *ran-mf N* + *NE*)⟩ **and**
i-dom: ⟨*i* ∉# *dom-m N*⟩ **and**
fresh: ⟨⋀*L*. *L*∈#*all-lits-of-mm* (*mset* '# *ran-mf N* + (*NE* + *UE*)) ⟹ *i* ∉ *fst* ' *set* (*W L*)⟩ **and**
[*iff*]: ⟨*L1* ≠ *L2*⟩ **and**
b: ⟨*b* ⟷ *length* (*L1* # *L2* # *UW*) = *2*⟩
shows
⟨*correct-watching* (*K* # *M*, *fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*,
D, *NE*, *UE*, *Q*, *W* (*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) ⟷
correct-watching (*M*, *N*, *D*, *NE*, *UE*, *Q'*, *W*)⟩
(**is** ⟨*?l* ⟷ *?c*⟩ **is** ⟨*correct-watching* (-, *?N*, -) = -⟩)
proof −
have [*iff*]: ⟨*L2* ≠ *L1*⟩
using ⟨*L1* ≠ *L2*⟩ **by** (*subst eq-commute*)
have [*simp*]: ⟨*clause-to-update L1* (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*, *D*, *NE*, *UE*, {#}, {#}) =
add-mset i (*clause-to-update L1* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))⟩ **for** *L2 UW*
using *i-dom*
by (*auto simp*: *clause-to-update-def intro*: *filter-mset-cong*)
have [*simp*]: ⟨*clause-to-update L2* (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*, *D*, *NE*, *UE*, {#}, {#}) =
add-mset i (*clause-to-update L2* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))⟩ **for** *L1 UW*
using *i-dom*
by (*auto simp*: *clause-to-update-def intro*: *filter-mset-cong*)
have [*simp*]: ⟨*x* ≠ *L1* ⟹ *x* ≠ *L2* ⟹
clause-to-update x (*M*, *fmupd i* (*L1* # *L2* # *UW*, *b'*) *N*, *D*, *NE*, *UE*, {#}, {#}) =
clause-to-update x (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})⟩ **for** *x UW*
using *i-dom*
by (*auto simp*: *clause-to-update-def intro*: *filter-mset-cong*)
have [*simp*]: ⟨*L1* ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*))⟩
⟨*L2* ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*))⟩
using *i-dom L1 L2 UW*
by (*fastforce simp*: *ran-m-mapsto-upd-notin*
all-lits-of-mm-add-mset all-lits-of-m-add-mset in-all-lits-of-m-ain-atms-of-iff
in-all-lits-of-mm-ain-atms-of-iff)+
have *H'*:
⟨{#*ia* ∈# *fst* '# *mset* (*W x*). *ia* = *i* ∨ *ia* ∈# *dom-m N*#} = {#*ia* ∈# *fst* '# *mset* (*W x*). *ia* ∈# *dom-m N*#}⟩
if ⟨*x* ∈# *all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*))⟩ **for** *x*
using *i-dom fresh*[*of x*] *that*

by (*auto simp*: *clause-to-update-def intro*!: *filter-mset-cong*)
have [*simp*]:
‹*clause-to-update L1* (*K* # *M*, *N*, *D*, *NE*, *UE*, {#}, {#}) = *clause-to-update L1* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})›
for *L1 N D NE UE M K*
by (*auto simp*: *clause-to-update-def*)

have [*simp*]: ‹*set-mset* (*all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m ?N*#} + (*NE* + *UE*))) =
set-mset (*all-lits-of-mm* ({#*mset* (*fst x*). *x* ∈# *ran-m N*#} + (*NE* + *UE*)))›
using *i-dom L1 L2 UW*
by (*fastforce simp*: *ran-m-mapsto-upd-notin*
all-lits-of-mm-add-mset all-lits-of-m-add-mset in-all-lits-of-m-ain-atms-of-iff
in-all-lits-of-mm-ain-atms-of-iff)

show *?thesis*
proof (*rule iffI*)
assume *corr*: *?l*
have
H: ‹$\bigwedge$*L ia K′ b″*. (*L*∈#*all-lits-of-mm*
(*mset* ‘# *ran-mf* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*) + (*NE* + *UE*)) ⟹
distinct-watched ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *L*) ∧
((*ia*, *K′*, *b″*)∈#*mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *L*) ⟶
ia ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*) ⟶
K′ ∈ *set* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N* ∝ *ia*) ∧ *K′* ≠ *L* ∧
correctly-marked-as-binary (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*) (*ia*, *K′*, *b″*)) ∧
((*ia*, *K′*, *b″*)∈#*mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *L*) ⟶
b″ ⟶ *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*)) ∧
{#*ia* ∈# *fst* ‘#
mset ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *L*).
ia ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*)#} =
clause-to-update L
(*K* # *M*, *fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*, *D*, *NE*, *UE*, {#}, {#}))›
using *corr* **unfolding** *correct-watching.simps*
by *fast+*

have ‹*x* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*)) ⟹
distinct-watched (*W x*) ∧
(*xa* ∈# *mset* (*W x*) ⟶ (((*case xa of* (*i*, *K*, *b″*) ⇒ *i* ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *x* ∧
correctly-marked-as-binary N (*i*, *K*, *b″*)) ∧
(*case xa of* (*i*, *K*, *b″*) ⇒ *b″* ⟶ *i* ∈# *dom-m N*)))) ∧
{#*i* ∈# *fst* ‘# *mset* (*W x*). *i* ∈# *dom-m N*#} = *clause-to-update x* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#})›
for *x xa*
supply *correctly-marked-as-binary.simps*[*simp*]
using *H*[*of x* ‹*fst xa*› ‹*fst* (*snd xa*)› ‹*snd* (*snd xa*)›] *fresh*[*of x*] *i-dom*
apply (*cases* ‹*x* = *L1*›; *cases* ‹*x* = *L2*›)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H′*)
subgoal
by (*cases xa*) (*force simp add*: *H′ split*: *if-splits*)
subgoal
by (*cases xa*)
(*force simp add*: *H′ split*: *if-splits*)
subgoal

by (*cases xa*)
(*force simp add*: *H′ split*: *if-splits*)
done
then show *?c*
unfolding *correct-watching.simps Ball-def*
by (*auto 5 5 simp add*: *all-lits-of-mm-add-mset all-lits-of-m-add-mset all-conj-distrib all-lits-of-mm-union dest*: *multi-member-split*)
next
assume *corr*: *?c*
have
H: ‹$\bigwedge L$ *ia K′ b″*. ($L \in\#$*all-lits-of-mm*
(*mset* ‘# *ran-mf N* + (*NE* + *UE*)) $\Longrightarrow$
distinct-watched (*W L*) ∧
((*ia*, *K′*, *b″*)∈#*mset* (*W L*) ⟶
ia ∈# *dom-m N* ⟶
K′ ∈ *set* (*N* ∝ *ia*) ∧ *K′* ≠ *L* ∧ *correctly-marked-as-binary N* (*ia*, *K′*, *b″*)) ∧
((*ia*, *K′*, *b″*)∈#*mset* (*W L*) ⟶ *b″* ⟶ *ia* ∈# *dom-m N*) ∧
{#*ia* ∈# *fst* ‘# *mset* (*W L*). *ia* ∈# *dom-m N*#} = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))›
using *corr* **unfolding** *correct-watching.simps*
by *blast+*
have ‹*x* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*) + (*NE* + *UE*)) ⟶
distinct-watched ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*) ∧
(*xa* ∈# *mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*) ⟶
(*case xa of* (*ia*, *K*, *b″*) ⇒ *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*) ⟶
K ∈ *set* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N* ∝ *ia*) ∧ *K* ≠ *x* ∧
correctly-marked-as-binary (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*) (*ia*, *K*, *b″*))) ∧
(*xa* ∈# *mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*) ⟶
(*case xa of* (*ia*, *K*, *b″*) ⇒ *b″* ⟶ *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*))) ∧
{#*ia* ∈# *fst* ‘# *mset* ((*W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)], *L2* := *W L2* @ [(*i*, *L1*, *b*)])) *x*). *ia* ∈# *dom-m* (*fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*)#} =
clause-to-update x (*K* # *M*, *fmupd i* (*L1* # *L2* # *UW*, *b′*) *N*, *D*, *NE*, *UE*, {#}, {#})›
for *x* :: ‹′*a literal*› **and** *xa*
supply *correctly-marked-as-binary.simps*[*simp*]
using *H*[*of x* ‹*fst xa*› ‹*fst* (*snd xa*)› ‹*snd* (*snd xa*)›] *fresh*[*of x*] *i-dom b*
apply (*cases* ‹*x* = *L1*›; *cases* ‹*x* = *L2*›)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H′*)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H′*)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H′*)
subgoal
by (*cases xa*)
(*auto dest*!: *multi-member-split simp*: *H′*)
done
then show *?l*
unfolding *correct-watching.simps Ball-def*
by *auto*
qed
qed

fun *equality-except-conflict-wl* :: ‹$'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl* ⇒ *bool*› **where**
‹*equality-except-conflict-wl* (M, N, D, NE, UE, WS, Q) $(M', N', D', NE', UE', WS', Q')$ ⟷
$M = M' \wedge N = N' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$›

fun *equality-except-trail-wl* :: ‹$'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl* ⇒ *bool*› **where**
‹*equality-except-trail-wl* (M, N, D, NE, UE, WS, Q) $(M', N', D', NE', UE', WS', Q')$ ⟷
$N = N' \wedge D = D' \wedge NE = NE' \wedge UE = UE' \wedge WS = WS' \wedge Q = Q'$›

lemma *equality-except-conflict-wl-get-clauses-wl*:
‹*equality-except-conflict-wl* S Y ⟹ *get-clauses-wl* S = *get-clauses-wl* Y›
by (*cases* S; *cases* Y) (*auto simp*:)
lemma *equality-except-trail-wl-get-clauses-wl*:
‹*equality-except-trail-wl* S Y⟹ *get-clauses-wl* S = *get-clauses-wl* Y›
by (*cases* S; *cases* Y) (*auto simp*:)

lemma *backtrack-wl-spec*:
‹(*backtrack-wl*, *backtrack-l*)
∈ {(T'::$'v$ *twl-st-wl*, T).
(T', T) ∈ *state-wl-l None* ∧
correct-watching T' ∧
get-conflict-wl T' ≠ *None* ∧
get-conflict-wl T' ≠ *Some* {#}} →
⟨{(T', T).
(T', T) ∈ *state-wl-l None* ∧
correct-watching T'}⟩*nres-rel*›
(**is** ‹*?bt* ∈ *?A* → ⟨*?B*⟩*nres-rel*›)
proof −
have *extract-shorter-conflict-wl*: ‹*extract-shorter-conflict-wl* S'
≤ ⇓ {(U'::$'v$ *twl-st-wl*, U).
(U', U) ∈ *state-wl-l None* ∧ *equality-except-conflict-wl* U' S' ∧
the (*get-conflict-wl* U') ⊆# *the* (*get-conflict-wl* S') ∧
get-conflict-wl U' ≠ *None*} (*extract-shorter-conflict-l* S)›
(**is** ‹- ≤ ⇓ *?extract* -›)
if ‹(S', S) ∈ *?A*›
for S' S
apply (*cases* S'; *cases* S)
apply *clarify*
unfolding *extract-shorter-conflict-wl-def extract-shorter-conflict-l-def*
apply (*rule RES-refine*)
using *that*
by (*auto simp*: *extract-shorter-conflict-wl-def extract-shorter-conflict-l-def*
mset-take-mset-drop-mset state-wl-l-def)

have *find-decomp-wl*: ‹*find-decomp-wl* L T'
≤ ⇓ {(U'::$'v$ *twl-st-wl*, U).
(U', U) ∈ *state-wl-l None* ∧ *equality-except-trail-wl* U' T' ∧
(∃ M. *get-trail-wl* T' = M @ *get-trail-wl* U') } (*find-decomp* L' T)›
(**is** ‹- ≤ ⇓ *?find* -›)
if ‹(S', S) ∈ *?A*› ‹$L = L'$› ‹(T', T) ∈ *?extract* S'›
for S' S T T' L L'
using *that*
apply (*cases* T; *cases* T')
apply *clarify*
unfolding *find-decomp-wl-def find-decomp-def prod.case*
apply (*rule RES-refine*)
apply (*auto 5 5 simp add*: *state-wl-l-def find-decomp-wl-def find-decomp-def*)

done

have *find-lit-of-max-level-wl*: ‹*find-lit-of-max-level-wl* T' *LLK*$'$
$\leq \Downarrow \{(L', L).\ L = L' \wedge L' \in\#$ *the* (*get-conflict-wl* T') $\wedge\ L' \in\#$ *the* (*get-conflict-wl* T') $-$ $\{\#-LLK'\#\}\}$
(*find-lit-of-max-level* T L)›
(**is** ‹- $\leq \Downarrow$ *?find-lit* -›)
if ‹$L = LLK'$› ‹$(T', T) \in$ *?find* S'›
for S' S T T' L *LLK*$'$
using *that*
apply (*cases* T; *cases* T'; *cases* S')
apply *clarify*
unfolding *find-lit-of-max-level-wl-def find-lit-of-max-level-def prod.case*
apply (*rule RES-refine*)
apply (*auto simp add*: *find-lit-of-max-level-wl-def find-lit-of-max-level-def state-wl-l-def*
dest: *in-diffD*)
done
have *empty*: ‹*literals-to-update-wl* $S' = \{\#\}$› **if** *bt*: ‹*backtrack-wl-inv* S'› **for** S'
using *bt* **apply** $-$
unfolding *backtrack-wl-inv-def backtrack-l-inv-def*
apply *normalize-goal*+
apply (*auto simp*: *twl-struct-invs-def*)
done
have *propagate-bt-wl*: ‹*propagate-bt-wl* (*lit-of* (*hd* (*get-trail-wl* S'))) L' U'
$\leq \Downarrow \{(T', T).\ (T', T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $T'\}$
(*propagate-bt-l* (*lit-of* (*hd* (*get-trail-l* S))) L U)›
(**is** ‹- $\leq \Downarrow$ *?propa* -›)
if *SS*$'$: ‹$(S', S) \in$ *?A*› **and**
UU$'$: ‹$(U', U) \in$ *?find* T'› **and**
LL$'$: ‹$(L', L) \in$ *?find-lit* U' (*lit-of* (*hd* (*get-trail-wl* S')))› **and**
TT$'$: ‹$(T', T) \in$ *?extract* S'› **and**
bt: ‹*backtrack-wl-inv* S'›
for S' S T T' L L' U U'
proof $-$
note *empty* = *empty*[*OF bt*]
define K' **where** ‹K' = *lit-of* (*hd* (*get-trail-l* S))›
obtain *MS NS DS NES UES W* **where**
S': ‹S' = (*MS*, *NS*, *Some DS*, *NES*, *UES*, $\{\#\}$, W)›
using *SS*$'$ *empty* **by** (*cases* S'; *cases* ‹*get-conflict-wl* S'›) *auto*
then obtain *DT* **where**
T': ‹T' = (*MS*, *NS*, *Some DT*, *NES*, *UES*, $\{\#\}$, W)› **and**
‹*DT* $\subseteq\#$ *DS*›
using *TT*$'$ **by** (*cases* T'; *cases* ‹*get-conflict-wl* T'›) *auto*
then obtain *MU MU*$'$ **where**
U': ‹U' = (*MU*, *NS*, *Some DT*, *NES*, *UES*, $\{\#\}$, W)› **and**
MU: ‹*MS* = *MU*$'$ @ *MU*› **and**
$U'U$: ‹$(U', U) \in$ *state-wl-l None*›
using *UU*$'$ **by** (*cases* U') *auto*
then have U: ‹U = (*MU*, *NS*, *Some DT*, *NES*, *UES*, $\{\#\}$, $\{\#\}$)›
by (*cases* U) (*auto simp*: *state-wl-l-def*)
have *MS*: ‹*MS* $\neq$ []›
using *bt* **unfolding** *backtrack-wl-inv-def backtrack-l-inv-def* S' **by** (*auto simp*: *state-wl-l-def*)
have ‹*correct-watching* S'›
using *SS*$'$ **by** *fast*
then have *corr*: ‹*correct-watching* (*MU*, *NS*, *None*, *NES*, *UES*, $\{\#K'\#\}$, W)›
unfolding S' *correct-watching.simps clause-to-update-def get-clauses-l.simps*

by *simp*
have *K-hd*[*simp*]: ‹*lit-of* (*hd MS*) = *K'*›
using *SS'* **unfolding** *K'-def* **by** (*auto simp*: *S'*)
have [*simp*]: ‹*L* = *L'*›
using *LL'* **by** *auto*
have *trail-no-alien*:
‹*atm-of* ‘ *lits-of-l* (*get-trail-wl S'*)
⊆ *atms-of-ms*
((λ*x*. *mset* (*fst x*)) ‘
{*a*. *a* ∈# *ran-m* (*get-clauses-wl S'*) ∧ *snd a*}) ∪
atms-of-mm (*get-unit-init-clss-wl S'*)› **and**
no-alien: ‹*atms-of DS* ⊆ *atms-of-ms*
((λ*x*. *mset* (*fst x*)) ‘
{*a*. *a* ∈# *ran-m* (*get-clauses-wl S'*) ∧ *snd a*}) ∪
atms-of-mm (*get-unit-init-clss-wl S'*)› **and**
dist: ‹*distinct-mset DS*›
using *SS' bt* **unfolding** *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
backtrack-wl-inv-def backtrack-l-inv-def cdcl$_W$*-restart-mset.no-strange-atm-def*
cdcl$_W$*-restart-mset.distinct-cdcl*$_W$*-state-def*
apply −
apply *normalize-goal*+
apply (*simp add*: *twl-st twl-st-l twl-st-wl*)
apply *normalize-goal*+
apply (*simp add*: *twl-st twl-st-l twl-st-wl S'*)
apply *normalize-goal*+
apply (*simp add*: *twl-st twl-st-l twl-st-wl S'*)
done
moreover have ‹*L'* ∈# *DS*›
using *LL' TT'* **by** (*auto simp*: *T' S' U' mset-take-mset-drop-mset*)
ultimately have
atm-L': ‹*atm-of L'* ∈ *atms-of-mm* (*mset* ‘# *init-clss-lf NS* + *NES*)› **and**
atm-confl: ‹∀ *L*∈#*DS*. *atm-of L* ∈ *atms-of-mm* (*mset* ‘# *init-clss-lf NS* + *NES*)›
by (*auto simp*: *cdcl*$_W$*-restart-mset.no-strange-atm-def cdcl*$_W$*-restart-mset-state S'*
mset-take-mset-drop-mset dest!: *atm-of-lit-in-atms-of*)
have *atm-K'*: ‹*atm-of K'* ∈ *atms-of-mm* (*mset* ‘# *init-clss-lf NS* + *NES*)›
using *trail-no-alien K-hd MS*
by (*cases MS*) (*auto simp*: *S'*
mset-take-mset-drop-mset simp del: *K-hd dest*!: *atm-of-lit-in-atms-of*)
have *dist*: ‹*distinct-mset DT*›
using ‹*DT* ⊆# *DS*› *dist* **by** (*rule distinct-mset-mono*)
have *fresh*: ‹*get-fresh-index-wl N* (*NUE*) *W* ≤
⇓ {(*i*, *i'*). *i* = *i'* ∧ *i* ∉# *dom-m N* ∧ (∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + *NUE*). *i* ∉ *fst* ‘ *set* (*W L*))} (*get-fresh-index N'*)›
if ‹*N* = *N'*› **for** *N N' NUE W*
unfolding *that get-fresh-index-def get-fresh-index-wl-def*
by (*auto intro*: *RES-refine*)
have [*refine0*]: ‹*SPEC* (λ*D'*. *the D* = *mset D'*) ≤ ⇓ {(*D'*, *E'*). *D'* = *E'* ∧ *the D* = *mset D'*}
(*SPEC* (λ*D'*. *the E* = *mset D'*))›
if ‹*D* = *E*› **for** *D E*
using *that* **by** (*auto intro*!: *RES-refine*)
show *?thesis*
unfolding *propagate-bt-wl-def propagate-bt-l-def S' T' U' U st-l-of-wl.simps get-trail-wl.simps*
list-of-mset-def K'-def[*symmetric*] *Let-def*
apply (*refine-vcg fresh*; *remove-dummy-vars*)
apply (*subst in-pair-collect-simp*)
apply (*intro conjI*)

subgoal using *SS'* **by** (*auto simp*: *corr state-wl-l-def S'*)
subgoal
apply *simp*
apply (*subst correct-watching-learn*)
subgoal using *atm-K'* **unfolding** *all-clss-lf-ran-m*[*symmetric*] *image-mset-union* **by** *auto*
subgoal using *atm-L'* **unfolding** *all-clss-lf-ran-m*[*symmetric*] *image-mset-union* **by** *auto*
subgoal using *atm-confl TT'* **unfolding** *all-clss-lf-ran-m*[*symmetric*] *image-mset-union*
by (*fastforce simp*: *S' T' dest*!: *in-atms-of-minusD*)
subgoal by *auto*
subgoal by *auto*
subgoal using *dist LL'* **by** (*auto simp*: *U' S' distinct-mset-remove1-All*)
subgoal by *auto*
apply (*rule corr*)
done
done
qed

have *propagate-unit-bt-wl*: ‹(*propagate-unit-bt-wl* (*lit-of* (*hd* (*get-trail-wl S'*))) *U'*,
propagate-unit-bt-l (*lit-of* (*hd* (*get-trail-l S*))) *U*)
∈ {(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'*} ›
(**is** ‹(-, -) ∈ *?propagate-unit-bt-wl* -›)
if
SS': ‹(*S'*, *S*) ∈ *?A*› **and**
TT': ‹(*T'*, *T*) ∈ *?extract S'*› **and**
UU': ‹(*U'*, *U*) ∈ *?find T'*› **and**
bt: ‹*backtrack-wl-inv S'*›
for *S' S T T' L L' U U' K'*
proof −
obtain *MS NS DS NES UES W* **where**
S': ‹*S'* = (*MS*, *NS*, *Some DS*, *NES*, *UES*, {#}, *W*)›
using *SS' UU' empty*[*OF bt*] **by** (*cases S'*; *cases* ‹*get-conflict-wl S'*›) *auto*
then obtain *DT* **where**
T': ‹*T'* = (*MS*, *NS*, *Some DT*, *NES*, *UES*, {#}, *W*)› **and**
DT-DS: ‹*DT* ⊆# *DS*›
using *TT'* **by** (*cases T'*; *cases* ‹*get-conflict-wl T'*›) *auto*
have *T*: ‹*T* = (*MS*, *NS*, *Some DT*, *NES*, *UES*, {#}, {#})›
using *TT'* **by** (*auto simp*: *S' T' state-wl-l-def*)
obtain *MU MU'* **where**
U': ‹*U'* = (*MU*, *NS*, *Some DT*, *NES*, *UES*, {#}, *W*)› **and**
MU: ‹*MS* = *MU'* @ *MU*› **and**
U: ‹(*U'*, *U*) ∈ *state-wl-l None*›
using *UU' T'* **by** (*cases U'*) *auto*
have *U*: ‹*U* = (*MU*, *NS*, *Some DT*, *NES*, *UES*, {#}, {#})›
using *UU'* **by** (*auto simp*: *U' state-wl-l-def*)
obtain *S1 S2* **where**
S1: ‹(*S'*, *S1*) ∈ *state-wl-l None*› **and**
S2: ‹(*S1*, *S2*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs S2*›
using *bt* **unfolding** *backtrack-wl-inv-def backtrack-l-inv-def*
by *blast*
have ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of S2*)›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *fast*
then have *K*: ‹*set-mset* (*all-lits-of-mm* (*mset* ‘# *ran-mf NS* + *NES* + *add-mset* (*the* (*Some DT*)) *UES*)) =
set-mset (*all-lits-of-mm* (*mset* ‘# *ran-mf NS* + (*NES* + *UES*)))›

apply (*subst all-clss-lf-ran-m*[*symmetric*])+
apply (*subst image-mset-union*)+
using *S1 S2 atms-of-subset-mset-mono*[*OF DT-DS*]
by (*fastforce simp*: *all-lits-of-mm-union all-lits-of-mm-add-mset state-wl-l-def twl-st-l-def S′ cdcl*$_W$*-restart-mset.no-strange-atm-def cdcl*$_W$*-restart-mset-state mset-take-mset-drop-mset′ in-all-lits-of-mm-ain-atms-of-iff in-all-lits-of-m-ain-atms-of-iff*)
then have *K′*: ‹*set-mset* (*all-lits-of-mm* (*mset ‘# ran-mf NS* + (*NES* + *add-mset* (*the* (*Some DT*)) *UES*))) =
set-mset (*all-lits-of-mm* (*mset ‘# ran-mf NS* + (*NES* + *UES*)))›
by (*auto simp*: *ac-simps*)
have ‹*correct-watching S′*›
using *SS′* **by** *fast*
then have *corr*: ‹*correct-watching* (*Propagated* (− *lit-of* (*hd MS*)) *0* # *MU*, *NS*, *None*, *NES*, *add-mset* (*the* (*Some DT*)) *UES*, *unmark* (*hd MS*), *W*)›
unfolding *S′ correct-watching.simps clause-to-update-def get-clauses-l.simps K K′* **.**

show *?thesis*
unfolding *propagate-unit-bt-wl-def propagate-unit-bt-l-def S′ T′ U U′ st-l-of-wl.simps get-trail-wl.simps list-of-mset-def*
apply *clarify*
apply (*refine-rcg*)
subgoal using *SS′* **by** (*auto simp*: *S′ state-wl-l-def*)
subgoal by (*rule corr*)
done
qed
show *?thesis*
unfolding *st-l-of-wl.simps get-trail-wl.simps list-of-mset-def backtrack-wl-def backtrack-l-def*
apply (*refine-vcg find-decomp-wl find-lit-of-max-level-wl extract-shorter-conflict-wl propagate-bt-wl propagate-unit-bt-wl*;
remove-dummy-vars)
subgoal using *backtrack-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

Backtrack, Skip, Resolve or Decide

definition *cdcl-twl-o-prog-wl-pre* **where**
‹*cdcl-twl-o-prog-wl-pre S* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *state-wl-l None* ∧
correct-watching S ∧
cdcl-twl-o-prog-l-pre S′)›

definition *cdcl-twl-o-prog-wl* :: ‹*′v twl-st-wl* ⇒ (*bool* × *′v twl-st-wl*) *nres*› **where**
‹*cdcl-twl-o-prog-wl S* =
do {
ASSERT(*cdcl-twl-o-prog-wl-pre S*);
do {
if get-conflict-wl S = *None*
then decide-wl-or-skip S
else do {

if count-decided (*get-trail-wl S*) > *0*
then do {
T ← *skip-and-resolve-loop-wl S*;
ASSERT(*get-conflict-wl T* ≠ *None* ∧ *get-conflict-wl T* ≠ *Some* {#});
U ← *backtrack-wl T*;
RETURN (*False*, *U*)
}
else do {*RETURN* (*True*, *S*)}
}
}
}
⟩

lemma *cdcl-twl-o-prog-wl-spec*:
⟨(*cdcl-twl-o-prog-wl*, *cdcl-twl-o-prog-l*) ∈ {(*S*::′*v twl-st-wl*, *S*′::′*v twl-st-l*).
(*S*, *S*′) ∈ *state-wl-l None* ∧
correct-watching S} $\rightarrow_f$
⟨{((*brk*::*bool*, *T*::′*v twl-st-wl*), *brk*′::*bool*, *T*′::′*v twl-st-l*).
(*T*, *T*′) ∈ *state-wl-l None* ∧
brk = *brk*′ ∧
correct-watching T}⟩*nres-rel*⟩
(**is** ⟨*?o* ∈ *?A* $\rightarrow_f$ ⟨*?B*⟩ *nres-rel*⟩)
proof −
have *find-unassigned-lit-wl*: ⟨*find-unassigned-lit-wl S* ≤ ⇓ *Id* (*find-unassigned-lit-l S*′)⟩
if ⟨(*S*, *S*′) ∈ *state-wl-l None*⟩
for *S* :: ⟨′*v twl-st-wl*⟩ **and** *S*′ :: ⟨′*v twl-st-l*⟩
unfolding *find-unassigned-lit-wl-def find-unassigned-lit-l-def*
using *that*
by (*cases S*; *cases S*′) (*auto simp*: *state-wl-l-def*)
have [*iff*]: ⟨*correct-watching* (*decide-lit-wl L S*) ⟷ *correct-watching S*⟩ **for** *L S*
by (*cases S*; *auto simp*: *decide-lit-wl-def correct-watching.simps clause-to-update-def*)
have [*iff*]: ⟨(*decide-lit-wl L S*, *decide-lit-l L S*′) ∈ *state-wl-l None*⟩
if ⟨(*S*, *S*′) ∈ *state-wl-l None*⟩
for *L S S*′
using *that* **by** (*cases S*; *auto simp*: *decide-lit-wl-def decide-lit-l-def state-wl-l-def*)
have *option-id*: ⟨*x* = *x*′ ⟹ (*x*,*x*′) ∈ ⟨*Id*⟩*option-rel*⟩ **for** *x x*′ **by** *auto*
show *cdcl-o*: ⟨*?o* ∈ *?A* $\rightarrow_f$
⟨{((*brk*::*bool*, *T*::′*v twl-st-wl*), *brk*′::*bool*, *T*′::′*v twl-st-l*).
(*T*, *T*′) ∈ *state-wl-l None* ∧
brk = *brk*′ ∧
correct-watching T}⟩*nres-rel*⟩
unfolding *cdcl-twl-o-prog-wl-def cdcl-twl-o-prog-l-def decide-wl-or-skip-def*
decide-l-or-skip-def fref-param1[*symmetric*]
apply (*refine-vcg skip-and-resolve-loop-wl-spec*[*to*-⇓] *backtrack-wl-spec*[*to*-⇓]
find-unassigned-lit-wl option-id)
subgoal unfolding *cdcl-twl-o-prog-wl-pre-def* **by** *blast*
subgoal by *auto*
subgoal unfolding *decide-wl-or-skip-pre-def* **by** *blast*
subgoal by (*auto simp*:)
subgoal unfolding *decide-wl-or-skip-pre-def* **by** *auto*
subgoal by *auto*
subgoal by (*auto simp*:)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal by (*auto simp*:)
subgoal by (*auto simp*:)
subgoal by *auto*
done
qed

Full Strategy

definition *cdcl-twl-stgy-prog-wl-inv* :: ‹'*v twl-st-wl* ⇒ *bool* × '*v twl-st-wl* ⇒ *bool*› **where**
‹*cdcl-twl-stgy-prog-wl-inv* S_0 ≡ λ(*brk*, *T*).
(∃ *T'* S_0'. (*T*, *T'*) ∈ *state-wl-l None* ∧
(S_0, S_0') ∈ *state-wl-l None* ∧
cdcl-twl-stgy-prog-l-inv S_0' (*brk*, *T'*))›

definition *cdcl-twl-stgy-prog-wl* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*cdcl-twl-stgy-prog-wl* S_0 =
do {
(*brk*, *T*) ← $WHILE_T{}^{cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}inv\ S_0}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*). *do* {
T ← *unit-propagation-outer-loop-wl S*;
cdcl-twl-o-prog-wl T
})
(*False*, S_0);
RETURN T
}›

theorem *cdcl-twl-stgy-prog-wl-spec*:
‹(*cdcl-twl-stgy-prog-wl*, *cdcl-twl-stgy-prog-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧
correct-watching S} →
⟨*state-wl-l None*⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* → ⟨*?B*⟩ *nres-rel*›)
proof −
have *H*: ‹((*False*, *S'*), *False*, *S*) ∈ {((*brk'*, *T'*), (*brk*, *T*)). (*T'*, *T*) ∈ *state-wl-l None* ∧ *brk'* = *brk* ∧
correct-watching T'}›
if ‹(*S'*, *S*) ∈ *state-wl-l None*› **and**
‹*correct-watching S'*›
for *S'* :: ‹'*v twl-st-wl*› **and** *S* :: ‹'*v twl-st-l*›
using *that* **by** *auto*

show *?thesis*
unfolding *cdcl-twl-stgy-prog-wl-def cdcl-twl-stgy-prog-l-def*
apply (*refine-rcg H unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
subgoal for *S' S* **by** (*cases S'*) *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-prog-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal for *S' S brk'T' brkT brk' T'* **by** *auto*
subgoal by *fast*
subgoal by *auto*
done

qed

theorem *cdcl-twl-stgy-prog-wl-spec'*:
‹(*cdcl-twl-stgy-prog-wl*, *cdcl-twl-stgy-prog-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*} →
⟨{(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* → ⟨*?B*⟩ *nres-rel*›)
proof −
have *H*: ‹((*False*, *S'*), *False*, *S*) ∈ {((*brk'*, *T'*), (*brk*, *T*)). (*T'*, *T*) ∈ *state-wl-l None* ∧ *brk'* = *brk* ∧
correct-watching T'}›
if ‹(*S'*, *S*) ∈ *state-wl-l None*› **and**
‹*correct-watching S'*›
for *S'* :: ‹'*v twl-st-wl*› **and** *S* :: ‹'*v twl-st-l*›
using *that* **by** *auto*
thm *unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
show *?thesis*
unfolding *cdcl-twl-stgy-prog-wl-def cdcl-twl-stgy-prog-l-def*
apply (*refine-rcg H unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
subgoal for *S' S* **by** (*cases S'*) *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-prog-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal for *S' S brk'T' brkT brk' T'* **by** *auto*
subgoal by *fast*
subgoal by *auto*
done
qed

definition *cdcl-twl-stgy-prog-wl-pre* **where**
‹*cdcl-twl-stgy-prog-wl-pre S U* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *cdcl-twl-stgy-prog-l-pre T U* ∧ *correct-watching S*)›

lemma *cdcl-twl-stgy-prog-wl-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-wl S* ≤ ⇓ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S'*)›
proof −
obtain *T* **where** *T*: ‹(*S*, *T*) ∈ *state-wl-l None*› ‹*cdcl-twl-stgy-prog-l-pre T S'*› ‹*correct-watching S*›
using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-pre-def* **by** *blast*
show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-wl-spec*[*to-*⇓, *of S T*]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step'*)
apply (*rule cdcl-twl-stgy-prog-l-spec-final*[*of - S'*])
subgoal using *T* **by** *fast*
subgoal unfolding *conc-fun-chain* **by** *auto*
done
done
qed

definition *cdcl-twl-stgy-prog-break-wl* :: ⟨*'v twl-st-wl* ⇒ *'v twl-st-wl nres*⟩ **where**
⟨*cdcl-twl-stgy-prog-break-wl* S_0 =
do {
b ← *SPEC*(λ-. *True*);
(*b*, *brk*, *T*) ← $WHILE_T{}^{\lambda(\text{-},\, S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}inv\ S_0\ S}$
(λ(*b*, *brk*, -). *b* ∧ ¬*brk*)
(λ(-, *brk*, *S*). *do* {
T ← *unit-propagation-outer-loop-wl S*;
T ← *cdcl-twl-o-prog-wl T*;
b ← *SPEC*(λ-. *True*);
RETURN (*b*, *T*)
})
(*b*, *False*, S_0);
if brk then RETURN T
else cdcl-twl-stgy-prog-wl T
}⟩

theorem *cdcl-twl-stgy-prog-break-wl-spec'*:
⟨(*cdcl-twl-stgy-prog-break-wl*, *cdcl-twl-stgy-prog-break-l*) ∈ {(*S*::*'v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\rightarrow_f$
⟨{(*S*::*'v twl-st-wl*, *S'*). (*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*⟩
(**is** ⟨*?o* ∈ *?A* $\rightarrow_f$ ⟨*?B*⟩ *nres-rel*⟩)
proof −
have *H*: ⟨((*b'*, *False*, *S'*), *b*, *False*, *S*) ∈ {((*b'*, *brk'*, *T'*), (*b*, *brk*, *T*)).
(*T'*, *T*) ∈ *state-wl-l None* ∧ *brk'* = *brk* ∧ *b'* = *b* ∧
correct-watching T'}⟩
if ⟨(*S'*, *S*) ∈ *state-wl-l None*⟩ **and**
⟨*correct-watching S'*⟩ **and**
⟨(*b'*, *b*) ∈ *bool-rel*⟩
for *S'* :: ⟨*'v twl-st-wl*⟩ **and** *S* :: ⟨*'v twl-st-l*⟩ **and** *b' b* :: *bool*
using *that* **by** *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-prog-break-wl-def cdcl-twl-stgy-prog-break-l-def fref-param1*[*symmetric*]
apply (*refine-rcg H unit-propagation-outer-loop-wl-spec*[*THEN fref-to-Down*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*]
cdcl-twl-stgy-prog-wl-spec'[*unfolded fref-param1*, *THEN fref-to-Down*])
subgoal for *S' S* **by** (*cases S'*) *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-prog-wl-inv-def* **by** *blast*
subgoal by *auto*
subgoal by *auto*
subgoal for *S' S brk'T' brkT brk' T'* **by** *auto*
subgoal by *fast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *fast*
subgoal by *auto*
done
qed

theorem *cdcl-twl-stgy-prog-break-wl-spec*:
⟨(*cdcl-twl-stgy-prog-break-wl*, *cdcl-twl-stgy-prog-break-l*) ∈ {(*S*::*'v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧

correct-watching S} $\to_f$
⟨*state-wl-l None*⟩*nres-rel*⟩
(**is** ⟨*?o* ∈ *?A* $\to_f$ ⟨*?B*⟩ *nres-rel*⟩)
using *cdcl-twl-stgy-prog-break-wl-spec′*
apply −
apply (*rule mem-set-trans*)
prefer *2* **apply** *assumption*
apply (*match-fun-rel*, *solves simp*)
apply (*match-fun-rel*; *solves auto*)
done

lemma *cdcl-twl-stgy-prog-break-wl-spec-final*:
assumes
⟨*cdcl-twl-stgy-prog-wl-pre S S′*⟩
shows
⟨*cdcl-twl-stgy-prog-break-wl S* ≤ ⇓ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S′*)⟩
proof −
obtain *T* **where** *T*: ⟨(*S*, *T*) ∈ *state-wl-l None*⟩ ⟨*cdcl-twl-stgy-prog-l-pre T S′*⟩ ⟨*correct-watching S*⟩
using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-pre-def* **by** *blast*
show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-break-wl-spec*[*unfolded fref-param1*[*symmetric*], *to-*⇓, *of S T*]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step′*)
apply (*rule cdcl-twl-stgy-prog-break-l-spec-final*[*of - S′*])
subgoal using *T* **by** *fast*
subgoal unfolding *conc-fun-chain* **by** *auto*
done
done
qed

end
theory *Watched-Literals-Watch-List-Restart*
imports *Watched-Literals-List-Restart Watched-Literals-Watch-List*
begin

To ease the proof, we introduce the following "alternative" definitions, that only considers variables that are present in the initial clauses (which are never deleted from the set of clauses, but only moved to another component).

fun *correct-watching′* :: ⟨′*v twl-st-wl* ⇒ *bool*⟩ **where**
⟨*correct-watching′* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*).
distinct-watched (*W L*) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*).
i ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧ *correctly-marked-as-binary N* (*i*, *K*, *b*)) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*).
b ⟶ *i* ∈# *dom-m N*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*W L*)) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))⟩

fun *correct-watching′′* :: ⟨′*v twl-st-wl* ⇒ *bool*⟩ **where**
⟨*correct-watching′′* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
(∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*).

distinct-watched (*W L*) ∧
(∀(*i*, *K*, *b*)∈#*mset* (*W L*).
i ∈# *dom-m N* ⟶ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L*) ∧
filter-mset (λ*i*. *i* ∈# *dom-m N*) (*fst* ‘# *mset* (*W L*)) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))›

lemma *correct-watching′-correct-watching″*: ‹*correct-watching′ S* ⟹ *correct-watching″ S*›
by (*cases S*) *auto*

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

definition *remove-all-annot-true-clause-imp-wl-inv*
:: ‹′*v twl-st-wl* ⇒ - ⇒ *nat* × ′*v twl-st-wl* ⇒ *bool*›
where
‹*remove-all-annot-true-clause-imp-wl-inv S xs* = (λ(*i*, *T*).
correct-watching″ S ∧ *correct-watching″ T* ∧
(∃ *S′ T′*. (*S*, *S′*) ∈ *state-wl-l None* ∧ (*T*, *T′*) ∈ *state-wl-l None* ∧
remove-all-annot-true-clause-imp-inv S′ xs (*i*, *T′*)))›

definition *remove-all-annot-true-clause-one-imp-wl*
where
‹*remove-all-annot-true-clause-one-imp-wl* = (λ(*C*, *S*). *do* {
if C ∈# *dom-m* (*get-clauses-wl S*) *then*
if irred (*get-clauses-wl S*) *C*
then RETURN (*drop-clause-add-move-init S C*)
else RETURN (*drop-clause S C*)
else do {
RETURN S
}
})›

definition *remove-all-annot-true-clause-imp-wl*
:: ‹′*v literal* ⇒ ′*v twl-st-wl* ⇒ (′*v twl-st-wl*) *nres*›
where
‹*remove-all-annot-true-clause-imp-wl* = (λ*L S*. *do* {
let xs = *get-watched-wl S L*;
(-, *T*) ← $WHILE_T{}^{\lambda(i,\ T).\ remove\text{-}all\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}wl\text{-}inv\ S\ xs\ (i,\ T)}$
(λ(*i*, *T*). *i* < *length xs*)
(λ(*i*, *T*). *do* {
ASSERT(*i* < *length xs*);
let (*C*, -, -) = *xs*!*i*;
if C ∈# *dom-m* (*get-clauses-wl T*) ∧ *length* ((*get-clauses-wl T*) ∝ *C*) ≠ *2*
then do {
T ← *remove-all-annot-true-clause-one-imp-wl* (*C*, *T*);
RETURN (*i*+*1*, *T*)
}
else
RETURN (*i*+*1*, *T*)
})
(*0*, *S*);
RETURN T
})›

lemma *reduce-dom-clauses-fmdrop*:

⟨*reduce-dom-clauses N0 N* ⟹ *reduce-dom-clauses N0* (*fmdrop C N*)⟩
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *reduce-dom-clauses-def distinct-mset-remove1-All*)

lemma *correct-watching-fmdrop*:
assumes
irred: ⟨¬ *irred N C*⟩ **and**
C: ⟨*C* ∈# *dom-m N*⟩ **and**
⟨*correct-watching′* (*M′*, *N*, *D*, *NE*, *UE*, *Q*, *W*)⟩ **and**
C2: ⟨*length* (*N* ∝ *C*) ≠ *2*⟩
shows ⟨*correct-watching′* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *Q*, *W*)⟩
proof −
have
Hdist: ⟨⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹
distinct-watched (*W L*)⟩ **and**
H1: ⟨⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)⟩ **and**
H1′: ⟨⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *b* ⟹ *i* ∈# *dom-m N*⟩ **and**
H2: ⟨⋀*L*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹
{#*i* ∈# *fst* ‘# *mset* (*W L*). *i* ∈# *dom-m N*#} =
{#*C* ∈# *dom-m* (*get-clauses-l* (*M′*, *N*, *D*, *NE*, *UE*, {#}, {#})).
L ∈ *set* (*watched-l* (*get-clauses-l* (*M′*, *N*, *D*, *NE*, *UE*, {#}, {#}) ∝ *C*))#}⟩
using *assms*
unfolding *correct-watching′.simps clause-to-update-def*
by *fast*+
have *1*: ⟨{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =
{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}⟩ **for** *L*
apply (*rule filter-mset-cong2*)
using *distinct-mset-dom*[*of N*] *C irred*
by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
have *2*: ⟨*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} =
removeAll-mset C {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}⟩ **for** *L*
apply (*rule distinct-mset-remove1-All*)
using *distinct-mset-dom*[*of N*]
by (*auto intro*: *distinct-mset-filter*)
have [*simp*]: ⟨*filter-mset* (λ*i*. *i* ∈# *remove1-mset C* (*dom-m N*)) *A* =
removeAll-mset C (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)⟩ **for** *A*
by (*induction A*)
(*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
show *?thesis*
unfolding *correct-watching′.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L* **using** *Hdist*[*of L*] *distinct-mset-dom*[*of N*]
H1[*of L* ⟨*fst iK*⟩ ⟨*fst* (*snd iK*)⟩ ⟨*snd* (*snd iK*)⟩] *C irred*
H1′[*of L* ⟨*fst iK*⟩ ⟨*fst* (*snd iK*)⟩ ⟨*snd* (*snd iK*)⟩]
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ⟨*fst iK*⟩ ⟨*fst* (*snd iK*)⟩ ⟨*snd* (*snd iK*)⟩] *C irred*

H1'[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›]
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred H1'*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C2*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
apply (*subst* (*asm*) *init-clss-lf-fmdrop-irrelev*, *assumption*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
qed

lemma *correct-watching''-fmdrop*:
assumes
irred: ‹¬ *irred N C*› **and**
C: ‹*C* ∈# *dom-m N*› **and**
‹*correct-watching''* (*M'*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›
shows ‹*correct-watching''* (*M*, *fmdrop C N*, *D*, *NE*, *UE*, *Q*, *W*)›
proof −
have
Hdist: ‹⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹ *distinct-watched* (*W L*)› **and**
H1: ‹⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹ (*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L*› **and**
H2: ‹⋀*L*. *L*∈#*all-lits-of-mm* (*mset* ‘# *init-clss-lf N* + *NE*) ⟹
{#*i* ∈# *fst* ‘# *mset* (*W L*). *i* ∈# *dom-m N*#} =
{#*C* ∈# *dom-m* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, {#}, {#})).
L ∈ *set* (*watched-l* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, {#}, {#}) ∝ *C*))#}›
using *assms*
unfolding *correct-watching''.simps clause-to-update-def*
by *fast+*
have *1*: ‹{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =
{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule filter-mset-cong2*)
using *distinct-mset-dom*[*of N*] *C irred*
by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
have *2*: ‹*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} =
removeAll-mset C {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule distinct-mset-remove1-All*)
using *distinct-mset-dom*[*of N*]
by (*auto intro*: *distinct-mset-filter*)
have [*simp*]: ‹*filter-mset* (λ*i*. *i* ∈# *remove1-mset C* (*dom-m N*)) *A* =
removeAll-mset C (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)› **for** *A*

by (*induction A*)
(*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
show *?thesis*
unfolding *correct-watching''.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L* **using** *Hdist*[*of L*] *distinct-mset-dom*[*of N*]
H1[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
apply (*subst* (*asm*) *init-clss-lf-fmdrop-irrelev*, *assumption*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
qed

lemma *correct-watching''-fmdrop'*:
assumes
irred: ‹*irred N C*› **and**
C: ‹*C* ∈# *dom-m N*› **and**
‹*correct-watching''* (*M'*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›
shows ‹*correct-watching''* (*M*, *fmdrop C N*, *D*, *add-mset* (*mset* (*N* ∝ *C*)) *NE*, *UE*, *Q*, *W*)›
proof −
have
Hdist: ‹⋀*L*. *L*∈#*all-lits-of-mm* (*mset* '# *init-clss-lf N* + *NE*) ⟹
distinct-watched (*W L*)› **and**
H1: ‹⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* '# *init-clss-lf N* + *NE*) ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹
K ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L*› **and**
H2: ‹⋀*L*. *L*∈#*all-lits-of-mm* (*mset* '# *init-clss-lf N* + *NE*) ⟹
{#*i* ∈# *fst* '# *mset* (*W L*). *i* ∈# *dom-m N*#} =
{#*C* ∈# *dom-m* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, {#}, {#})).
L ∈ *set* (*watched-l* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, {#}, {#}) ∝ *C*))#}›
using *assms*
unfolding *correct-watching''.simps clause-to-update-def*
by *blast*+
have *1*: ‹{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =
{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule filter-mset-cong2*)
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*snd iK*›] *C irred*
by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
have *2*: ‹*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} =

removeAll-mset C {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule distinct-mset-remove1-All*)
using *distinct-mset-dom*[*of N*]
by (*auto intro*: *distinct-mset-filter*)
have [*simp*]: ‹*filter-mset* (λ*i*. *i* ∈# *remove1-mset C* (*dom-m N*)) *A* =
removeAll-mset C (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)› **for** *A*
by (*induction A*)
(*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
show *?thesis*
unfolding *correct-watching''.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
Hdist[*of L*]
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset*
distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest: *all-lits-of-mm-diffD*
dest: *multi-member-split*)
done
qed

lemma *correct-watching''-fmdrop''*:
assumes
irred: ‹¬*irred N C*› **and**
C: ‹*C* ∈# *dom-m N*› **and**
‹*correct-watching''* (*M'*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›
shows ‹*correct-watching''* (*M*, *fmdrop C N*, *D*, *NE*, *add-mset* (*mset* (*N* ∝ *C*)) *UE*, *Q*, *W*)›
proof −
have
Hdist: ‹⋀*L*. *L*∈#*all-lits-of-mm* (*mset* '# *init-clss-lf N* + *NE*) ⟹
distinct-watched (*W L*)› **and**
H1: ‹⋀*L i K b*. *L*∈#*all-lits-of-mm* (*mset* '# *init-clss-lf N* + *NE*) ⟹
(*i*, *K*, *b*)∈#*mset* (*W L*) ⟹ *i* ∈# *dom-m N* ⟹ *K* ∈ *set* (*N* ∝ *i*) ∧
K ≠ *L*› **and**
H2: ‹⋀*L*. *L*∈#*all-lits-of-mm* (*mset* '# *init-clss-lf N* + *NE*) ⟹
{#*i* ∈# *fst* '# *mset* (*W L*). *i* ∈# *dom-m N*#} =
{#*C* ∈# *dom-m* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, {#}, {#})).
L ∈ *set* (*watched-l* (*get-clauses-l* (*M'*, *N*, *D*, *NE*, *UE*, {#}, {#}) ∝ *C*))#}›
using *assms*
unfolding *correct-watching''.simps clause-to-update-def*
by *blast+*
have *1*: ‹{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*fmdrop C N* ∝ *Ca*))#} =

{#*Ca* ∈# *dom-m* (*fmdrop C N*). *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule filter-mset-cong2*)
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*snd iK*›] *C irred*
by (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
have *2*: ‹*remove1-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#} = *removeAll-mset C* {#*Ca* ∈# *dom-m N*. *L* ∈ *set* (*watched-l* (*N* ∝ *Ca*))#}› **for** *L*
apply (*rule distinct-mset-remove1-All*)
using *distinct-mset-dom*[*of N*]
by (*auto intro*: *distinct-mset-filter*)
have [*simp*]: ‹*filter-mset* (λ*i*. *i* ∈# *remove1-mset C* (*dom-m N*)) *A* = *removeAll-mset C* (*filter-mset* (λ*i*. *i* ∈# *dom-m N*) *A*)› **for** *A*
by (*induction A*)
(*auto simp*: *distinct-mset-remove1-All distinct-mset-dom*)
show *?thesis*
unfolding *correct-watching′′.simps clause-to-update-def*
apply (*intro conjI impI ballI*)
subgoal for *L*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred Hdist*[*of L*]
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L iK*
using *distinct-mset-dom*[*of N*] *H1*[*of L* ‹*fst iK*› ‹*fst* (*snd iK*)› ‹*snd* (*snd iK*)›] *C irred*
apply (*auto simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond correctly-marked-as-binary.simps dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
subgoal for *L*
using *C irred* **apply** −
unfolding *get-clauses-l.simps*
apply (*subst 1*)
by (*auto 5 1 simp*: *image-mset-remove1-mset-if clause-to-update-def image-filter-replicate-mset distinct-mset-remove1-All filter-mset-neq-cond 2 H2 dest*: *all-lits-of-mm-diffD dest*: *multi-member-split*)
done
qed

definition *remove-one-annot-true-clause-one-imp-wl-pre* **where**
‹*remove-one-annot-true-clause-one-imp-wl-pre i T* ⟷
(∃ *T′*. (*T*, *T′*) ∈ *state-wl-l None* ∧
remove-one-annot-true-clause-one-imp-pre i T′ ∧
correct-watching′′ T)›

definition *remove-one-annot-true-clause-one-imp-wl*
:: ‹*nat* ⇒ ′*v twl-st-wl* ⇒ (*nat* × ′*v twl-st-wl*) *nres*›
where
‹*remove-one-annot-true-clause-one-imp-wl* = (λ*i S*. *do* {
ASSERT(*remove-one-annot-true-clause-one-imp-wl-pre i S*);
ASSERT(*is-proped* (*rev* (*get-trail-wl S*) ! *i*));
(*L*, *C*) ← *SPEC*(λ(*L*, *C*). (*rev* (*get-trail-wl S*))!*i* = *Propagated L C*);
ASSERT(*Propagated L C* ∈ *set* (*get-trail-wl S*));
if C = *0 then RETURN* (*i*+*1*, *S*)

else do {
ASSERT(C ∈# *dom-m* (*get-clauses-wl* S));
S ← *replace-annot-l* L C S;
S ← *remove-and-add-cls-l* C S;
— S ← *remove-all-annot-true-clause-imp-wl* L S;
RETURN (i+*1*, S)
}
})⟩

lemma *remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp*:
⟨(*uncurry remove-one-annot-true-clause-one-imp-wl*, *uncurry remove-one-annot-true-clause-one-imp*)
∈ *nat-rel* $\times_f$ {(S, T). (S, T) ∈ *state-wl-l None* ∧ *correct-watching*′′ S} $\rightarrow_f$
⟨*nat-rel* $\times_f$ {(S, T). (S, T) ∈ *state-wl-l None* ∧ *correct-watching*′′ S}⟩*nres-rel*⟩
(**is** ⟨- ∈ - $\times_f$ *?A* $\rightarrow_f$ -⟩)
proof −
have [*refine0*]: ⟨*replace-annot-l* L C S ≤
⇓ {(S', T'). (S', T') ∈ *?A* ∧ *get-clauses-wl* S' = *get-clauses-wl* S} (*replace-annot-l* L' C' T')⟩
if ⟨(L, L') ∈ *Id*⟩ **and** ⟨(S, T') ∈ *?A*⟩ **and** ⟨(C, C') ∈ *Id*⟩ **for** L L' S T' C C'
using *that* **by** (*cases* S; *cases* T')
(*fastforce simp*: *replace-annot-l-def state-wl-l-def*
correct-watching′′*.simps clause-to-update-def*
intro: *RES-refine*)
have [*refine0*]: ⟨*remove-and-add-cls-l* C S ≤⇓ *?A* (*remove-and-add-cls-l* C' S')⟩
if ⟨(C, C') ∈ *Id*⟩ **and** ⟨(S, S') ∈ *?A*⟩ **and**
⟨C ∈# *dom-m* (*get-clauses-wl* S)⟩
for C C' S S'
using *that* **unfolding** *remove-and-add-cls-l-def*
by *refine-rcg*
(*auto intro*!: *RES-refine simp*: *state-wl-l-def*
intro: *correct-watching*′′*-fmdrop correct-watching*′′*-fmdrop*′′
correct-watching′′*-fmdrop*′)
show *?thesis*
unfolding *remove-one-annot-true-clause-one-imp-wl-def remove-one-annot-true-clause-one-imp-def*
uncurry-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*)
subgoal for x y **unfolding** *remove-one-annot-true-clause-one-imp-wl-pre-def*
by (*rule exI*[*of* - ⟨*snd* y⟩]) *auto*
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by *simp*
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by *simp*
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*simp add*: *state-wl-l-def*)
subgoal by (*auto 5 5 simp add*: *state-wl-l-def*)
subgoal by (*auto simp add*: *state-wl-l-def*)
done
qed

definition *remove-one-annot-true-clause-imp-wl-inv* **where**
⟨*remove-one-annot-true-clause-imp-wl-inv* S = (λ(i, T).

$(\exists S'\ T'.\ (S,\ S') \in$ *state-wl-l None* $\wedge$ $(T,\ T') \in$ *state-wl-l None* $\wedge$
correct-watching'' S $\wedge$ *correct-watching''* T $\wedge$
remove-one-annot-true-clause-imp-inv S' $(i,\ T')))$›

definition *remove-one-annot-true-clause-imp-wl* :: ‹'v twl-st-wl ⇒ ('v twl-st-wl) nres›
where
‹*remove-one-annot-true-clause-imp-wl* = (λS. *do* {
  $k \leftarrow$ *SPEC*(λk. (∃ *M1 M2 K*. (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*get-trail-wl* S)) ∧
      count-decided M1 = *0* ∧ k = *length M1*)
    ∨ (*count-decided* (*get-trail-wl S*) = *0* ∧ k = *length* (*get-trail-wl S*)));
  (-, S) ← $WHILE_T^{remove\text{-}one\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}wl\text{-}inv\ S}$
   (λ(i, S). i < k)
   (λ(i, S). *remove-one-annot-true-clause-one-imp-wl i S*)
   (*0*, S);
  RETURN S
})›

lemma *remove-one-annot-true-clause-imp-wl-remove-one-annot-true-clause-imp*:
‹(*remove-one-annot-true-clause-imp-wl*, *remove-one-annot-true-clause-imp*)
  ∈ {(S, T). (S, T) ∈ *state-wl-l None* ∧ *correct-watching''* S} $\rightarrow_f$
   ⟨{(S, T). (S, T) ∈ *state-wl-l None* ∧ *correct-watching''* S}⟩*nres-rel*›
proof −
 show *?thesis*
  unfolding *remove-one-annot-true-clause-imp-wl-def remove-one-annot-true-clause-imp-def*
   uncurry-def
  apply (*intro frefI nres-relI*)
  apply (*refine-vcg*
   WHILEIT-refine[**where**
    R = ‹*nat-rel* $\times_f$ {(S, T). (S, T) ∈ *state-wl-l None* ∧ *correct-watching''* S}›]
  remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp[*THEN fref-to-Down-curry*])
  subgoal by *force*
  subgoal by *auto*
  subgoal for *x y k ka xa x'*
   unfolding *remove-one-annot-true-clause-imp-wl-inv-def*
   apply (*subst case-prod-beta*)
   apply (*rule-tac x=*‹y› **in** *exI*)
   apply (*rule-tac x=*‹*snd x'*› **in** *exI*)
   apply (*subst* (*asm*)(*17*) *surjective-pairing*)
   apply (*subst* (*asm*)(*22*) *surjective-pairing*)
   unfolding *prod-rel-iff* **by** *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  done
qed

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

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

correct-watching′ S)⟩

definition *mark-to-delete-clauses-wl-pre* :: ⟨′*v twl-st-wl* ⇒ *bool*⟩
where
⟨*mark-to-delete-clauses-wl-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧ *mark-to-delete-clauses-l-pre T*)⟩

definition *mark-garbage-wl*:: ⟨*nat* ⇒ ′*v twl-st-wl* ⇒ ′*v twl-st-wl*⟩ **where**
⟨*mark-garbage-wl* = (λ*C* (*M*, *N0*, *D*, *NE*, *UE*, *WS*, *Q*). (*M*, *fmdrop C N0*, *D*, *NE*, *UE*, *WS*, *Q*))⟩

definition *mark-to-delete-clauses-wl* :: ⟨′*v twl-st-wl* ⇒ ′*v twl-st-wl nres*⟩ **where**
⟨*mark-to-delete-clauses-wl* = (λ*S*. *do* {
  ASSERT(*mark-to-delete-clauses-wl-pre S*);
  xs ← *collect-valid-indices-wl S*;
  l ← *SPEC*(λ-:: *nat*. *True*);
  (-, *S*, -) ← $WHILE_T{}^{mark\text{-}to\text{-}delete\text{-}clauses\text{-}wl\text{-}inv\ S\ xs}$
    (λ(*i*, *S*, *xs*). *i* < *length xs*)
    (λ(*i*, *T*, *xs*). *do* {
      if(*xs*!*i* ∉# *dom-m* (*get-clauses-wl T*)) *then RETURN* (*i*, *T*, *delete-index-and-swap xs i*)
      else do {
        ASSERT(*0* < *length* (*get-clauses-wl T*∝(*xs*!*i*)));
        can-del ← *SPEC*(λ*b*. *b* ⟶
          (*Propagated* (*get-clauses-wl T*∝(*xs*!*i*)!*0*) (*xs*!*i*) ∉ *set* (*get-trail-wl T*)) ∧
          ¬*irred* (*get-clauses-wl T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl T* ∝ (*xs*!*i*)) ≠ *2*);
        ASSERT(*i* < *length xs*);
        if can-del
        then
          RETURN (*i*, *mark-garbage-wl* (*xs*!*i*) *T*, *delete-index-and-swap xs i*)
        else
          RETURN (*i*+*1*, *T*, *xs*)
      }
    })
    (*l*, *S*, *xs*);
  RETURN S
})⟩

lemma *mark-to-delete-clauses-wl-mark-to-delete-clauses-l*:
⟨(*mark-to-delete-clauses-wl*, *mark-to-delete-clauses-l*)
  ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching′ S*} $\rightarrow_f$
    ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching′ S*}⟩*nres-rel*⟩
proof −
  have [*refine0*]: ⟨*collect-valid-indices-wl S* ≤ ⇓ *Id* (*collect-valid-indices S′*)⟩
    if ⟨(*S*, *S′*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching′ S* ∧
        mark-to-delete-clauses-wl-pre S}⟩
    for *S S′*
    using *that* **by** (*auto simp*: *collect-valid-indices-wl-def collect-valid-indices-def*)
  have *if-inv*: ⟨(*if A then RETURN P else RETURN Q*) = *RETURN* (*if A then P else Q*)⟩ **for** *A P Q*
    by *auto*
  have *Ball-range*[*simp*]: ⟨(∀ *x*∈*range f* ∪ *range g*. *P x*)⟷(∀ *x*. *P* (*f x*) ∧ *P* (*g x*))⟩ **for** *P f g*
    by *auto*
  show *?thesis*
    unfolding *mark-to-delete-clauses-wl-def mark-to-delete-clauses-l-def*
      uncurry-def
    apply (*intro frefI nres-relI*)
    apply (*refine-vcg*
      WHILEIT-refine[**where**

$R = \langle\{((i, S, xs), (j, T, ys)).\ i = j \wedge (S, T) \in \textit{state-wl-l None} \wedge \textit{correct-watching}'\ S \wedge$
$xs = ys\}\rangle]$
remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp[*THEN fref-to-Down-curry*])
subgoal unfolding *mark-to-delete-clauses-wl-pre-def* **by** *blast*
subgoal by *auto*
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal unfolding *mark-to-delete-clauses-wl-inv-def* **by** *fast*
subgoal by *auto*
subgoal by (*force simp*: *state-wl-l-def*)
subgoal by *auto*
subgoal by (*force simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def can-delete-def*)
subgoal by *auto*
subgoal by (*force simp*: *state-wl-l-def*)
subgoal
by (*auto simp*: *state-wl-l-def correct-watching-fmdrop mark-garbage-wl-def*
mark-garbage-l-def
split: *prod.splits*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by *auto*
done
qed

This is only a specification and must be implemented. There are two ways to do so:

1. clean the watch lists and then iterate over all clauses to rebuild them.
2. iterate over the watch list and check whether the clause index is in the domain or not.

   It is not clear which is faster (but option 1 requires only 1 memory access per clause instead of two). The first option is implemented in SPASS-SAT. The latter version (partly) in cadical.

definition *rewatch-clauses* :: $\langle$*'v twl-st-wl* ⇒ *'v twl-st-wl nres*$\rangle$ **where**
$\langle$*rewatch-clauses* $= (\lambda(M, N, D, NE, UE, Q, W).\ SPEC(\lambda(M', N', D', NE', UE', Q', W').$
$(M, N, D, NE, UE, Q) = (M', N', D', NE', UE', Q') \wedge$
correct-watching $(M, N', D, NE, UE, Q, W'))))\rangle$

definition *mark-to-delete-clauses-wl-post* **where**
$\langle$*mark-to-delete-clauses-wl-post* $S\ T \longleftrightarrow$
$(\exists S'\ T'.\ (S, S') \in \textit{state-wl-l None} \wedge (T, T') \in \textit{state-wl-l None} \wedge$
mark-to-delete-clauses-l-post $S'\ T' \wedge$ *correct-watching* $S\ \wedge$
correct-watching $T)\rangle$

definition *cdcl-twl-full-restart-wl-prog* :: $\langle$*'v twl-st-wl* ⇒ *'v twl-st-wl nres*$\rangle$ **where**
$\langle$*cdcl-twl-full-restart-wl-prog* S = *do* {
— *remove-one-annot-true-clause-imp-wl S*
ASSERT(*mark-to-delete-clauses-wl-pre S*);
T ← *mark-to-delete-clauses-wl S*;
ASSERT(*mark-to-delete-clauses-wl-post S T*);
RETURN T
}$\rangle$

lemma *correct-watching-correct-watching*: $\langle$*correct-watching* $S \Longrightarrow$ *correct-watching'* $S\rangle$
apply (*cases S*, *simp only*: *correct-watching.simps correct-watching'.simps*)

apply (*subst* (*asm*) *all-clss-lf-ran-m*[*symmetric*])
unfolding *image-mset-union all-lits-of-mm-union*
by *auto*

lemma (**in** −) [*twl-st-l*, *simp*]:
‹(*Sa*, *x*) ∈ *twl-st-l None* ⟹ *get-all-learned-clss x* = *mset* '# (*get-learned-clss-l Sa*) + *get-unit-learned-clauses-l Sa*›
by (*cases Sa*; *cases x*) (*auto simp*: *twl-st-l-def get-learned-clss-l-def mset-take-mset-drop-mset'*)

lemma *cdcl-twl-full-restart-wl-prog-final-rel*:
assumes
S-Sa: ‹(*S*, *Sa*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching' S*}› **and**
pre-Sa: ‹*mark-to-delete-clauses-l-pre Sa*› **and**
pre-S: ‹*mark-to-delete-clauses-wl-pre S*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching' S*}› **and**
pre-l: ‹*mark-to-delete-clauses-l-post Sa Ta*›
shows ‹*mark-to-delete-clauses-wl-post S T*›
proof −
obtain *x* **where**
Sa-x: ‹(*Sa*, *x*) ∈ *twl-st-l None*› **and**
st: ‹*remove-one-annot-true-clause*** *Sa Ta*› **and**
list-invs: ‹*twl-list-invs Sa*› **and**
struct: ‹*twl-struct-invs x*› **and**
confl: ‹*get-conflict-l Sa* = *None*› **and**
upd: ‹*clauses-to-update-l Sa* = {#}›
using *pre-l*
unfolding *mark-to-delete-clauses-l-post-def* **by** *blast*

have *corr-S*: ‹*correct-watching' S*› **and** *corr-T*: ‹*correct-watching' T*› **and**
S-Sa: ‹(*S*, *Sa*) ∈ *state-wl-l None*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ *state-wl-l None*›
using *S-Sa T-Ta* **by** *auto*

have ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of x*)›
using *struct* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *auto*
then have ‹*set-mset* (*all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl S*) + *get-unit-init-clss-wl S*)) =
set-mset (*all-lits-of-mm* (*mset* '# *ran-mf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*))›
apply (*subst all-clss-lf-ran-m*[*symmetric*])
using *Sa-x S-Sa*
unfolding *image-mset-union* $cdcl_W$*-restart-mset.no-strange-atm-def all-lits-of-mm-union*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff get-learned-clss-l-def*
twl-st get-unit-clauses-wl-alt-def)

then have *corr-S'*: ‹*correct-watching S*›
using *corr-S*
by (*cases S*; *simp only*: *correct-watching'.simps correct-watching.simps*)
simp
obtain *y* **where**
‹*cdcl-twl-restart-l*** *Sa Ta*› **and**
Ta-y: ‹(*Ta*, *y*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *x y*› **and**
struct: ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF st list-invs confl upd Sa-x struct*]

by *blast*

have ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of y*)›
using *struct* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *auto*
then have ‹*set-mset* (*all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl T*) + *get-unit-init-clss-wl T*)) =
set-mset (*all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl T*) + *get-unit-clauses-wl T*))›
apply (*subst all-clss-lf-ran-m*[*symmetric*])
using *T-Ta Ta-y*
unfolding *image-mset-union* $cdcl_W$*-restart-mset.no-strange-atm-def all-lits-of-mm-union*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff get-learned-clss-l-def*
twl-st get-unit-clauses-wl-alt-def)

then have *corr-T′*: ‹*correct-watching T*›
using *corr-T*
by (*cases T*; *simp only*: *correct-watching′.simps correct-watching.simps*)
simp

show *?thesis*
using *S-Sa T-Ta corr-T′ corr-S′ pre-l*
unfolding *mark-to-delete-clauses-wl-post-def*
by *blast*
qed

lemma *cdcl-twl-full-restart-wl-prog-final-rel′*:
assumes
S-Sa: ‹(*S*, *Sa*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*}› **and**
pre-Sa: ‹*mark-to-delete-clauses-l-pre Sa*› **and**
pre-S: ‹*mark-to-delete-clauses-wl-pre S*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching′ S*}› **and**
pre-l: ‹*mark-to-delete-clauses-l-post Sa Ta*›
shows ‹*mark-to-delete-clauses-wl-post S T*›
proof −
obtain *x* **where**
Sa-x: ‹(*Sa*, *x*) ∈ *twl-st-l None*› **and**
st: ‹*remove-one-annot-true-clause*** *Sa Ta*› **and**
list-invs: ‹*twl-list-invs Sa*› **and**
struct: ‹*twl-struct-invs x*› **and**
confl: ‹*get-conflict-l Sa* = *None*› **and**
upd: ‹*clauses-to-update-l Sa* = {#}›
using *pre-l*
unfolding *mark-to-delete-clauses-l-post-def* **by** *blast*

have *corr-S*: ‹*correct-watching S*› **and** *corr-T*: ‹*correct-watching′ T*› **and**
S-Sa: ‹(*S*, *Sa*) ∈ *state-wl-l None*› **and**
T-Ta: ‹(*T*, *Ta*) ∈ *state-wl-l None*›
using *S-Sa T-Ta* **by** *auto*
have *corr-S*: ‹*correct-watching′ S*›
using *correct-watching-correct-watching*[*OF corr-S*] **.**
have ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of x*)›
using *struct* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *auto*
then have ‹*set-mset* (*all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl S*) + *get-unit-init-clss-wl S*)) =
set-mset (*all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl S*) + *get-unit-clauses-wl S*))›

apply (*subst all-clss-lf-ran-m*[*symmetric*])
using *Sa-x S-Sa*
unfolding *image-mset-union* $cdcl_W$*-restart-mset.no-strange-atm-def all-lits-of-mm-union*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff get-learned-clss-l-def*
twl-st get-unit-clauses-wl-alt-def)

then have *corr-S'*: ‹*correct-watching S*›
using *corr-S*
by (*cases S*; *simp only*: *correct-watching'.simps correct-watching.simps*)
simp
obtain *y* **where**
‹*cdcl-twl-restart-l*** *Sa Ta*› **and**
Ta-y: ‹(*Ta*, *y*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *x y*› **and**
struct: ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF st list-invs confl upd Sa-x struct*]
by *blast*

have ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of y*)›
using *struct* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *auto*
then have ‹*set-mset* (*all-lits-of-mm* (*mset* ‘# *init-clss-lf* (*get-clauses-wl T*) + *get-unit-init-clss-wl T*)) =
set-mset (*all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl T*) + *get-unit-clauses-wl T*))›
apply (*subst all-clss-lf-ran-m*[*symmetric*])
using *T-Ta Ta-y*
unfolding *image-mset-union* $cdcl_W$*-restart-mset.no-strange-atm-def all-lits-of-mm-union*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff get-learned-clss-l-def*
twl-st get-unit-clauses-wl-alt-def)

then have *corr-T'*: ‹*correct-watching T*›
using *corr-T*
by (*cases T*; *simp only*: *correct-watching'.simps correct-watching.simps*)
simp

show *?thesis*
using *S-Sa T-Ta corr-T' corr-S' pre-l*
unfolding *mark-to-delete-clauses-wl-post-def*
by *blast*
qed

lemma *cdcl-twl-full-restart-wl-prog-cdcl-full-twl-restart-l-prog*:
‹(*cdcl-twl-full-restart-wl-prog*, *cdcl-twl-full-restart-l-prog*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\rightarrow_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
unfolding *cdcl-twl-full-restart-wl-prog-def cdcl-twl-full-restart-l-prog-def*
rewatch-clauses-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
mark-to-delete-clauses-wl-mark-to-delete-clauses-l[*THEN fref-to-Down*]
remove-one-annot-true-clause-imp-wl-remove-one-annot-true-clause-imp[*THEN fref-to-Down*])
subgoal unfolding *mark-to-delete-clauses-wl-pre-def*
by (*blast intro*: *correct-watching-correct-watching*)
subgoal unfolding *mark-to-delete-clauses-wl-pre-def* **by** (*blast intro*: *correct-watching-correct-watching*)

subgoal
  by (*rule cdcl-twl-full-restart-wl-prog-final-rel′*)
subgoal by (*auto simp*: *state-wl-l-def mark-to-delete-clauses-wl-post-def*)
done

definition (**in** −) *cdcl-twl-local-restart-wl-spec* :: ‹′*v twl-st-wl* ⇒ ′*v twl-st-wl nres*› **where**
‹*cdcl-twl-local-restart-wl-spec* = (λ(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
  (*M*, *Q*) ← *SPEC*(λ(*M*′, *Q*′). (∃ *K M2*. (*Decided K* # *M*′, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
    Q′ = {#}) ∨ (*M*′ = *M* ∧ *Q*′ = *Q*));
  RETURN (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)
})›

lemma *cdcl-twl-local-restart-wl-spec-cdcl-twl-local-restart-l-spec*:
‹(*cdcl-twl-local-restart-wl-spec*, *cdcl-twl-local-restart-l-spec*)
  ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\rightarrow_f$
    ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
proof −
have [*refine0*]:
  ‹⋀*x y x1 x2 x1a x2a x1b x2b x1c x2c x1d x2d x1e x2e x1f x2f x1g x2g x1h x2h x1i x2i x1j x2j x1k x2k*.
    (*x*, *y*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} ⟹
    x2d = (*x1e*, *x2e*) ⟹
    x2c = (*x1d*, *x2d*) ⟹
    x2b = (*x1c*, *x2c*) ⟹
    x2a = (*x1b*, *x2b*) ⟹
    x2 = (*x1a*, *x2a*) ⟹
    y = (*x1*, *x2*) ⟹
    x2j = (*x1k*, *x2k*) ⟹
    x2i = (*x1j*, *x2j*) ⟹
    x2h = (*x1i*, *x2i*) ⟹
    x2g = (*x1h*, *x2h*) ⟹
    x2f = (*x1g*, *x2g*) ⟹
    x = (*x1f*, *x2f*) ⟹
    SPEC (λ(*M*′, *Q*′). (∃ *K M2*. (*Decided K* # *M*′, *M2*) ∈ *set* (*get-all-ann-decomposition x1f*) ∧
      Q′ = {#}) ∨ *M*′ = *x1f* ∧ *Q*′ = *x1k*)
     ≤ ⇓*Id* (*SPEC* (λ(*M*′, *Q*′) .(∃ *K M2*. (*Decided K* # *M*′, *M2*) ∈ *set* (*get-all-ann-decomposition x1*) ∧
      Q′ = {#}) ∨ *M*′ = *x1* ∧ *Q*′ = *x2e*))›
  by (*auto simp*: *state-wl-l-def*)
show *?thesis*
  unfolding *cdcl-twl-local-restart-wl-spec-def cdcl-twl-local-restart-l-spec-def*
    rewatch-clauses-def
  apply (*intro frefI nres-relI*)
  apply (*refine-vcg*)
  apply *assumption+*
  subgoal by (*auto simp*: *state-wl-l-def correct-watching.simps clause-to-update-def*)
  done
qed

definition *cdcl-twl-restart-wl-prog* **where**
‹*cdcl-twl-restart-wl-prog S* = *do* {
  b ← *SPEC*(λ-. *True*);
  if b then cdcl-twl-local-restart-wl-spec S else cdcl-twl-full-restart-wl-prog S
}›

lemma *cdcl-twl-restart-wl-prog-cdcl-twl-restart-l-prog*:

⟨(*cdcl-twl-restart-wl-prog*, *cdcl-twl-restart-l-prog*)
  $\in$ {(*S*, *T*). (*S*, *T*) $\in$ *state-wl-l None* $\wedge$ *correct-watching S*} $\rightarrow_f$
    ⟨{(*S*, *T*). (*S*, *T*) $\in$ *state-wl-l None* $\wedge$ *correct-watching S*}⟩*nres-rel*⟩
unfolding *cdcl-twl-restart-wl-prog-def cdcl-twl-restart-l-prog-def*
  rewatch-clauses-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg cdcl-twl-local-restart-wl-spec-cdcl-twl-local-restart-l-spec*[*THEN fref-to-Down*]
    cdcl-twl-full-restart-wl-prog-cdcl-full-twl-restart-l-prog[*THEN fref-to-Down*])
subgoal by *auto*
done

definition (**in** −) *restart-abs-wl-pre* :: ⟨′*v twl-st-wl* ⇒ *bool* ⇒ *bool*⟩ **where**
⟨*restart-abs-wl-pre S brk* ⟷
  ($\exists$ *S*′. (*S*, *S*′) $\in$ *state-wl-l None* $\wedge$ *restart-abs-l-pre S*′ *brk*
    $\wedge$ *correct-watching S*)⟩

context *twl-restart-ops*
begin

definition (**in** *twl-restart-ops*) *restart-required-wl* :: ⟨′*v twl-st-wl* ⇒ *nat* ⇒ *bool nres*⟩ **where**
⟨*restart-required-wl S n* = *SPEC* (λ*b*. *b* ⟶ *f n* < *size* (*get-learned-clss-wl S*))⟩

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-abs-wl-inv*
  :: ⟨′*v twl-st-wl* ⇒ *bool* ⇒ ′*v twl-st-wl* ⇒ *nat* ⇒ *bool*⟩ **where**
⟨*cdcl-twl-stgy-restart-abs-wl-inv* S_0 *brk T n* ≡
  ($\exists$ S_0′ *T*′.
    (S_0, S_0′) $\in$ *state-wl-l None* $\wedge$
    (*T*, *T*′) $\in$ *state-wl-l None* $\wedge$
    cdcl-twl-stgy-restart-abs-l-inv S_0′ *brk T*′ *n* $\wedge$
    correct-watching T)⟩
end

context *twl-restart-ops*
begin

definition *cdcl-GC-clauses-pre-wl* :: ⟨′*v twl-st-wl* ⇒ *bool*⟩ **where**
⟨*cdcl-GC-clauses-pre-wl S* ⟷ (
  $\exists$ *T*. (*S*, *T*) $\in$ *state-wl-l None* $\wedge$
    correct-watching′′ *S* $\wedge$
    cdcl-GC-clauses-pre T
)⟩

definition *cdcl-GC-clauses-wl* :: ⟨′*v twl-st-wl* ⇒ ′*v twl-st-wl nres*⟩ **where**
⟨*cdcl-GC-clauses-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). *do* {
  ASSERT(*cdcl-GC-clauses-pre-wl* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*));
  let b = *True*;
  if b then do {
    (*N*′, -) ← *SPEC* (λ(*N*′′, *m*). *GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N*′′) $\wedge$
      0 ∉# *dom-m N*′′);
    Q ← *SPEC*(λ*Q*. *correct-watching*′ (*M*, *N*′, *D*, *NE*, *UE*, *WS*, *Q*));
    RETURN (*M*, *N*′, *D*, *NE*, *UE*, *WS*, *Q*)
  }
  else RETURN (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)})⟩

lemma *cdcl-GC-clauses-wl-cdcl-GC-clauses*:
‹(*cdcl-GC-clauses-wl*, *cdcl-GC-clauses*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching'' S*} $\rightarrow_f$ ⟨{(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching' S*}⟩*nres-rel*›
unfolding *cdcl-GC-clauses-wl-def cdcl-GC-clauses-def*
apply (*intro frefI nres-relI*)
apply *refine-vcg*
subgoal unfolding *cdcl-GC-clauses-pre-wl-def* **by** *blast*
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by *auto*
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by *auto*
done

definition *cdcl-twl-full-restart-wl-GC-prog-post* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl* ⇒ *bool*› **where**
‹*cdcl-twl-full-restart-wl-GC-prog-post S T* ⟷
(∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧
cdcl-twl-full-restart-l-GC-prog-pre S' ∧
cdcl-twl-restart-l S' T' ∧ *correct-watching' T* ∧
set-mset (*all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl T*)+ *get-unit-init-clss-wl T*)) =
set-mset (*all-lits-of-mm* (*mset* '# *ran-mf* (*get-clauses-wl T*)+ *get-unit-clauses-wl T*)))›

definition (**in** −) *cdcl-twl-local-restart-wl-spec0* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*cdcl-twl-local-restart-wl-spec0* = (λ(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
(*M*, *Q*) ← *SPEC*(λ(*M'*, *Q'*). (∃ *K M2*. (*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
Q' = {#} ∧ *count-decided M'* = *0*) ∨ (*M'* = *M* ∧ *Q'* = *Q* ∧ *count-decided M'* = *0*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)
})›

definition *mark-to-delete-clauses-wl2-inv*
:: ‹'*v twl-st-wl* ⇒ *nat list* ⇒ *nat* × '*v twl-st-wl*× *nat list* ⇒ *bool*›
where
‹*mark-to-delete-clauses-wl2-inv* = (λ*S xs0* (*i*, *T*, *xs*).
∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧
mark-to-delete-clauses-l-inv S' xs0 (*i*, *T'*, *xs*) ∧
correct-watching'' S)›

definition *mark-to-delete-clauses-wl2* :: ‹'*v twl-st-wl* ⇒ '*v twl-st-wl nres*› **where**
‹*mark-to-delete-clauses-wl2* = (λ*S*. *do* {
ASSERT(*mark-to-delete-clauses-wl-pre S*);
xs ← *collect-valid-indices-wl S*;
l ← *SPEC*(λ-:: *nat*. *True*);
(-, *S*, -) ← $WHILE_T{}^{mark\text{-}to\text{-}delete\text{-}clauses\text{-}wl2\text{-}inv\ S\ xs}$
(λ(*i*, *S*, *xs*). *i* < *length xs*)
(λ(*i*, *T*, *xs*). *do* {
if(*xs*!*i* ∉# *dom-m* (*get-clauses-wl T*)) *then RETURN* (*i*, *T*, *delete-index-and-swap xs i*)
else do {
ASSERT(*0* < *length* (*get-clauses-wl T*∝(*xs*!*i*)));
can-del ← *SPEC*(λ*b*. *b* ⟶
(*Propagated* (*get-clauses-wl T*∝(*xs*!*i*)!*0*) (*xs*!*i*) ∉ *set* (*get-trail-wl T*)) ∧
¬*irred* (*get-clauses-wl T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl T* ∝ (*xs*!*i*)) ≠ *2*);
ASSERT(*i* < *length xs*);
if can-del

then
  RETURN (*i*, *mark-garbage-wl* (*xs*!*i*) *T*, *delete-index-and-swap xs i*)
else
  RETURN (*i*+*1*, *T*, *xs*)
}
})
(*l*, *S*, *xs*);
RETURN S
})›

lemma *mark-to-delete-clauses-wl-mark-to-delete-clauses-l2*:
‹(*mark-to-delete-clauses-wl2*, *mark-to-delete-clauses-l*)
  ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching''* *S*} $\to_f$
  ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching''* *S*}⟩*nres-rel*›
proof −
  have [*refine0*]: ‹*collect-valid-indices-wl S* ≤ ⇓ *Id* (*collect-valid-indices S'*)›
    if ‹(*S*, *S'*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching''* *S* ∧
        mark-to-delete-clauses-wl-pre S}›
    for *S S'*
    using *that* **by** (*auto simp*: *collect-valid-indices-wl-def collect-valid-indices-def*)
  have *if-inv*: ‹(*if A then RETURN P else RETURN Q*) = *RETURN* (*if A then P else Q*)› **for** *A P Q*
    by *auto*
  have *Ball-range*[*simp*]: ‹(∀ *x*∈*range f* ∪ *range g*. *P x*)⟷(∀ *x*. *P* (*f x*) ∧ *P* (*g x*))› **for** *P f g*
    by *auto*
  show *?thesis*
    unfolding *mark-to-delete-clauses-wl2-def mark-to-delete-clauses-l-def*
      uncurry-def
    apply (*intro frefI nres-relI*)
    apply (*refine-vcg*
      WHILEIT-refine[**where**
        R = ‹{((*i*, *S*, *xs*), (*j*, *T*, *ys*)). *i* = *j* ∧ (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching''* *S* ∧
          xs = *ys*}›]
    remove-one-annot-true-clause-one-imp-wl-remove-one-annot-true-clause-one-imp[*THEN fref-to-Down-curry*])
    subgoal unfolding *mark-to-delete-clauses-wl-pre-def* **by** *blast*
    subgoal by *auto*
    subgoal by (*auto simp*: *state-wl-l-def*)
    subgoal unfolding *mark-to-delete-clauses-wl2-inv-def* **by** *fast*
    subgoal by *auto*
    subgoal by (*force simp*: *state-wl-l-def*)
    subgoal by *auto*
    subgoal by (*force simp*: *state-wl-l-def*)
    subgoal by (*auto simp*: *state-wl-l-def can-delete-def*)
    subgoal by *auto*
    subgoal by (*force simp*: *state-wl-l-def*)
    subgoal
      by (*auto simp*: *state-wl-l-def correct-watching-fmdrop mark-garbage-wl-def*
          mark-garbage-l-def correct-watching''-fmdrop
        split: *prod.splits*)
    subgoal by (*auto simp*: *state-wl-l-def*)
    subgoal by *auto*
    done
qed

definition *cdcl-twl-full-restart-wl-GC-prog-pre*
  :: ‹'*v twl-st-wl* ⇒ *bool*›

where
‹*cdcl-twl-full-restart-wl-GC-prog-pre S* $\longleftrightarrow$
($\exists$ *T*. (*S*, *T*) $\in$ *state-wl-l None* $\wedge$ *correct-watching' S* $\wedge$ *cdcl-twl-full-restart-l-GC-prog-pre T*)›

definition *cdcl-twl-full-restart-wl-GC-prog* **where**
‹*cdcl-twl-full-restart-wl-GC-prog S* = *do* {
ASSERT(*cdcl-twl-full-restart-wl-GC-prog-pre S*);
S' $\leftarrow$ *cdcl-twl-local-restart-wl-spec0 S*;
T $\leftarrow$ *remove-one-annot-true-clause-imp-wl S'*;
ASSERT(*mark-to-delete-clauses-wl-pre T*);
U $\leftarrow$ *mark-to-delete-clauses-wl2 T*;
V $\leftarrow$ *cdcl-GC-clauses-wl U*;
ASSERT(*cdcl-twl-full-restart-wl-GC-prog-post S V*);
RETURN V
}›

lemma *cdcl-twl-local-restart-wl-spec0-cdcl-twl-local-restart-l-spec0*:
‹(*x*, *y*) $\in$ {(*S*, *S'*). (*S*, *S'*) $\in$ *state-wl-l None* $\wedge$ *correct-watching'' S*} $\Longrightarrow$
cdcl-twl-local-restart-wl-spec0 x
$\le$ $\Downarrow$ {(*S*, *S'*). (*S*, *S'*) $\in$ *state-wl-l None* $\wedge$ *correct-watching'' S*}
(*cdcl-twl-local-restart-l-spec0 y*)›
by (*cases x*; *cases y*)
(*auto simp*: *cdcl-twl-local-restart-wl-spec0-def cdcl-twl-local-restart-l-spec0-def*
state-wl-l-def image-iff correct-watching''.simps clause-to-update-def
conc-fun-RES RES-RETURN-RES2)

lemma *cdcl-twl-full-restart-wl-GC-prog-post-correct-watching*:
assumes
pre: ‹*cdcl-twl-full-restart-l-GC-prog-pre y*› **and**
y-Va: ‹*cdcl-twl-restart-l y Va*›
‹(*V*, *Va*) $\in$ {(*S*, *S'*). (*S*, *S'*) $\in$ *state-wl-l None* $\wedge$ *correct-watching' S*}›
shows ‹(*V*, *Va*) $\in$ {(*S*, *S'*). (*S*, *S'*) $\in$ *state-wl-l None* $\wedge$ *correct-watching S*}› **and**
‹*set-mset* (*all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl V*)+ *get-unit-init-clss-wl V*)) =
set-mset (*all-lits-of-mm* (*mset* '# *ran-mf* (*get-clauses-wl V*)+ *get-unit-clauses-wl V*))›
proof −
obtain *x* **where**
y-x: ‹(*y*, *x*) $\in$ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs x*› **and**
list-invs: ‹*twl-list-invs y*›
using *pre* **unfolding** *cdcl-twl-full-restart-l-GC-prog-pre-def* **by** *blast*
obtain *V'* **where** ‹*cdcl-twl-restart x V'*› **and** *Va-V'*: ‹(*Va*, *V'*) $\in$ *twl-st-l None*›
using *cdcl-twl-restart-l-cdcl-twl-restart*[*OF y-x list-invs struct-invs*] *y-Va*
unfolding *conc-fun-RES* **by** *auto*
then have ‹*twl-struct-invs V'*›
using *struct-invs* **by** (*blast dest*: *cdcl-twl-restart-twl-struct-invs*)
then have ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of V'*)›
unfolding *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *blast*
then show ‹*set-mset* (*all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl V*)+ *get-unit-init-clss-wl V*))
=
set-mset (*all-lits-of-mm* (*mset* '# *ran-mf* (*get-clauses-wl V*)+ *get-unit-clauses-wl V*))›
using *assms*(*3*) *Va-V'*
apply (*cases V*; *cases V'*)
apply (*auto simp*: *state-wl-l-def* $cdcl_W$*-restart-mset.no-strange-atm-def*
twl-st-l-def $cdcl_W$*-restart-mset-state image-image mset-take-mset-drop-mset'*
in-all-lits-of-mm-ain-atms-of-iff atms-of-ms-def atms-of-def atm-of-eq-atm-of

conj-disj-distribR Collect-disj-eq ex-disj-distrib
split: *if-splits*
dest!: *multi-member-split*[*of* - ‹*ran-m* -›])
apply (*auto dest*!: *split-list*
dest!: *multi-member-split*)
done
then have ‹*correct-watching*′ *V* ⟹ *correct-watching V*›
by (*cases V*)
(*auto simp*: *correct-watching.simps correct-watching*′*.simps*)
then show‹(*V*, *Va*) ∈ {(*S*, *S*′). (*S*, *S*′) ∈ *state-wl-l None* ∧ *correct-watching S*}›
using *assms* **by** (*auto simp*: *cdcl-twl-full-restart-wl-GC-prog-post-def*)
qed

lemma *cdcl-twl-full-restart-wl-GC-prog*:
‹(*cdcl-twl-full-restart-wl-GC-prog*, *cdcl-twl-full-restart-l-GC-prog*) ∈ {(*S*::′*v twl-st-wl*, *S*′).
(*S*, *S*′) ∈ *state-wl-l None* ∧ *correct-watching*′ *S*} $\rightarrow_f$ ⟨{(*S*::′*v twl-st-wl*, *S*′).
(*S*, *S*′) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
unfolding *cdcl-twl-full-restart-wl-GC-prog-def cdcl-twl-full-restart-l-GC-prog-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
remove-one-annot-true-clause-imp-wl-remove-one-annot-true-clause-imp[*THEN fref-to-Down*]
mark-to-delete-clauses-wl-mark-to-delete-clauses-l2[*THEN fref-to-Down*]
cdcl-GC-clauses-wl-cdcl-GC-clauses[*THEN fref-to-Down*]
cdcl-twl-local-restart-wl-spec0-cdcl-twl-local-restart-l-spec0)
subgoal unfolding *cdcl-twl-full-restart-wl-GC-prog-pre-def* **by** *blast*
subgoal by (*auto dest*: *correct-watching*′*-correct-watching*′′)
subgoal unfolding *mark-to-delete-clauses-wl-pre-def* **by** *fast*
subgoal for *x y S S*′ *T Ta U Ua V Va*
using *cdcl-twl-full-restart-wl-GC-prog-post-correct-watching*[*of y Va V*]
unfolding *cdcl-twl-full-restart-wl-GC-prog-post-def*
by *fast*
subgoal for *x y S*′ *S*′*a T Ta U Ua V Va*
by (*rule cdcl-twl-full-restart-wl-GC-prog-post-correct-watching*)
done

definition (**in** *twl-restart-ops*) *restart-prog-wl*
:: ′*v twl-st-wl* ⇒ *nat* ⇒ *bool* ⇒ (′*v twl-st-wl* × *nat*) *nres*
where
‹*restart-prog-wl S n brk* = *do* {
ASSERT(*restart-abs-wl-pre S brk*);
b ← *restart-required-wl S n*;
b2 ← *SPEC*(λ-. *True*);
if b2 ∧ *b* ∧ ¬*brk then do* {
T ← *cdcl-twl-full-restart-wl-GC-prog S*;
RETURN (*T*, *n* + *1*)
}
else if b ∧ ¬*brk then do* {
T ← *cdcl-twl-restart-wl-prog S*;
RETURN (*T*, *n* + *1*)
}
else
RETURN (*S*, *n*)
}›

lemma *cdcl-twl-full-restart-wl-prog-cdcl-twl-restart-l-prog*:

‹(*uncurry2 restart-prog-wl*, *uncurry2 restart-prog-l*)
  ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$
    ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\times_f$ *nat-rel*⟩*nres-rel*›
  (**is** ‹- ∈ *?R* $\times_f$ - $\times_f$ - $\rightarrow_f$ ⟨*?R′*⟩*nres-rel*›)
proof −
  have [*refine0*]: ‹*restart-required-wl a b* ≤ ⇓ *Id* (*restart-required-l a′ b′*)›
    if ‹(*a*, *a′*) ∈ *?R*› **and** ‹(*b*, *b′*) ∈ *nat-rel*› **for** *a a′ b b′*
    using *that* **unfolding** *restart-required-wl-def restart-required-l-def*
    by (*auto simp*: *twl-st-l*)
  show *?thesis*
    unfolding *uncurry-def restart-prog-wl-def restart-prog-l-def rewatch-clauses-def*
    apply (*intro frefI nres-relI*)
    apply (*refine-rcg*
      cdcl-twl-restart-wl-prog-cdcl-twl-restart-l-prog[*THEN fref-to-Down*]
      cdcl-twl-full-restart-wl-GC-prog[*THEN fref-to-Down*])
    subgoal unfolding *restart-abs-wl-pre-def*
       by (*fastforce simp*: *correct-watching-correct-watching*)
    subgoal by *auto*
    subgoal by *auto*
    subgoal by *auto*
    subgoal by (*auto simp*: *correct-watching-correct-watching*)
    subgoal by *auto*
    subgoal by *auto*
    subgoal
      by *auto*
    subgoal by *auto*
    subgoal by *auto*
    done
qed

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-prog-wl*
  :: ‹′*v twl-st-wl* ⇒ ′*v twl-st-wl nres*›
where
  ‹*cdcl-twl-stgy-restart-prog-wl* (S_0::′*v twl-st-wl*) =
  do {
    (*brk*, *T*, -) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}inv\ S_0\ brk\ T\ n}$
      (λ(*brk*, -). ¬*brk*)
      (λ(*brk*, *S*, *n*).
      do {
        T ← *unit-propagation-outer-loop-wl S*;
        (*brk*, *T*) ← *cdcl-twl-o-prog-wl T*;
        (*T*, *n*) ← *restart-prog-wl T n brk*;
        RETURN (*brk*, *T*, *n*)
      })
      (*False*, S_0::′*v twl-st-wl*, *0*);
    RETURN T
  }›

lemma *cdcl-twl-stgy-restart-prog-wl-cdcl-twl-stgy-restart-prog-l*:
  ‹(*cdcl-twl-stgy-restart-prog-wl*, *cdcl-twl-stgy-restart-prog-l*)
    ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\rightarrow_f$
      ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
  (**is** ‹- ∈ *?R* $\rightarrow_f$ ⟨*?S*⟩*nres-rel*›)
proof −

have [*refine0*]:
‹$(x,\ y) \in\ ?R \Longrightarrow ((False,\ x,\ 0),\ False,\ y,\ 0) \in bool\text{-}rel \times_r\ ?R \times_r\ nat\text{-}rel$› **for** *x y*
by *auto*
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-wl-def cdcl-twl-stgy-restart-prog-l-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg WHILEIT-refine*[**where**
R=‹$\{(S,\ T).\ (S,\ T) \in state\text{-}wl\text{-}l\ None \wedge\ correct\text{-}watching\ S\}$›]
unit-propagation-outer-loop-wl-spec[*THEN fref-to-Down*]
cdcl-twl-full-restart-wl-prog-cdcl-twl-restart-l-prog[*THEN fref-to-Down-curry2*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-inv-def* **by** *fastforce*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *correct-watching-correct-watching*)
subgoal by *auto*
subgoal by *auto*
done
qed

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-prog-early-wl*
:: ‹$'v\ twl\text{-}st\text{-}wl \Rightarrow 'v\ twl\text{-}st\text{-}wl\ nres$›
where
‹*cdcl-twl-stgy-restart-prog-early-wl* (S_0::$'v$ *twl-st-wl*) = *do* {
ebrk ← *RES UNIV*;
(-, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(\text{-},\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(-, *brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-wl S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-wl T*;
(*T*, *n*) ← *restart-prog-wl T n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk*, *T*, *n*)
})
(*ebrk*, *False*, S_0::$'v$ *twl-st-wl*, *0*);
if ¬ *brk then do* {
(*brk*, *T*, -) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-wl S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-wl T*;
(*T*, *n*) ← *restart-prog-wl T n brk*;
RETURN (*brk*, *T*, *n*)
})
(*False*, *T*::$'v$ *twl-st-wl*, *n*);
RETURN T
}
else RETURN T
}›

lemma *cdcl-twl-stgy-restart-prog-early-wl-cdcl-twl-stgy-restart-prog-early-l*:

‹(*cdcl-twl-stgy-restart-prog-early-wl*, *cdcl-twl-stgy-restart-prog-early-l*)
  ∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\rightarrow_f$
    ⟨{(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
 (**is** ‹- ∈ *?R* $\rightarrow_f$ ⟨*?S*⟩*nres-rel*›)
proof −
 show *?thesis*
  unfolding *cdcl-twl-stgy-restart-prog-early-wl-def cdcl-twl-stgy-restart-prog-early-l-def*
  apply (*intro frefI nres-relI*)
  apply (*refine-rcg WHILEIT-refine*[**where** *R*=‹*bool-rel* $\times_r$ *?R* $\times_r$ *nat-rel*›]
    WHILEIT-refine[**where** *R*=‹*bool-rel* $\times_r$ *bool-rel* $\times_r$ *?R* $\times_r$ *nat-rel*›]
   unit-propagation-outer-loop-wl-spec[*THEN fref-to-Down*]
   cdcl-twl-full-restart-wl-prog-cdcl-twl-restart-l-prog[*THEN fref-to-Down-curry2*]
   cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
  subgoal by *auto*
  subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-inv-def* **by** *fastforce*
  subgoal by *auto*
  subgoal by (*auto simp*: *correct-watching-correct-watching*)
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by (*auto simp*: *correct-watching-correct-watching*)
  subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-inv-def* **by** *fastforce*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  subgoal by *auto*
  done
qed

theorem *cdcl-twl-stgy-restart-prog-wl-spec*:
 ‹(*cdcl-twl-stgy-restart-prog-wl*, *cdcl-twl-stgy-restart-prog-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
    (*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*} → ⟨*state-wl-l None*⟩*nres-rel*›
  (**is** ‹*?o* ∈ *?A* → ⟨*?B*⟩ *nres-rel*›)
 using *cdcl-twl-stgy-restart-prog-wl-cdcl-twl-stgy-restart-prog-l*[**where** '*a*='*v*]
 unfolding *fref-param1* **apply** −
 apply (*match-spec*; *match-fun-rel+*; (*fast intro*: *nres-rel-mono*)?)
 by (*metis* (*no-types*, *lifting*) *in-pair-collect-simp nres-rel-mono subrelI*)

theorem *cdcl-twl-stgy-restart-prog-early-wl-spec*:
 ‹(*cdcl-twl-stgy-restart-prog-early-wl*, *cdcl-twl-stgy-restart-prog-early-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
    (*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*} → ⟨*state-wl-l None*⟩*nres-rel*›
  (**is** ‹*?o* ∈ *?A* → ⟨*?B*⟩ *nres-rel*›)
 using *cdcl-twl-stgy-restart-prog-early-wl-cdcl-twl-stgy-restart-prog-early-l*[**where** '*a*='*v*]
 unfolding *fref-param1* **apply** −
 by (*match-spec*; *match-fun-rel+*; (*fast intro*: *nres-rel-mono*)?; *match-fun-rel?*)
  auto

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-prog-bounded-wl*
 :: ‹'*v twl-st-wl* ⇒ (*bool* × '*v twl-st-wl*) *nres*›
where
 ‹*cdcl-twl-stgy-restart-prog-bounded-wl* (S_0::'*v twl-st-wl*) = *do* {
  ebrk ← *RES UNIV*;
  (-, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(\text{-},\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}inv\ S_0\ brk\ T\ n}$

$(\lambda(ebrk, brk, -). \neg brk \wedge \neg ebrk)$
$(\lambda(-, brk, S, n).$
do {
$T \leftarrow$ *unit-propagation-outer-loop-wl* S;
$(brk, T) \leftarrow$ *cdcl-twl-o-prog-wl* T;
$(T, n) \leftarrow$ *restart-prog-wl* T n brk;
$ebrk \leftarrow$ *RES UNIV*;
RETURN $(ebrk, brk, T, n)$
})
$(ebrk, False, S_0::'v\ twl\text{-}st\text{-}wl, 0)$;
RETURN (brk, T)
}›

lemma *cdcl-twl-stgy-restart-prog-bounded-wl-cdcl-twl-stgy-restart-prog-bounded-l*:
‹(*cdcl-twl-stgy-restart-prog-bounded-wl*, *cdcl-twl-stgy-restart-prog-bounded-l*)
$\in \{(S, T).\ (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\} \rightarrow_f$
$\langle$*bool-rel* $\times_r \{(S, T).\ (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\}\rangle$*nres-rel*›
(**is** ‹- ∈ ?R $\rightarrow_f$ ⟨?S⟩*nres-rel*›)
proof −
show *?thesis*
unfolding *cdcl-twl-stgy-restart-prog-bounded-wl-def cdcl-twl-stgy-restart-prog-bounded-l-def*
apply (*intro frefI nres-relI*)
apply (*refine-rcg*
WHILEIT-refine[**where** R=‹*bool-rel* $\times_r$ *bool-rel* $\times_r$ *?R* $\times_r$ *nat-rel*›]
unit-propagation-outer-loop-wl-spec[*THEN fref-to-Down*]
cdcl-twl-full-restart-wl-prog-cdcl-twl-restart-l-prog[*THEN fref-to-Down-curry2*]
cdcl-twl-o-prog-wl-spec[*THEN fref-to-Down*])
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-inv-def* **by** *fastforce*
subgoal by *auto*
subgoal by (*auto simp*: *correct-watching-correct-watching*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

theorem *cdcl-twl-stgy-restart-prog-bounded-wl-spec*:
‹(*cdcl-twl-stgy-restart-prog-bounded-wl*, *cdcl-twl-stgy-restart-prog-bounded-l*) $\in \{(S::'v\ twl\text{-}st\text{-}wl, S').$
$(S, S') \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\} \rightarrow \langle$*bool-rel* $\times_r$ *state-wl-l None*$\rangle$*nres-rel*›
(**is** ‹?o ∈ ?A → ⟨?B⟩ *nres-rel*›)
using *cdcl-twl-stgy-restart-prog-bounded-wl-cdcl-twl-stgy-restart-prog-bounded-l*[**where** $'a='v$]
unfolding *fref-param1* **apply** −
by (*match-spec*; *match-fun-rel*+; (*fast intro*: *nres-rel-mono*)?; *match-fun-rel?*)
auto

end

end
theory *Watched-Literals-Watch-List-Domain*
imports *Watched-Literals-Watch-List*
begin

We refine the implementation by adding a *domain* on the literals

1.4.4 State Conversion

Functions and Types:

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

1.4.5 Refinement

Set of all literals of the problem

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

abbreviation *all-lits-st* :: ⟨′*v twl-st-wl* ⇒ ′*v literal multiset*⟩ **where**
⟨*all-lits-st S* ≡ *all-lits* (*get-clauses-wl S*) (*get-unit-clauses-wl S*)⟩

definition *all-atms* :: ⟨- ⇒ - ⇒ ′*v multiset*⟩ **where**
⟨*all-atms N NUE* = *atm-of* ‘# *all-lits N NUE*⟩

abbreviation *all-atms-st* :: ⟨′*v twl-st-wl* ⇒ ′*v multiset*⟩ **where**
⟨*all-atms-st S* ≡ *atm-of* ‘# *all-lits-st S*⟩

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

context
fixes $\mathcal{A}_{in}$ :: ⟨*nat multiset*⟩
begin

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

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

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

definition *is-*$\mathcal{L}_{all}$ :: ⟨*nat literal multiset* ⇒ *bool*⟩ **where**
⟨*is-*$\mathcal{L}_{all}$ *S* ⟷ *set-mset* $\mathcal{L}_{all}$ = *set-mset S*⟩

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

lemma *literals-are-in-*$\mathcal{L}_{in}$*-empty*[*simp*]: ⟨*literals-are-in-*$\mathcal{L}_{in}$ {#}⟩
by (*auto simp*: *literals-are-in-*$\mathcal{L}_{in}$*-def*)

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

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

lemma *literals-are-in-*$\mathcal{L}_{in}$*-mono*:
assumes *N*: ⟨*literals-are-in-*$\mathcal{L}_{in}$ *D*′⟩ **and** *D*: ⟨*D* ⊆# *D*′⟩
shows ⟨*literals-are-in-*$\mathcal{L}_{in}$ *D*⟩
proof −

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

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

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

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

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

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

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

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

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

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

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

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

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

end

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

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

definition *blits-in-$\mathcal{L}_{in}$* :: ⟨*nat twl-st-wl* ⇒ *bool*⟩ **where**
⟨*blits-in-$\mathcal{L}_{in}$ S* ⟷
(∀ *L* ∈# $\mathcal{L}_{all}$ (*all-atms-st S*). ∀ (*i*, *K*, *b*) ∈ *set* (*watched-by S L*). *K* ∈# $\mathcal{L}_{all}$ (*all-atms-st S*))⟩

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

lemma *literals-are-in-$\mathcal{L}_{in}$-nth*:
fixes *C* :: *nat*
assumes *dom*: ⟨*C* ∈# *dom-m* (*get-clauses-wl S*)⟩ **and**
⟨*literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ *S*⟩
shows ⟨*literals-are-in-$\mathcal{L}_{in}$* $\mathcal{A}$ (*mset* (*get-clauses-wl S* ∝ *C*))⟩
proof −
let *?N* = ⟨*get-clauses-wl S*⟩
have ⟨*?N* ∝ *C* ∈# *ran-mf ?N*⟩
using *dom* **by** (*auto simp*: *ran-m-def*)
then have ⟨*mset* (*?N* ∝ *C*) ∈# *mset* '# (*ran-mf ?N*)⟩
by *blast*
from *all-lits-of-m-subset-all-lits-of-mmD*[*OF this*] **show** *?thesis*
using *assms*(*2*) **unfolding** *is-$\mathcal{L}_{all}$-def literals-are-in-$\mathcal{L}_{in}$-def literals-are-$\mathcal{L}_{in}$-def*
by (*auto simp add*: *all-lits-of-mm-union all-lits-def*)
qed

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

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-in-lits-of-l-atms*:

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

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

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

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

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

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

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

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

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

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

lemma *in-$\mathcal{L}_{all}$-atm-of-$\mathcal{A}_{in}$*: ‹*L* $\in$# $\mathcal{L}_{all}$ $\mathcal{A}_{in}$ $\longleftrightarrow$ *atm-of L* $\in$# $\mathcal{A}_{in}$›
  by (*cases L*) (*auto simp*: $\mathcal{L}_{all}$*-def*)

lemma *literals-are-in-$\mathcal{L}_{in}$-alt-def*:
  ‹*literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}$ S $\longleftrightarrow$ atms-of S $\subseteq$ atms-of ($\mathcal{L}_{all}$ $\mathcal{A}$)*›
  apply (*auto simp*: *literals-are-in-$\mathcal{L}_{in}$-def all-lits-of-mm-union lits-of-def*
      in-all-lits-of-m-ain-atms-of-iff in-all-lits-of-mm-ain-atms-of-iff atms-of-$\mathcal{L}_{all}$-$\mathcal{A}_{in}$
      atm-of-eq-atm-of uminus-$\mathcal{A}_{in}$-iff subset-iff in-$\mathcal{L}_{all}$-atm-of-$\mathcal{A}_{in}$)
  apply (*auto simp*: *atms-of-def*)
  done

lemma
  assumes
      x2-T: ‹(*x2*, *T*) $\in$ *state-wl-l b*› **and**
      struct: ‹*twl-struct-invs U*› **and**
      T-U: ‹(*T*, *U*) $\in$ *twl-st-l b'*›
  shows
    literals-are-$\mathcal{L}_{in}$-literals-are-$\mathcal{L}_{in}$-trail:
      ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2 $\Longrightarrow$ literals-are-in-$\mathcal{L}_{in}$-trail $\mathcal{A}_{in}$ (get-trail-wl x2)*›
     (**is** ‹*- $\Longrightarrow$?trail*›) **and**
    literals-are-$\mathcal{L}_{in}$-literals-are-in-$\mathcal{L}_{in}$-conflict:
    ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2 $\Longrightarrow$ get-conflict-wl x2 $\neq$ None $\Longrightarrow$ literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (the (get-conflict-wl x2))*› **and**
    conflict-not-tautology:
      ‹*get-conflict-wl x2 $\neq$ None $\Longrightarrow$ $\neg$tautology (the (get-conflict-wl x2))*›
proof −
  have
    alien: ‹*cdcl$_W$-restart-mset.no-strange-atm (state$_W$-of U)*› **and**
    confl: ‹*cdcl$_W$-restart-mset.cdcl$_W$-conflicting (state$_W$-of U)*› **and**
    M-lev: ‹*cdcl$_W$-restart-mset.cdcl$_W$-M-level-inv (state$_W$-of U)*› **and**
    dist: ‹*cdcl$_W$-restart-mset.distinct-cdcl$_W$-state (state$_W$-of U)*›
   using *struct* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
   by *fast*+

  show *lits-trail*: ‹*literals-are-in-$\mathcal{L}_{in}$-trail $\mathcal{A}_{in}$ (get-trail-wl x2)*›
    if ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ x2*›
    using *alien that x2-T T-U* **unfolding** *is-$\mathcal{L}_{all}$-def*
      literals-are-in-$\mathcal{L}_{in}$-trail-def cdcl$_W$-restart-mset.no-strange-atm-def
      literals-are-$\mathcal{L}_{in}$-def all-lits-def all-atms-def
    by (*subst* (*asm*) *all-clss-l-ran-m*[*symmetric*])
     (*auto 5 2*
        simp del: *all-clss-l-ran-m union-filter-mset-complement*
        simp: *twl-st twl-st-l twl-st-wl all-lits-of-mm-union lits-of-def*
        convert-lits-l-def image-image in-all-lits-of-mm-ain-atms-of-iff
        get-unit-clauses-wl-alt-def)

  {
    assume *conf*: ‹*get-conflict-wl x2 $\neq$ None*›
    show *lits-confl*: ‹*literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (the (get-conflict-wl x2))*›

if ⟨*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ *x2*⟩
using *x2-T T-U alien that conf* **unfolding** *is-*$\mathcal{L}_{all}$*-alt-def*
cdcl$_W$*-restart-mset.no-strange-atm-def literals-are-in-*$\mathcal{L}_{in}$*-alt-def*
literals-are-$\mathcal{L}_{in}$*-def all-lits-def all-atms-def*
apply (*subst* (*asm*) *all-clss-l-ran-m*[*symmetric*])
unfolding *image-mset-union all-lits-of-mm-union*
by (*auto simp add*: *twl-st all-lits-of-mm-union lits-of-def*
image-image in-all-lits-of-mm-ain-atms-of-iff
in-all-lits-of-m-ain-atms-of-iff
get-unit-clauses-wl-alt-def
simp del: *all-clss-l-ran-m*)

have *M-confl*: ⟨*get-trail-wl x2* $\models$*as CNot* (*the* (*get-conflict-wl x2*))⟩
using *confl conf x2-T T-U* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
by (*auto 5 5 simp*: *twl-st true-annots-def*)
moreover have *n-d*: ⟨*no-dup* (*get-trail-wl x2*)⟩
using *M-lev x2-T T-U* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-M-level-inv-def*
by (*auto simp*: *twl-st*)
ultimately show *4*: ⟨¬*tautology* (*the* (*get-conflict-wl x2*))⟩
using *n-d M-confl*
by (*meson no-dup-consistentD tautology-decomp′ true-annots-true-cls-def-iff-negation-in-model*)
}
qed

lemma *literals-are-in-*$\mathcal{L}_{in}$*-trail-atm-of*:
⟨*literals-are-in-*$\mathcal{L}_{in}$*-trail* $\mathcal{A}_{in}$ *M* ⟷ *atm-of* ‘ *lits-of-l M* ⊆ *set-mset* $\mathcal{A}_{in}$⟩
apply (*rule iffI*)
subgoal by (*auto dest*: *literals-are-in-*$\mathcal{L}_{in}$*-trail-in-lits-of-l-atms*)
subgoal by (*fastforce simp*: *literals-are-in-*$\mathcal{L}_{in}$*-trail-def lits-of-def in-*$\mathcal{L}_{all}$*-atm-of-*$\mathcal{A}_{in}$)
done

lemma *literals-are-in-*$\mathcal{L}_{in}$*-poss-remdups-mset*:
⟨*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (*poss* (*remdups-mset* (*atm-of* ‘# *C*))) ⟷ *literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ *C*⟩
by (*induction C*)
(*auto simp*: *literals-are-in-*$\mathcal{L}_{in}$*-add-mset in-*$\mathcal{L}_{all}$*-atm-of-in-atms-of-iff atm-of-eq-atm-of*
dest!: *multi-member-split*)

lemma *literals-are-in-*$\mathcal{L}_{in}$*-negs-remdups-mset*:
⟨*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (*negs* (*remdups-mset* (*atm-of* ‘# *C*))) ⟷ *literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ *C*⟩
by (*induction C*)
(*auto simp*: *literals-are-in-*$\mathcal{L}_{in}$*-add-mset in-*$\mathcal{L}_{all}$*-atm-of-in-atms-of-iff atm-of-eq-atm-of*
dest!: *multi-member-split*)

lemma $\mathcal{L}_{all}$*-atm-of-all-lits-of-m*:
⟨*set-mset* ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits-of-m C*)) = *set-mset C* ∪ *uminus* ‘ *set-mset C*⟩
supply *lit-eq-Neg-Pos-iff*[*iff*]
by (*auto simp*: $\mathcal{L}_{all}$*-def all-lits-of-m-def image-iff dest*!: *multi-member-split*)

lemma *atm-of-all-lits-of-mm*:
⟨*set-mset* (*atm-of* ‘# *all-lits-of-mm bw*) = *atms-of-mm bw*⟩
⟨*atm-of* ‘ *set-mset* (*all-lits-of-mm bw*) = *atms-of-mm bw*⟩
using *in-all-lits-of-mm-ain-atms-of-iff* **apply** (*auto simp*: *image-iff*)
by (*metis* (*full-types*) *image-eqI literal.sel*(*1*))+

lemma *in-set-all-atms-iff*:
⟨*y* ∈# *all-atms bu bw* ⟷

$y \in$ *atms-of-mm* (*mset* ‘# *ran-mf bu*) $\vee$ $y \in$ *atms-of-mm bw*›
by (*auto simp*: *all-atms-def all-lits-def in-all-lits-of-mm-ain-atms-of-iff atm-of-all-lits-of-mm*)

lemma $\mathcal{L}_{all}$-*union*:
‹*set-mset* ($\mathcal{L}_{all}$ ($A + B$)) = *set-mset* ($\mathcal{L}_{all}$ A) $\cup$ *set-mset* ($\mathcal{L}_{all}$ B)›
by (*auto simp*: $\mathcal{L}_{all}$-*def*)

lemma $\mathcal{L}_{all}$-*cong*:
‹*set-mset* A = *set-mset* B $\Longrightarrow$ *set-mset* ($\mathcal{L}_{all}$ A) = *set-mset* ($\mathcal{L}_{all}$ B)›
by (*auto simp*: $\mathcal{L}_{all}$-*def*)

lemma *atms-of*-$\mathcal{L}_{all}$-*cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ $\Longrightarrow$ *atms-of* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *atms-of* ($\mathcal{L}_{all}$ $\mathcal{B}$)›
unfolding $\mathcal{L}_{all}$-*def*
by *auto*

definition *unit-prop-body-wl-D-inv*
:: ‹*nat twl-st-wl* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat literal* $\Rightarrow$ *bool*› **where**
‹*unit-prop-body-wl-D-inv* T' j w L $\longleftrightarrow$
unit-prop-body-wl-inv T' j w L $\wedge$ *literals-are*-$\mathcal{L}_{in}$ (*all-atms-st* T') T' $\wedge$ $L \in\#$ $\mathcal{L}_{all}$ (*all-atms-st* T')›

- should be the definition of *unit-prop-body-wl-find-unwatched-inv*.
- the distinctiveness should probably be only a property, not a part of the definition.

definition *unit-prop-body-wl-D-find-unwatched-inv* **where**
‹*unit-prop-body-wl-D-find-unwatched-inv* f C S $\longleftrightarrow$
unit-prop-body-wl-find-unwatched-inv f C S $\wedge$
($f \neq$ *None* $\longrightarrow$ *the* $f \geq 2$ $\wedge$ *the* $f <$ *length* (*get-clauses-wl* $S \propto C$) $\wedge$
get-clauses-wl $S \propto C$! (*the* f) $\neq$ *get-clauses-wl* $S \propto C$! *0* $\wedge$
get-clauses-wl $S \propto C$! (*the* f) $\neq$ *get-clauses-wl* $S \propto C$! *1*)›

definition *unit-propagation-inner-loop-wl-loop-D-inv* **where**
‹*unit-propagation-inner-loop-wl-loop-D-inv* L = (λ(j, w, S).
literals-are-$\mathcal{L}_{in}$ (*all-atms-st* S) S $\wedge$ $L \in\#$ $\mathcal{L}_{all}$ (*all-atms-st* S) $\wedge$
unit-propagation-inner-loop-wl-loop-inv L (j, w, S))›

definition *unit-propagation-inner-loop-wl-loop-D-pre* **where**
‹*unit-propagation-inner-loop-wl-loop-D-pre* L = (λ(j, w, S).
unit-propagation-inner-loop-wl-loop-D-inv L (j, w, S) $\wedge$
unit-propagation-inner-loop-wl-loop-pre L (j, w, S))›

definition *unit-propagation-inner-loop-body-wl-D*
:: ‹*nat literal* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat twl-st-wl* $\Rightarrow$
(*nat* $\times$ *nat* $\times$ *nat twl-st-wl*) *nres*› **where**
‹*unit-propagation-inner-loop-body-wl-D* L j w S = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-D-pre* L (j, w, S));
let (C, K, b) = (*watched-by* S L) ! w;
let S = *keep-watch* L j w S;
ASSERT(*unit-prop-body-wl-D-inv* S j w L);
let val-K = *polarity* (*get-trail-wl* S) K;
if val-K = *Some True*
then RETURN (j+*1*, w+*1*, S)

else do {
  if b then do {
    ASSERT(*propagate-proper-bin-case L K S C*);
    if val-K = *Some False*
    then do {*RETURN* (*j*+*1*, *w*+*1*, *set-conflict-wl* (*get-clauses-wl S* $\propto$ *C*) *S*)}
    else do {
      let i = (*if* ((*get-clauses-wl S*)$\propto$*C*) ! *0* = *L then 0 else 1*);
      RETURN (*j*+*1*, *w*+*1*, *propagate-lit-wl-bin K C i S*)
    }
  } — Now the costly operations:
  else if C $\notin$# *dom-m* (*get-clauses-wl S*)
  then RETURN (*j*, *w*+*1*, *S*)
  else do {
    let i = (*if* ((*get-clauses-wl S*)$\propto$*C*) ! *0* = *L then 0 else 1*);
    let L' = ((*get-clauses-wl S*)$\propto$*C*) ! (*1* − *i*);
    let val-L' = *polarity* (*get-trail-wl S*) *L'*;
    if val-L' = *Some True*
    then update-blit-wl L C b j w L' S
    else do {
      f ← *find-unwatched-l* (*get-trail-wl S*) (*get-clauses-wl S* $\propto$*C*);
      ASSERT (*unit-prop-body-wl-D-find-unwatched-inv f C S*);
      case f of
        None ⇒ *do* {
          if val-L' = *Some False*
          then RETURN (*j*+*1*, *w*+*1*, *set-conflict-wl* (*get-clauses-wl S* $\propto$ *C*) *S*)
          else RETURN (*j*+*1*, *w*+*1*, *propagate-lit-wl L' C i S*)
        }
      | *Some f* ⇒ *do* {
          let K = *get-clauses-wl S* $\propto$ *C* ! *f*;
          let val-L' = *polarity* (*get-trail-wl S*) *K*;
          if val-L' = *Some True*
          then update-blit-wl L C b j w K S
          else update-clause-wl L C b j w i f S
        }
    }
  }
}
})›

declare *Id-refine*[*refine-vcg del*] *refine0*(*5*)[*refine-vcg del*]

lemma *unit-prop-body-wl-D-inv-clauses-distinct-eq*:
assumes
x[*simp*]: ‹*watched-by S K* ! *w* = (*x1*, *x2*)› **and**
inv: ‹*unit-prop-body-wl-D-inv* (*keep-watch K i w S*) *i w K*› **and**
y: ‹*y* < *length* (*get-clauses-wl S* $\propto$ (*fst* (*watched-by S K* ! *w*)))› **and**
w: ‹*fst*(*watched-by S K* ! *w*) ∈# *dom-m* (*get-clauses-wl* (*keep-watch K i w S*))› **and**
y': ‹*y'* < *length* (*get-clauses-wl S* $\propto$ (*fst* (*watched-by S K* ! *w*)))› **and**
w-le: ‹*w* < *length* (*watched-by S K*)›
shows ‹*get-clauses-wl S* $\propto$ *x1* ! *y* =
get-clauses-wl S $\propto$ *x1* ! *y'* ⟷ *y* = *y'*› (**is** ‹*?eq* ⟷ *?y*›)
proof
assume *eq*: *?eq*
let *?S* = ‹*keep-watch K i w S*›
let *?C* = ‹*fst* (*watched-by ?S K* ! *w*)›
have *dom*: ‹*fst* (*watched-by* (*keep-watch K i w S*) *K* ! *w*) ∈# *dom-m* (*get-clauses-wl* (*keep-watch K i*

w S))›
‹*fst* (*watched-by* (*keep-watch K i w S*) *K* ! *w*) ∈# *dom-m* (*get-clauses-wl S*)›
using *w-le assms* **by** (*auto simp*: *x twl-st-wl*)
obtain *T U* **where**
ST: ‹(*?S*, *T*) ∈ *state-wl-l* (*Some* (*K*, *w*))› **and**
TU: ‹(*set-clauses-to-update-l*
(*clauses-to-update-l*
(*remove-one-lit-from-wq ?C T*) +
{#*?C*#})
(*remove-one-lit-from-wq ?C T*),
U)
∈ *twl-st-l* (*Some K*)› **and**
struct-U: ‹*twl-struct-invs U*› **and**
i-w: ‹$i \leq w$› **and**
w-le: ‹$w <$ *length* (*watched-by* (*keep-watch K i w S*) *K*)›
using *inv w* **unfolding** *unit-prop-body-wl-D-inv-def unit-prop-body-wl-inv-def*
unit-prop-body-wl-inv-def unit-propagation-inner-loop-body-l-inv-def x fst-conv
apply −
apply (*simp only*: *simp-thms dom*)
apply *normalize-goal*+
by *blast*
have ‹$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state* ($state_W$*-of U*)›
using *struct-U* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
by *fast*
then have ‹*distinct-mset-mset* (*mset* '# *ran-mf* (*get-clauses-wl S*))›
using *ST TU*
unfolding *image-Un* $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def*
all-clss-lf-ran-m[*symmetric*] *image-mset-union*
by (*auto simp*: *drop-Suc twl-st-wl twl-st-l twl-st*)
then have ‹*distinct* (*get-clauses-wl S* ∝ *C*)› **if** ‹$C > 0$› **and** ‹*C* ∈# *dom-m* (*get-clauses-wl S*)›
for *C*
using *that ST TU* **unfolding** $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def*
distinct-mset-set-def
by (*auto simp*: *nth-in-set-tl mset-take-mset-drop-mset* $cdcl_W$*-restart-mset-state*
distinct-mset-set-distinct twl-st)
moreover have ‹$?C > 0$› **and** ‹*?C* ∈# *dom-m* (*get-clauses-wl S*)›
using *inv w* **unfolding** *unit-propagation-inner-loop-body-l-inv-def unit-prop-body-wl-D-inv-def*
unit-prop-body-wl-inv-def x **apply** −
apply (*simp only*: *simp-thms twl-st-wl x fst-conv dom*)
apply *normalize-goal*+
apply (*solves simp*)
apply (*simp only*: *simp-thms twl-st-wl x fst-conv dom*)
done
ultimately have ‹*distinct* (*get-clauses-wl S* ∝ *?C*)›
by *blast*
moreover have ‹*fst* (*watched-by* (*keep-watch K i w S*) *K* ! *w*) = *fst* (*watched-by S K* ! *w*)›
using *i-w w-le*
by (*cases S*; *cases* ‹*i*=*w*›) (*auto simp*: *keep-watch-def*)
ultimately show *?y*
using *y* y' *eq*
by (*auto simp*: *nth-eq-iff-index-eq twl-st-wl x*)
next
assume *?y*
then show *?eq* **by** *blast*
qed

lemma *in-all-lits-uminus-iff*[*simp*]: ‹$(- \ xa \in\# \ \textit{all-lits} \ N \ NUE) = (xa \in\# \ \textit{all-lits} \ N \ NUE)$›
  unfolding *all-lits-def*
  by (*auto simp*: *in-all-lits-of-mm-uminus-iff*)

lemma *is-$\mathcal{L}_{all}$-all-atms-st-all-lits-st*[*simp*]:
  ‹*is-$\mathcal{L}_{all}$* (*all-atms-st S*) (*all-lits-st S*)›
  unfolding *is-$\mathcal{L}_{all}$-def*
  by (*auto simp*: *in-$\mathcal{L}_{all}$-atm-of-in-atms-of-iff atms-of-$\mathcal{L}_{all}$-$\mathcal{A}_{in}$ atm-of-eq-atm-of*)

lemma *literals-are-$\mathcal{L}_{in}$-all-atms-st*:
  ‹*blits-in-$\mathcal{L}_{in}$ S* $\Longrightarrow$ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*›
  unfolding *literals-are-$\mathcal{L}_{in}$-def*
  by *auto*

lemma *blits-in-$\mathcal{L}_{in}$-keep-watch*:
  assumes ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *f*, *g*)› **and**
    w:‹$w <$ *length* (*watched-by* (*a*, *b*, *c*, *d*, *e*, *f*, *g*) *K*)›
  shows ‹*blits-in-$\mathcal{L}_{in}$* (*a*, *b*, *c*, *d*, *e*, *f*, *g* (K := (*g K*)[j := *g K* ! *w*]))›
proof −
  let *?S* = ‹(*a*, *b*, *c*, *d*, *e*, *f*, *g*)›
  let *?T* = ‹(*a*, *b*, *c*, *d*, *e*, *f*, *g* (K := (*g K*)[j := *g K* ! *w*]))›
  let *?g* = ‹*g* (K := (*g K*)[j := *g K* ! *w*])›
  have *H*: ‹$\bigwedge$*L i K b*. *L*$\in\#$ $\mathcal{L}_{all}$ (*all-atms-st ?S*) $\Longrightarrow$ (*i*, *K*, *b*) $\in$*set* (*g L*) $\Longrightarrow$
        K $\in\#$ $\mathcal{L}_{all}$ (*all-atms-st ?S*)›
    using *assms*
    unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
    by *blast*
  have ‹ *L*$\in\#\mathcal{L}_{all}$ (*all-atms-st ?S*) $\Longrightarrow$ (*i*, K', b') $\in$*set* (*?g L*) $\Longrightarrow$
        K' $\in\#$ $\mathcal{L}_{all}$ (*all-atms-st ?S*)› **for** *L i* K' b'
    using *H*[*of L i* K'] *H*[*of L* ‹*fst* (*g K* ! *w*)› ‹*fst* (*snd* (*g K* ! *w*))›]
      nth-mem[*OF w*]
    unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
    by (*cases* ‹$j <$ *length* (*g K*)›; *cases* ‹*g K* ! *w*›)
      (*auto split*: *if-splits elim*!: *in-set-upd-cases*)
  moreover have ‹*all-atms-st ?S* = *all-atms-st ?T*›
    by (*auto simp*: *all-lits-def all-atms-def*)
  ultimately show *?thesis*
    unfolding *blits-in-$\mathcal{L}_{in}$-def watched-by.simps*
    by *force*
qed

We mark as safe intro rule, since we will always be in a case where the equivalence holds, although in general the equivalence does not hold.

lemma *literals-are-$\mathcal{L}_{in}$-keep-watch*[*twl-st-wl*, *simp*, *intro*!]:
  ‹*literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ *S* $\Longrightarrow$ $w <$ *length* (*watched-by S K*) $\Longrightarrow$ *literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ (*keep-watch K j w S*)›
  by (*cases S*) (*auto simp*: *keep-watch-def literals-are-$\mathcal{L}_{in}$-def*
      blits-in-$\mathcal{L}_{in}$-keep-watch all-lits-def all-atms-def)

lemma *all-lits-update-swap*[*simp*]:
  ‹*x1* $\in\#$ *dom-m x1aa* $\Longrightarrow$ $n <$ *length* (*x1aa* $\propto$ *x1*) $\Longrightarrow$$n' <$ *length* (*x1aa* $\propto$ *x1*) $\Longrightarrow$
     all-lits (*x1aa*(*x1* $\hookrightarrow$ *swap* (*x1aa* $\propto$ *x1*) *n* n')) = *all-lits x1aa*›
  using *distinct-mset-dom*[*of x1aa*]
  unfolding *all-lits-def*
  by (*auto simp*: *ran-m-def if-distrib image-mset-If filter-mset-eq not-in-iff*[*THEN iffD1*]
      removeAll-mset-filter-mset[*symmetric*]
    dest!: *multi-member-split*[*of x1*]

intro!: *ext*)

lemma *blits-in-*$\mathcal{L}_{in}$*-propagate*:
‹$x1 \in\# \; dom\text{-}m \; x1aa \Longrightarrow n < length \; (x1aa \propto x1) \Longrightarrow n' < length \; (x1aa \propto x1) \Longrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (Propagated \; A \; x1' \; \# \; x1b, \; x1aa$
$(x1 \hookrightarrow swap \; (x1aa \propto x1) \; n \; n'), \; D, \; x1c, \; x1d,$
$add\text{-}mset \; A' \; x1e, \; x2e) \longleftrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1b, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)$›
‹$x1 \in\# \; dom\text{-}m \; x1aa \Longrightarrow n < length \; (x1aa \propto x1) \Longrightarrow n' < length \; (x1aa \propto x1) \Longrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1b, \; x1aa$
$(x1 \hookrightarrow swap \; (x1aa \propto x1) \; n \; n'), \; D, \; x1c, \; x1d, x1e, \; x2e) \longleftrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1b, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)$›
‹$blits\text{-}in\text{-}\mathcal{L}_{in}$
$(Propagated \; A \; x1' \; \# \; x1b, \; x1aa, \; D, \; x1c, \; x1d,$
$add\text{-}mset \; A' \; x1e, \; x2e) \longleftrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1b, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)$›
‹$x1' \in\# \; dom\text{-}m \; x1aa \Longrightarrow n < length \; (x1aa \propto x1') \Longrightarrow n' < length \; (x1aa \propto x1') \Longrightarrow$
$K \in\# \; \mathcal{L}_{all} \; (all\text{-}atms\text{-}st \; (x1b, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)) \Longrightarrow blits\text{-}in\text{-}\mathcal{L}_{in}$
$(x1a, \; x1aa(x1' \hookrightarrow swap \; (x1aa \propto x1') \; n \; n'), \; D, \; x1c, \; x1d,$
$x1e, \; x2e$
$(x1aa \propto x1' \; ! \; n' :=$
$x2e \; (x1aa \propto x1' \; ! \; n') \; @ \; [(x1', \; K, \; b')])) \longleftrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1a, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)$›
‹$K \in\# \; \mathcal{L}_{all} \; (all\text{-}atms\text{-}st \; (x1b, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)) \Longrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1a, \; x1aa, \; D, \; x1c, \; x1d,$
$x1e, \; x2e$
$(x1aa \propto x1' \; ! \; n' := x2e \; (x1aa \propto x1' \; ! \; n') \; @ \; [(x1', \; K, \; b')])) \longleftrightarrow$
$blits\text{-}in\text{-}\mathcal{L}_{in} \; (x1a, \; x1aa, \; D, \; x1c, \; x1d, \; x1e, \; x2e)$›
unfolding *blits-in-*$\mathcal{L}_{in}$*-def*
by (*auto split*: *if-splits*)[5]

lemma *literals-are-*$\mathcal{L}_{in}$*-set-conflict-wl*:
‹$literals\text{-}are\text{-}\mathcal{L}_{in} \; \mathcal{A} \; (set\text{-}conflict\text{-}wl \; D \; S) \longleftrightarrow literals\text{-}are\text{-}\mathcal{L}_{in} \; \mathcal{A} \; S$›
by (*cases S*; *auto simp*: *blits-in-*$\mathcal{L}_{in}$*-def literals-are-*$\mathcal{L}_{in}$*-def set-conflict-wl-def*)

lemma *blits-in-*$\mathcal{L}_{in}$*-keep-watch'*:
assumes K': ‹$K' \in\# \; \mathcal{L}_{all} \; (all\text{-}atms\text{-}st \; (a, \; b, \; c, \; d, \; e, \; f, \; g))$› **and**
w:‹$blits\text{-}in\text{-}\mathcal{L}_{in} \; (a, \; b, \; c, \; d, \; e, \; f, \; g)$›
shows ‹$blits\text{-}in\text{-}\mathcal{L}_{in} \; (a, \; b, \; c, \; d, \; e, \; f, \; g \; (K := (g \; K)[j := (i, \; K', \; b')]))$›
proof −
let $?\mathcal{A}$ = ‹$all\text{-}atms\text{-}st \; (a, \; b, \; c, \; d, \; e, \; f, \; g)$›
let $?g$ = ‹$g \; (K := (g \; K)[j := (i, \; K', \; b')])$›
have H: ‹$\bigwedge L \; i \; K \; b'. \; L \in\# \mathcal{L}_{all} \; ?\mathcal{A} \Longrightarrow (i, \; K, \; b') \in set \; (g \; L) \Longrightarrow K \in\# \; \mathcal{L}_{all} \; ?\mathcal{A}$›
using *assms*
unfolding *blits-in-*$\mathcal{L}_{in}$*-def watched-by.simps*
by *blast*
have ‹ $L \in\# \mathcal{L}_{all} \; ?\mathcal{A} \Longrightarrow (i, \; K', \; b') \in set \; (?g \; L) \Longrightarrow K' \in\# \; \mathcal{L}_{all} \; ?\mathcal{A}$› **for** $L \; i \; K' \; b'$
using $H[of \; L \; i \; K'] \; K'$
unfolding *blits-in-*$\mathcal{L}_{in}$*-def watched-by.simps*
by (*cases* ‹$j < length \; (g \; K)$›; *cases* ‹$g \; K \; ! \; w$›)
(*auto split*: *if-splits elim*!: *in-set-upd-cases*)

then show *?thesis*
unfolding *blits-in-*$\mathcal{L}_{in}$*-def watched-by.simps*
by *force*
qed

lemma *literals-are-$\mathcal{L}_{in}$-all-atms-stD*[*dest*]:
‹*literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ *S* $\Longrightarrow$ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*›
unfolding *literals-are-$\mathcal{L}_{in}$-def*
by *auto*

lemma *blits-in-$\mathcal{L}_{in}$-set-conflict*[*simp*]: ‹*blits-in-$\mathcal{L}_{in}$* (*set-conflict-wl D S*) = *blits-in-$\mathcal{L}_{in}$ S*›
by (*cases S*) (*auto simp*: *blits-in-$\mathcal{L}_{in}$-def set-conflict-wl-def*)

lemma *unit-propagation-inner-loop-body-wl-D-spec*:
fixes *S* :: ‹*nat twl-st-wl*› **and** *K* :: ‹*nat literal*› **and** *w* :: *nat*
assumes
K: ‹$K \in\# \mathcal{L}_{all}\ \mathcal{A}$› **and**
$\mathcal{A}_{in}$: ‹*literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ *S*›
shows ‹*unit-propagation-inner-loop-body-wl-D K j w S* $\leq$
$\Downarrow$ {((*j'*, *n'*, *T'*), (*j*, *n*, *T*)). $j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ *T'*}
(*unit-propagation-inner-loop-body-wl K j w S*)›
proof –
obtain *M N D NE UE Q W* **where**
S: ‹*S* = (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›
by (*cases S*)
have *f'*: ‹(*f*, *f'*) ∈ ⟨*Id*⟩*option-rel*› **if** ‹(*f*, *f'*) ∈ *Id*› **for** *f f'*
using *that* **by** *auto*
define *find-unwatched-wl* :: ‹(*nat*,*nat*) *ann-lits* ⇒ -› **where**
‹*find-unwatched-wl* = *find-unwatched-l*›
let *?C* = ‹*fst* ((*watched-by S K*) ! *w*)›
have *find-unwatched*: ‹*find-unwatched-wl* (*get-trail-wl S*) ((*get-clauses-wl S*)$\propto$*D*)
$\leq \Downarrow$ {(*L*, *L'*). $L = L' \wedge (L \neq None \longrightarrow$ *the L* < *length* ((*get-clauses-wl S*)$\propto$*C*) $\wedge$ *the L* $\geq$ *2*)}
(*find-unwatched-l* (*get-trail-wl S*) ((*get-clauses-wl S*)$\propto$*C*))›
(**is** ‹- $\leq \Downarrow$ *?find-unwatched* -›)
if ‹*C* = *D*›
for *C D* **and** *L* **and** *K* **and** *S*
unfolding *find-unwatched-l-def find-unwatched-wl-def that*
by (*auto simp*: *intro*!: *RES-refine*)

have *propagate-lit-wl*:
‹((*j*+*1*, *w* + *1*,
propagate-lit-wl
(*get-clauses-wl S* $\propto$ *x1a* ! (*1* − (*if get-clauses-wl S* $\propto$ *x1a* ! *0* = *K then 0 else 1*)))
x1a
(*if get-clauses-wl S* $\propto$ *x1a* ! *0* = *K then 0 else 1*)
S),
j+*1*, *w* + *1*,
propagate-lit-wl
(*get-clauses-wl S* $\propto$ *x1* !
(*1* − (*if get-clauses-wl S* $\propto$ *x1* ! *0* = *K then 0*
else 1)))
x1
(*if get-clauses-wl S* $\propto$ *x1* ! *0* = *K then 0 else 1*) *S*)
∈ {((*j'*, *n'*, *T'*), *j*, *n*, *T*).
$j' = j \wedge$
$n' = n \wedge$
$T = T' \wedge$
literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ *T'*}›
if ‹*unit-prop-body-wl-D-inv S j w K*› **and** ‹¬*x1* ∉# *dom-m* (*get-clauses-wl S*)› **and**
‹(*watched-by S K*) ! *w* = (*x1a*, *x2a*)› **and**

⟨(*watched-by S K*) ! w = (*x1*, *x2*)⟩ **and**
⟨*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*⟩
for $f\ f'\ j\ S$ *x1 x2 x1a x2a*
unfolding *propagate-lit-wl-def S*
apply *clarify*
apply *refine-vcg*
using *that* $\mathcal{A}_{in}$
by (*auto simp*: *clauses-def unit-prop-body-wl-find-unwatched-inv-def*
mset-take-mset-drop-mset′ S unit-prop-body-wl-D-inv-def unit-prop-body-wl-inv-def
ran-m-mapsto-upd unit-propagation-inner-loop-body-l-inv-def blits-in-$\mathcal{L}_{in}$*-propagate*
state-wl-l-def image-mset-remove1-mset-if literals-are-$\mathcal{L}_{in}$*-def*)

have *update-clause-wl*: ⟨*update-clause-wl K x1′ b′ j w*
(*if get-clauses-wl S* ∝ *x1′* ! *0* = *K then 0 else 1*) *n S*
$\le \Downarrow \{((j', n', T'), j, n, T).\ j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T'\}$
(*update-clause-wl K x1 b j w*
(*if get-clauses-wl S* ∝ *x1* ! *0* = *K then 0 else 1*) n' *S*)⟩
if ⟨$(n, n') \in Id$⟩ **and** ⟨*unit-prop-body-wl-D-inv S j w K*⟩
⟨$(f, f') \in$ *?find-unwatched x1 S*⟩ **and**
⟨f = *Some n*⟩ ⟨f' = *Some n′*⟩ **and**
⟨*unit-prop-body-wl-D-find-unwatched-inv f x1′ S*⟩ **and**
⟨¬*x1* ∉# *dom-m* (*get-clauses-wl S*)⟩ **and**
⟨*watched-by S K* ! w = (*x1*, *x2*)⟩ **and**
⟨*watched-by S K* ! w = (*x1′*, *x2′*)⟩ **and**
⟨$(b, b') \in Id$⟩ **and**
⟨*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*⟩
for $n\ n'\ f\ f'$ *S x1 x2 x1′ x2′ b b′*
unfolding *update-clause-wl-def S*
apply *refine-vcg*
using *that* $\mathcal{A}_{in}$
by (*auto simp*: *clauses-def mset-take-mset-drop-mset unit-prop-body-wl-find-unwatched-inv-def*
mset-take-mset-drop-mset′ S unit-prop-body-wl-D-inv-def unit-prop-body-wl-inv-def
ran-m-clause-upd unit-propagation-inner-loop-body-l-inv-def blits-in-$\mathcal{L}_{in}$*-propagate*
state-wl-l-def image-mset-remove1-mset-if literals-are-$\mathcal{L}_{in}$*-def*)
have *H*: ⟨*watched-by S K* ! $w = A \Longrightarrow$ *watched-by* (*keep-watch K j w S*) *K* ! $w = A$⟩
for *S j w K A x1*
by (*cases S*; *cases* ⟨*j*=*w*⟩) (*auto simp*: *keep-watch-def*)
have *update-blit-wl*: ⟨*update-blit-wl K x1a b′ j w*
(*get-clauses-wl* (*keep-watch K j w S*) ∝ *x1a* !
(*1* − (*if get-clauses-wl* (*keep-watch K j w S*) ∝ *x1a* ! *0* = *K then 0 else 1*)))
(*keep-watch K j w S*)
$\le \Downarrow \{((j', n', T'), j, n, T).$
$j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T'\}$
(*update-blit-wl K x1 b j w*
(*get-clauses-wl* (*keep-watch K j w S*) ∝ *x1* !
(*1* −
(*if get-clauses-wl* (*keep-watch K j w S*) ∝ *x1* ! *0* = *K then 0*
else 1)))
(*keep-watch K j w S*))⟩
if
x: ⟨*watched-by S K* ! w = (*x1*, *x2*)⟩ **and**
xa: ⟨*watched-by S K* ! w = (*x1a*, *x2a*)⟩ **and**
unit: ⟨*unit-prop-body-wl-D-inv* (*keep-watch K j w S*) *j w K*⟩ **and**
x1: ⟨¬*x1* ∉# *dom-m* (*get-clauses-wl* (*keep-watch K j w S*))⟩ **and**
bb′: ⟨$(b, b') \in Id$⟩
for *x1 x2 x1a x2a b b′*

proof –
have [*simp*]: ‹*x1a* = *x1*› **and** *x1a*: ‹*x1* ∈# *dom-m* (*get-clauses-wl S*)›
‹*fst* (*watched-by* (*keep-watch K j w S*) *K* ! *w*) ∈# *dom-m* (*get-clauses-wl* (*keep-watch K j w S*))›
using *x xa x1 unit* **unfolding** *unit-prop-body-wl-D-inv-def unit-prop-body-wl-inv-def*
by *auto*

have ‹*get-clauses-wl S* ∝*x1* ! *0* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$ ∧ *get-clauses-wl S* ∝ *x1* ! *Suc 0* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
using *assms that*
literals-are-in-$\mathcal{L}_{in}$*-nth*[*of x1 S*]
literals-are-in-$\mathcal{L}_{in}$*-in-*$\mathcal{L}_{all}$[*of* $\mathcal{A}$ ‹*get-clauses-wl S* ∝ *x1*› *0*]
literals-are-in-$\mathcal{L}_{in}$*-in-*$\mathcal{L}_{all}$[*of* $\mathcal{A}$ ‹*get-clauses-wl S* ∝ *x1*› *1*]
unfolding *unit-prop-body-wl-D-inv-def unit-prop-body-wl-inv-def*
unit-propagation-inner-loop-body-l-inv-def x1a **apply** (*simp only*: *x1a fst-conv simp-thms*)
apply *normalize-goal*+
by (*auto simp del*: *simp*: *x1a*)
then show *?thesis*
using *assms unit bb′*
by (*cases S*)
(*auto simp*: *keep-watch-def update-blit-wl-def literals-are-*$\mathcal{L}_{in}$*-def*
blits-in-$\mathcal{L}_{in}$*-propagate blits-in-*$\mathcal{L}_{in}$*-keep-watch′ unit-prop-body-wl-D-inv-def*)
qed
have *update-blit-wl′*: ‹*update-blit-wl K x1a b′ j w* (*get-clauses-wl* (*keep-watch K j w S*) ∝ *x1a* ! *x*)
(*keep-watch K j w S*)
≤ ⇓ {((*j′*, *n′*, *T′*), *j*, *n*, *T*).
j′ = *j* ∧ *n′* = *n* ∧ *T* = *T′* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T′*}
(*update-blit-wl K x1 b j w*
(*get-clauses-wl* (*keep-watch K j w S*) ∝ *x1* ! *x′*)
(*keep-watch K j w S*))›
if
x1: ‹*watched-by S K* ! *w* = (*x1*, *x2*)› **and**
xa: ‹*watched-by S K* ! *w* = (*x1a*, *x2a*)› **and**
unw: ‹*unit-prop-body-wl-D-find-unwatched-inv f x1a* (*keep-watch K j w S*)› **and**
dom: ‹¬*x1* ∉# *dom-m*(*get-clauses-wl* (*keep-watch K j w S*))› **and**
unit: ‹*unit-prop-body-wl-D-inv* (*keep-watch K j w S*) *j w K*› **and**
f: ‹*f* = *Some x*› **and**
xx′: ‹(*x*, *x′*) ∈ *nat-rel*› **and**
bb′: ‹(*b*, *b′*) ∈ *Id*›
for *x1 x2 x1a x2a f fa x x′ b b′*
proof –
have [*simp*]: ‹*x1a* = *x1*› ‹*x* = *x′*›
using *x1 xa xx′* **by** *auto*

have *x1a*: ‹*x1* ∈# *dom-m* (*get-clauses-wl S*)›
‹*fst* (*watched-by S K* ! *w*) ∈# *dom-m* (*get-clauses-wl S*)›
using *dom x1* **by** *auto*
have ‹*get-clauses-wl S* ∝*x1* ! *x* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
using *assms that*
literals-are-in-$\mathcal{L}_{in}$*-nth*[*of x1 S*]
literals-are-in-$\mathcal{L}_{in}$*-in-*$\mathcal{L}_{all}$[*of* $\mathcal{A}$ ‹*get-clauses-wl S* ∝ *x1*› *x*]
unw
unfolding *unit-prop-body-wl-D-find-unwatched-inv-def*
by *auto*
then show *?thesis*
using *assms bb′*
by (*cases S*) (*auto simp*: *keep-watch-def update-blit-wl-def literals-are-*$\mathcal{L}_{in}$*-def*
blits-in-$\mathcal{L}_{in}$*-propagate blits-in-*$\mathcal{L}_{in}$*-keep-watch′*)

qed

have *set-conflict-rel*:
‹((j + 1, w + 1,
set-conflict-wl (*get-clauses-wl* (*keep-watch K j w S*) $\propto$ *x1a*) (*keep-watch K j w S*)),
j + 1, w + 1,
set-conflict-wl (*get-clauses-wl* (*keep-watch K j w S*) $\propto$ *x1*) (*keep-watch K j w S*))
$\in$ {((j', n', T'), j, n, T). $j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ T'}›
if
pre: ‹*unit-propagation-inner-loop-wl-loop-D-pre K* (*j*, *w*, *S*)› **and**
x: ‹*watched-by S K* ! *w* = (*x1*, *x2*)› **and**
xa: ‹*watched-by S K* ! *w* = (*x1a*, *x2a'*)› **and**
xa': ‹*x2a'* = (*x2a*, *x3*)› **and**
unit: ‹*unit-prop-body-wl-D-inv* (*keep-watch K j w S*) *j w K*› **and**
dom: ‹$\neg$ *x1a* $\notin$# *dom-m* (*get-clauses-wl* (*keep-watch K j w S*))›
for *x1 x2 x1a x2a f fa x2a' x3*
proof −
have [*simp*]: ‹*blits-in-$\mathcal{L}_{in}$*
(*set-conflict-wl D* (*a*, *b*, *c*, *d*, *e*, *fb*, *g*(*K* := (*g K*)[*j* := *de*]))) $\longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ ((*a*, *b*, *c*, *d*, *e*, *fb*, *g*(*K* := (*g K*)[*j* := *de*])))›
for *a b c d e f g de D*
by (*auto simp*: *blits-in-$\mathcal{L}_{in}$-def set-conflict-wl-def*)

have [*simp*]: ‹*x1a* = *x1*›
using *xa x* **by** *auto*

have ‹*x2a* $\in$# $\mathcal{L}_{all}$ $\mathcal{A}$›
using *xa x dom assms pre unit nth-mem*[*of w* ‹*watched-by S K*›] *xa'*
by (*cases S*)
(*auto simp*: *unit-prop-body-wl-D-inv-def literals-are-$\mathcal{L}_{in}$-def*
unit-prop-body-wl-inv-def blits-in-$\mathcal{L}_{in}$-def keep-watch-def
unit-propagation-inner-loop-wl-loop-D-pre-def
dest!: *multi-member-split split*: *if-splits*)
then show *?thesis*
using *assms that* **by** (*cases S*) (*auto simp*: *keep-watch-def literals-are-$\mathcal{L}_{in}$-set-conflict-wl*
literals-are-$\mathcal{L}_{in}$-def blits-in-$\mathcal{L}_{in}$-keep-watch')
qed
have *bin-set-conflict*:
‹((j + 1, w + 1, *set-conflict-wl* (*get-clauses-wl* (*keep-watch K j w S*) $\propto$ *x1b*) (*keep-watch K j w S*)),
j + 1, w + 1,
set-conflict-wl (*get-clauses-wl* (*keep-watch K j w S*) $\propto$ *x1*) (*keep-watch K j w S*))
$\in$ {((j', n', T'), j, n, T). $j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ T'}›
if
‹*unit-propagation-inner-loop-wl-loop-pre K* (*j*, *w*, *S*)› **and**
‹*unit-propagation-inner-loop-wl-loop-D-pre K* (*j*, *w*, *S*)› **and**
‹*x2* = (*x1a*, *x2a*)› **and**
‹*watched-by S K* ! *w* = (*x1*, *x2*)› **and**
‹*x2b* = (*x1c*, *x2c*)› **and**
‹*watched-by S K* ! *w* = (*x1b*, *x2b*)› **and**
‹*unit-prop-body-wl-inv* (*keep-watch K j w S*) *j w K*› **and**
‹*unit-prop-body-wl-D-inv* (*keep-watch K j w S*) *j w K*› **and**
‹*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1c* $\neq$ *Some True*› **and**
‹*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1a* $\neq$ *Some True*› **and**
‹*x2c*› **and**
‹*x2a*› **and**
‹*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1c* = *Some False*› **and**

⟨*polarity* (*get-trail-wl* (*keep-watch* *K* *j* *w* *S*)) *x1a* = *Some False*⟩
for *x1 x2 x1a x2a x1b x2b x1c x2c*
proof −
show *?thesis*
using *that assms*
by (*auto simp*: *literals-are-$\mathcal{L}_{in}$-set-conflict-wl unit-propagation-inner-loop-wl-loop-pre-def*)
qed
have *bin-prop*:
⟨((*j* + *1*, *w* + *1*,
propagate-lit-wl-bin x1c x1b (*if get-clauses-wl* (*keep-watch K j w S*) ∝ *x1b* ! *0* = *K then 0 else 1*)
(*keep-watch K j w S*)),
j + *1*, *w* + *1*,
propagate-lit-wl-bin x1a x1 (*if get-clauses-wl* (*keep-watch K j w S*) ∝ *x1* ! *0* = *K then 0 else 1*)
(*keep-watch K j w S*))
∈ {((*j'*, *n'*, *T'*), *j*, *n*, *T*). *j'* = *j* ∧ *n'* = *n* ∧ *T* = *T'* ∧ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ T'*}⟩
if
⟨*unit-propagation-inner-loop-wl-loop-pre K* (*j*, *w*, *S*)⟩ **and**
⟨*unit-propagation-inner-loop-wl-loop-D-pre K* (*j*, *w*, *S*)⟩ **and**
⟨*x2* = (*x1a*, *x2a*)⟩ **and**
⟨*watched-by S K* ! *w* = (*x1*, *x2*)⟩ **and**
⟨*x2b* = (*x1c*, *x2c*)⟩ **and**
⟨*watched-by S K* ! *w* = (*x1b*, *x2b*)⟩ **and**
⟨*unit-prop-body-wl-inv* (*keep-watch K j w S*) *j w K*⟩ **and**
⟨*unit-prop-body-wl-D-inv* (*keep-watch K j w S*) *j w K*⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1c* ≠ *Some True*⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1a* ≠ *Some True*⟩ **and**
⟨*x2c*⟩ **and**
⟨*x2a*⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1c* ≠ *Some False*⟩ **and**
⟨*polarity* (*get-trail-wl* (*keep-watch K j w S*)) *x1a* ≠ *Some False*⟩ **and**
⟨*propagate-proper-bin-case K x1a* (*keep-watch K j w S*) *x1*⟩
for *x1 x2 x1a x2a x1b x2b x1c x2c*
unfolding *propagate-lit-wl-bin-def S propagate-proper-bin-case-def*
apply *clarify*
apply *refine-vcg*
using *that* $\mathcal{A}_{in}$
by (*simp-all add*: *unit-prop-body-wl-find-unwatched-inv-def*
propagate-proper-bin-case-def unit-prop-body-wl-inv-def
S unit-prop-body-wl-D-inv-def keep-watch-def state-wl-l-def literals-are-$\mathcal{L}_{in}$-def
Let-def blits-in-$\mathcal{L}_{in}$-propagate)
show *?thesis*
unfolding *unit-propagation-inner-loop-body-wl-D-def find-unwatched-wl-def*[*symmetric*]
unfolding *unit-propagation-inner-loop-body-wl-def*
supply [[*goals-limit=1*]]
apply (*refine-rcg find-unwatched f'*)
subgoal using *assms* **unfolding** *unit-propagation-inner-loop-wl-loop-D-inv-def*
unit-propagation-inner-loop-wl-loop-D-pre-def unit-propagation-inner-loop-wl-loop-pre-def
by *auto*
subgoal using *assms* **unfolding** *unit-prop-body-wl-D-inv-def*
unit-propagation-inner-loop-wl-loop-pre-def **by** *auto*
subgoal by *simp*
subgoal using *assms* **by** (*auto simp*: *unit-propagation-inner-loop-wl-loop-pre-def*)
subgoal by *simp*
subgoal
using *assms* **by** (*auto simp*: *unit-prop-body-wl-D-inv-clauses-distinct-eq*
unit-propagation-inner-loop-wl-loop-pre-def)

subgoal by *auto*
subgoal
by (*rule bin-set-conflict*)
subgoal for *x1 x2 x1a x2a x1b x2b x1c x2c*
by (*rule bin-prop*)
subgoal by *simp*
subgoal
using *assms* **by** (*auto simp*: *unit-prop-body-wl-D-inv-clauses-distinct-eq unit-propagation-inner-loop-wl-loop-pre-def*)
subgoal by *simp*
subgoal by (*rule update-blit-wl*) *auto*
subgoal by *simp*
subgoal
using *assms*
unfolding *unit-prop-body-wl-D-find-unwatched-inv-def unit-prop-body-wl-inv-def*
by (*cases* ‹*watched-by S K* ! *w*›)
(*auto simp*: *unit-prop-body-wl-D-inv-clauses-distinct-eq twl-st-wl*)
subgoal by (*auto simp*: *twl-st-wl*)
subgoal by (*auto simp*: *twl-st-wl*)
subgoal for *x1 x2 x1a x2a f fa*
by (*rule set-conflict-rel*)
subgoal
by (*rule propagate-lit-wl*[*OF - - H H*]; *assumption?*)
(*simp add*: *assms literals-are-*$\mathcal{L}_{in}$*-keep-watch assms unit-propagation-inner-loop-wl-loop-pre-def*)
subgoal by (*auto simp*: *twl-st-wl*)
subgoal by (*rule update-blit-wl'*) *auto*
subgoal by (*rule update-clause-wl*[*OF - - - - - - - H H*]; *assumption?*) (*auto simp*: *assms unit-propagation-inner-loop-wl-loop-pre-def*)
done
qed

lemma *unit-propagation-inner-loop-body-wl-D-unit-propagation-inner-loop-body-wl-D*:
‹(*uncurry3 unit-propagation-inner-loop-body-wl-D*, *uncurry3 unit-propagation-inner-loop-body-wl*) ∈
$[\lambda(((K, j), w), S).\ literals\text{-}are\text{-}\mathcal{L}_{in}\ \mathcal{A}\ S \wedge K \in\#\ \mathcal{L}_{all}\ \mathcal{A}]_f$
$Id \times_r Id \times_r Id \times_r Id \rightarrow \langle nat\text{-}rel \times_r nat\text{-}rel \times_r$
$\{(T', T).\ T = T' \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ \mathcal{A}\ T\}\rangle\ nres\text{-}rel$›
(**is** ‹*?G1*›) **and**
unit-propagation-inner-loop-body-wl-D-unit-propagation-inner-loop-body-wl-D-weak:
‹(*uncurry3 unit-propagation-inner-loop-body-wl-D*, *uncurry3 unit-propagation-inner-loop-body-wl*) ∈
$[\lambda(((K, j), w), S).\ literals\text{-}are\text{-}\mathcal{L}_{in}\ \mathcal{A}\ S \wedge K \in\#\ \mathcal{L}_{all}\ \mathcal{A}]_f$
$Id \times_r Id \times_r Id \times_r Id \rightarrow \langle nat\text{-}rel \times_r nat\text{-}rel \times_r Id\rangle\ nres\text{-}rel$›
(**is** ‹*?G2*›)
proof −
have *1*: ‹$nat\text{-}rel \times_r nat\text{-}rel \times_r \{(T', T).\ T = T' \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ \mathcal{A}\ T\} =$
$\{((j', n', T'), (j, (n, T))).\ j' = j \wedge n' = n \wedge T = T' \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ \mathcal{A}\ T'\}$›
by *auto*
show *?G1*
by (*auto simp add*: *fref-def nres-rel-def uncurry-def simp del*: *twl-st-of-wl.simps intro*!: *unit-propagation-inner-loop-body-wl-D-spec*[*of* - $\mathcal{A}$, *unfolded 1*[*symmetric*]])

then show *?G2*
apply −
apply (*match-spec*)
apply (*match-fun-rel*; *match-fun-rel?*)

by *fastforce+*
qed

definition *unit-propagation-inner-loop-wl-loop-D*
:: ‹*nat literal* ⇒ *nat twl-st-wl* ⇒ (*nat* × *nat* × *nat twl-st-wl*) *nres*›
where
‹*unit-propagation-inner-loop-wl-loop-D L* S_0 = *do* {
ASSERT($L \in\# \mathcal{L}_{all}$ (*all-atms-st* S_0));
let n = *length* (*watched-by* S_0 *L*);
$WHILE_T$ $^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}wl\text{-}loop\text{-}D\text{-}inv\ L}$
(λ(*j*, *w*, *S*). *w* < *n* ∧ *get-conflict-wl S* = *None*)
(λ(*j*, *w*, *S*). *do* {
unit-propagation-inner-loop-body-wl-D L j w S
})
(*0*, *0*, S_0)
}
›

lemma *unit-propagation-inner-loop-wl-spec*:
assumes $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› **and** *K*: ‹$K \in\# \mathcal{L}_{all}$ $\mathcal{A}$›
shows ‹*unit-propagation-inner-loop-wl-loop-D K S* ≤
⇓ {((*j'*, *n'*, *T'*), *j*, *n*, *T*). *j'* = *j* ∧ *n'* = *n* ∧ *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T'*}
(*unit-propagation-inner-loop-wl-loop K S*)›
proof −
have *u*: ‹*unit-propagation-inner-loop-body-wl-D K j w S* ≤
⇓ {((*j'*, *n'*, *T'*), *j*, *n*, *T*). *j'* = *j* ∧ *n'* = *n* ∧ *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T'*}
(*unit-propagation-inner-loop-body-wl K' j' w' S'*)›
if ‹$K \in\# \mathcal{L}_{all}$ $\mathcal{A}$› **and** ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› **and**
‹*S* = *S'*› ‹*K* = *K'*› ‹*w* = *w'*› ‹*j'*=*j*›
for *S S'* **and** *w w'* **and** *K K'* **and** *j' j*
using *unit-propagation-inner-loop-body-wl-D-spec*[*of K* $\mathcal{A}$ *S j w*] *that* **by** *auto*

show *?thesis*
unfolding *unit-propagation-inner-loop-wl-loop-D-def unit-propagation-inner-loop-wl-loop-def*
apply (*refine-vcg u*)
subgoal using *assms* **by** *auto*
subgoal using *assms* **by** *auto*
subgoal using *assms* **unfolding** *unit-propagation-inner-loop-wl-loop-D-inv-def*
by (*auto dest*: *literals-are-*$\mathcal{L}_{in}$*-set-mset-*$\mathcal{L}_{all}$)
subgoal by *auto*
subgoal using *K* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *unit-propagation-inner-loop-wl-D*
:: ‹*nat literal* ⇒ *nat twl-st-wl* ⇒ *nat twl-st-wl nres*› **where**
‹*unit-propagation-inner-loop-wl-D L* S_0 = *do* {
(*j*, *w*, *S*) ← *unit-propagation-inner-loop-wl-loop-D L* S_0;
ASSERT (*j* ≤ *w* ∧ *w* ≤ *length* (*watched-by S L*) ∧ $L \in\# \mathcal{L}_{all}$ (*all-atms-st* S_0) ∧
$L \in\# \mathcal{L}_{all}$ (*all-atms-st S*));
S ← *cut-watch-list j w L S*;

RETURN S
}›

lemma *unit-propagation-inner-loop-wl-D-spec*:
assumes $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› **and** *K*: ‹$K \in\# \mathcal{L}_{all}\ \mathcal{A}$›
shows ‹*unit-propagation-inner-loop-wl-D K S* ≤
$\Downarrow \{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T\}$
(*unit-propagation-inner-loop-wl K S*)›
proof −
have *cut-watch-list*: ‹*cut-watch-list x1b x1c K x2c* ⪢= *RETURN*
$\leq \Downarrow \{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T\}$
(*cut-watch-list x1 x1a K x2a*)›
if
‹(x, x')
$\in \{((j', n', T'), j, n, T).$
$j' = j \wedge n' = n \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T'\}$› **and**
‹*x2* = (*x1a*, *x2a*)› **and**
‹*x'* = (*x1*, *x2*)› **and**
‹*x2b* = (*x1c*, *x2c*)› **and**
‹*x* = (*x1b*, *x2b*)› **and**
‹*x1* ≤ *x1a* ∧ *x1a* ≤ *length* (*watched-by x2a K*)›
for *x x' x1 x2 x1a x2a x1b x2b x1c x2c*
proof −
show *?thesis*
using *that* **unfolding** *literals-are-*$\mathcal{L}_{in}$*-def*
by (*cases x2c*) (*auto simp*: *cut-watch-list-def*
blits-in-$\mathcal{L}_{in}$*-def dest*!: *in-set-takeD in-set-dropD*)
qed

show *?thesis*
unfolding *unit-propagation-inner-loop-wl-D-def unit-propagation-inner-loop-wl-def*
apply (*refine-vcg unit-propagation-inner-loop-wl-spec*[*of* $\mathcal{A}$])
subgoal using $\mathcal{A}_{in}$ **.**
subgoal using *K* **.**
subgoal by *auto*
subgoal by *auto*
subgoal using $\mathcal{A}_{in}$ *K* **by** *auto*
subgoal using $\mathcal{A}_{in}$ *K* **by** *auto*
subgoal by (*rule cut-watch-list*)
done
qed

definition *unit-propagation-outer-loop-wl-D-inv* **where**
‹*unit-propagation-outer-loop-wl-D-inv S* ⟷
unit-propagation-outer-loop-wl-inv S ∧
literals-are-$\mathcal{L}_{in}$ (*all-atms-st S*) *S*›

definition *unit-propagation-outer-loop-wl-D*
:: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*›
where
‹*unit-propagation-outer-loop-wl-D* S_0 =
$WHILE_T{}^{unit\text{-}propagation\text{-}outer\text{-}loop\text{-}wl\text{-}D\text{-}inv}$
(λ*S*. *literals-to-update-wl S* ≠ {#})
(λ*S*. *do* {
ASSERT(*literals-to-update-wl S* ≠ {#});
(*S'*, *L*) ← *select-and-remove-from-literals-to-update-wl S*;

$ASSERT(L \in\# \mathcal{L}_{all}$ (*all-atms-st S*));
unit-propagation-inner-loop-wl-D L S'
})
(S_0 :: *nat twl-st-wl*)›

lemma *literals-are-*$\mathcal{L}_{in}$*-set-lits-to-upd*[*twl-st-wl*, *simp*]:
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ (*set-literals-to-update-wl C S*) $\longleftrightarrow$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
by (*cases S*) (*auto simp*: *literals-are-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*-def*)

lemma *unit-propagation-outer-loop-wl-D-spec*:
assumes $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
shows ‹*unit-propagation-outer-loop-wl-D S* $\leq$
$\Downarrow$ {(T', T). $T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}
(*unit-propagation-outer-loop-wl S*)›
proof –
have *H*: ‹*set-mset* (*all-lits-of-mm* (*mset* '# *ran-mf* (*get-clauses-wl S'*) + *get-unit-clauses-wl S'*)) =
set-mset ($\mathcal{L}_{all}$ (*all-atms-st S'*))› **for** *S'*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff all-atms-def all-lits-def*
in-$\mathcal{L}_{all}$*-atm-of-*$\mathcal{A}_{in}$)
have *select*: ‹*select-and-remove-from-literals-to-update-wl S* $\leq$
$\Downarrow$ {((T', L'), (T, L)). $T = T' \wedge L = L' \wedge$
T = *set-literals-to-update-wl* (*literals-to-update-wl S* − {#*L*#}) *S*}
(*select-and-remove-from-literals-to-update-wl S'*)›
if ‹$S = S'$› **for** *S S'* :: ‹*nat twl-st-wl*›
unfolding *select-and-remove-from-literals-to-update-wl-def select-and-remove-from-literals-to-update-def*
apply (*rule RES-refine*)
using *that* **unfolding** *select-and-remove-from-literals-to-update-wl-def* **by** *blast*
have *unit-prop*: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $S \Longrightarrow$
$K \in\# \mathcal{L}_{all}\ \mathcal{A} \Longrightarrow$
unit-propagation-inner-loop-wl-D K S
$\leq \Downarrow$ {(T', T). $T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T} (*unit-propagation-inner-loop-wl K' S'*)›
if ‹$K = K'$› **and** ‹$S = S'$› **for** *K K'* **and** *S S'* :: ‹*nat twl-st-wl*›
unfolding *that* **by** (*rule unit-propagation-inner-loop-wl-D-spec*)
show *?thesis*
unfolding *unit-propagation-outer-loop-wl-D-def unit-propagation-outer-loop-wl-def H*
apply (*refine-vcg select unit-prop*)
subgoal using $\mathcal{A}_{in}$ **by** *simp*
subgoal unfolding *unit-propagation-outer-loop-wl-D-inv-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using $\mathcal{A}_{in}$ **apply** *simp* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using $\mathcal{A}_{in}$ **by** (*auto simp*: *twl-st-wl*)
subgoal for *S' S T'L' TL T' L' T L*
using $\mathcal{A}_{in}$ **by** *auto*
done
qed

lemma *unit-propagation-outer-loop-wl-D-spec'*:
shows ‹(*unit-propagation-outer-loop-wl-D*, *unit-propagation-outer-loop-wl*) $\in$
{(T', T). $T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T} $\rightarrow_f$
⟨{(T', T). $T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}⟩*nres-rel*›
apply (*intro frefI nres-relI*)
subgoal for *x y*
apply (*rule order-trans*)

apply (*rule unit-propagation-outer-loop-wl-D-spec*[*of* $\mathcal{A}$ *x*])
apply (*auto simp*: *prod-rel-def intro*: *conc-fun-R-mono*)
done
done

definition *skip-and-resolve-loop-wl-D-inv* **where**
⟨*skip-and-resolve-loop-wl-D-inv* S_0 *brk S* ≡
skip-and-resolve-loop-wl-inv S_0 *brk S* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*⟩

definition *skip-and-resolve-loop-wl-D*
:: ⟨*nat twl-st-wl* ⇒ *nat twl-st-wl nres*⟩
where
⟨*skip-and-resolve-loop-wl-D* S_0 =
do {
ASSERT(*get-conflict-wl* S_0 ≠ *None*);
(-, *S*) ←
$WHILE_T{}^{\lambda(brk,\ S).\ skip\text{-}and\text{-}resolve\text{-}loop\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ S}$
(λ(*brk*, *S*). ¬*brk* ∧ ¬*is-decided* (*hd* (*get-trail-wl S*)))
(λ(*brk*, *S*).
do {
ASSERT(¬*brk* ∧ ¬*is-decided* (*hd* (*get-trail-wl S*)));
let D' = *the* (*get-conflict-wl S*);
let (*L*, *C*) = *lit-and-ann-of-propagated* (*hd* (*get-trail-wl S*));
if $-L \notin\#\ D'$ *then*
do {*RETURN* (*False*, *tl-state-wl S*)}
else
if get-maximum-level (*get-trail-wl S*) (*remove1-mset* $(-L)$ D') =
count-decided (*get-trail-wl S*)
then
do {*RETURN* (*update-confl-tl-wl C L S*)}
else
do {*RETURN* (*True*, *S*)}
}
)
(*False*, S_0);
RETURN S
}
⟩

lemma *literals-are-*$\mathcal{L}_{in}$*-tl-state-wl*[*simp*]:
⟨*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ (*tl-state-wl S*) = *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*⟩
by (*cases S*)
(*auto simp*: *is-*$\mathcal{L}_{all}$*-def tl-state-wl-def literals-are-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*-def*)

lemma *get-clauses-wl-tl-state*: ⟨*get-clauses-wl* (*tl-state-wl T*) = *get-clauses-wl T*⟩
unfolding *tl-state-wl-def* **by** (*cases T*) *auto*

lemma *blits-in-*$\mathcal{L}_{in}$*-skip-and-resolve*[*simp*]:
⟨*blits-in-*$\mathcal{L}_{in}$ (*tl x1aa*, *N*, *D*, *ar*, *as*, *at*, *bd*) = *blits-in-*$\mathcal{L}_{in}$ (*x1aa*, *N*, *D*, *ar*, *as*, *at*, *bd*)⟩
⟨*blits-in-*$\mathcal{L}_{in}$
(*x1aa*, *N*,
Some (*resolve-cls-wl'* (*x1aa'*, *N'*, *x1ca'*, *ar'*, *as'*, *at'*, *bd'*) *x2b*
x1b),
ar, *as*, *at*, *bd*) =
blits-in-$\mathcal{L}_{in}$ (*x1aa*, *N*, *x1ca'*, *ar*, *as*, *at*, *bd*)⟩
by (*auto simp*: *blits-in-*$\mathcal{L}_{in}$*-def*)

lemma *skip-and-resolve-loop-wl-D-spec*:
  assumes $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
  shows ‹*skip-and-resolve-loop-wl-D S* $\leq$
    $\Downarrow$ $\{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T \wedge$ *get-clauses-wl* $T =$ *get-clauses-wl* $S\}$
      (*skip-and-resolve-loop-wl S*)›
    (**is** ‹- $\leq \Downarrow$ *?R* -›)
proof −
  define *invar* **where**
  ‹*invar* = (λ(*brk*, *T*). *skip-and-resolve-loop-wl-D-inv S brk T*)›
  have *1*: ‹((*get-conflict-wl S* = *Some* {#}, *S*), *get-conflict-wl S* = *Some* {#}, *S*) $\in$ *Id*›
    by *auto*

  show *?thesis*
    unfolding *skip-and-resolve-loop-wl-D-def skip-and-resolve-loop-wl-def*
    apply (*subst* (*2*) *WHILEIT-add-post-condition*)
    apply (*refine-rcg 1 WHILEIT-refine*[**where** R = ‹$\{((i', S'), (i, S)).\ i = i' \wedge (S', S) \in$ *?R*}›])
    subgoal using *assms* **by** *auto*
    subgoal unfolding *skip-and-resolve-loop-wl-D-inv-def* **by** *fast*
    subgoal by *fast*
    subgoal by *fast*
    subgoal by *fast*
    subgoal by *auto*
    subgoal
      unfolding *skip-and-resolve-loop-wl-D-inv-def update-confl-tl-wl-def*
      by (*auto split*: *prod.splits*) (*simp add*: *get-clauses-wl-tl-state*)
    subgoal by *auto*
    subgoal
      unfolding *skip-and-resolve-loop-wl-D-inv-def update-confl-tl-wl-def*
      by (*auto split*: *prod.splits simp*: *literals-are-*$\mathcal{L}_{in}$*-def*)
    subgoal by *auto*
    subgoal by *auto*
    done
qed

definition *find-lit-of-max-level-wl′* :: ‹- $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$
   nat literal nres› **where**
  ‹*find-lit-of-max-level-wl′ M N D NE UE Q W L* =
    find-lit-of-max-level-wl (*M*, *N*, *Some D*, *NE*, *UE*, *Q*, *W*) *L*›

definition (**in** −) *list-of-mset2*
  :: ‹*nat literal* $\Rightarrow$ *nat literal* $\Rightarrow$ *nat clause* $\Rightarrow$ *nat clause-l nres*›
where
  ‹*list-of-mset2 L L′ D* =
    SPEC ($\lambda E.$ *mset* $E = D \wedge E!0 = L \wedge E!1 = L' \wedge$ *length* $E \geq 2$)›

definition *single-of-mset* **where**
  ‹*single-of-mset D* = *SPEC*($\lambda L.\ D$ = *mset* [*L*])›

definition *backtrack-wl-D-inv* **where**
  ‹*backtrack-wl-D-inv S* $\longleftrightarrow$ *backtrack-wl-inv S* $\wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*›

definition *propagate-bt-wl-D*
  :: ‹*nat literal* $\Rightarrow$ *nat literal* $\Rightarrow$ *nat twl-st-wl* $\Rightarrow$ *nat twl-st-wl nres*›
where

⟨*propagate-bt-wl-D* = ($\lambda L\ L'$ (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
  D'' ← *list-of-mset2* (−*L*) L' (*the D*);
  i ← *get-fresh-index-wl N* (*NE*+*UE*) *W*;
  let b = (*length* D'' = *2*);
  RETURN (*Propagated* (−*L*) *i* # *M*, *fmupd i* (D'', *False*) *N*,
      None, *NE*, *UE*, {#*L*#}, *W*(−*L*:= *W* (−*L*) @ [(*i*, L', *b*)], L':= *W* L' @ [(*i*, −*L*, *b*)]))
  })⟩

definition *propagate-unit-bt-wl-D*
:: ⟨*nat literal* ⇒ *nat twl-st-wl* ⇒ (*nat twl-st-wl*) *nres*⟩
where
⟨*propagate-unit-bt-wl-D* = (λL (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
    D' ← *single-of-mset* (*the D*);
    RETURN (*Propagated* (−*L*) *0* # *M*, *N*, *None*, *NE*, *add-mset* {#D'#} *UE*, {#*L*#}, *W*)
  })⟩

definition *backtrack-wl-D* :: ⟨*nat twl-st-wl* ⇒ *nat twl-st-wl nres*⟩ **where**
⟨*backtrack-wl-D S* =
  do {
    ASSERT(*backtrack-wl-D-inv S*);
    let L = *lit-of* (*hd* (*get-trail-wl S*));
    S ← *extract-shorter-conflict-wl S*;
    S ← *find-decomp-wl L S*;

    if size (*the* (*get-conflict-wl S*)) > *1*
    then do {
      L' ← *find-lit-of-max-level-wl S L*;
      propagate-bt-wl-D L L' *S*
    }
    else do {
      propagate-unit-bt-wl-D L S
    }
}⟩

lemma *backtrack-wl-D-spec*:
fixes *S* :: ⟨*nat twl-st-wl*⟩
assumes $\mathcal{A}_{in}$: ⟨*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*⟩ **and** *confl*: ⟨*get-conflict-wl S* ≠ *None*⟩
shows ⟨*backtrack-wl-D S* ≤
  ⇓ {(T', *T*). *T* = T' ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
    (*backtrack-wl S*)⟩
proof −
have *1*: ⟨((*get-conflict-wl S* = *Some* {#}, *S*), *get-conflict-wl S* = *Some* {#}, *S*) ∈ *Id*⟩
  by *auto*

have *3*: ⟨*find-lit-of-max-level-wl S M* ≤
  ⇓{(L', *L*). L' ∈# *remove1-mset* (−*M*) (*the* (*get-conflict-wl S*)) ∧ L' = *L*}
   (*find-lit-of-max-level-wl* S' M')⟩
  if ⟨*S* = S'⟩ **and** ⟨*M* = M'⟩
  for *S* S' :: ⟨*nat twl-st-wl*⟩ **and** *M* M'
  using *that* **by** (*cases S*; *cases* S') (*auto simp*: *find-lit-of-max-level-wl-def intro*!: *RES-refine*)
have *H*: ⟨*mset* ‘# *mset* (*take n* (*tl xs*)) + *a* + (*mset* ‘# *mset* (*drop* (*Suc n*) *xs*) + *b*) =
mset ‘# *mset* (*tl xs*) + *a* + *b*⟩ **for** *n* **and** *xs* :: ⟨′*a list list*⟩ **and** *a b*
  apply (*subst* (*2*) *append-take-drop-id*[*of n* ⟨*tl xs*⟩, *symmetric*])
  apply (*subst mset-append*)
  by (*auto simp*: *drop-Suc*)
have *list-of-mset*: ⟨*list-of-mset2 L* L' *D* ≤

$\Downarrow \{(E,\ F).\ F = [L,\ L'] \ @\ \mathit{remove1}\ L\ (\mathit{remove1}\ L'\ E) \wedge D = \mathit{mset}\ E \wedge E!0 = L \wedge E!1 = L' \wedge E{=}F\}$
$(\mathit{list\text{-}of\text{-}mset}\ D')\rangle$
(**is** ‹- ≤ ⇓ *?list-of-mset* -›)
if ‹$D = D'$› **and** *uL-D*: ‹$L \in\# D$› **and** *L'-D*: ‹$L' \in\# D$› **and** *L-uL'*: ‹$L \neq L'$› **for** $D\ D'\ L\ L'$
unfolding *list-of-mset-def list-of-mset2-def*
proof (*rule RES-refine*)
fix *s*
assume *s*: ‹$s \in \{E.\ \mathit{mset}\ E = D \wedge E\ !\ 0 = L \wedge E\ !\ 1 = L' \wedge \mathit{length}\ E \geq 2\}$›
then show ‹$\exists s'\in\{D'a.\ D' = \mathit{mset}\ D'a\}.$
$(s,\ s')$
$\in \{(E,\ F).$
$F = [L,\ L'] \ @\ \mathit{remove1}\ L\ (\mathit{remove1}\ L'\ E) \wedge D = \mathit{mset}\ E \wedge E\ !\ 0 = L \wedge E\ !\ 1 = L' \wedge E{=}F\}$›
apply (*cases s*; *cases* ‹*tl s*›)
using *that* **by** (*auto simp*: *diff-single-eq-union diff-diff-add-mset*[*symmetric*]
simp del: *diff-diff-add-mset*)
qed

define *extract-shorter-conflict-wl'* **where**
‹*extract-shorter-conflict-wl'* S = *extract-shorter-conflict-wl* S› **for** S :: ‹*nat twl-st-wl*›
define *find-lit-of-max-level-wl'* **where**
‹*find-lit-of-max-level-wl'* S = *find-lit-of-max-level-wl* S› **for** S :: ‹*nat twl-st-wl*›

have *extract-shorter-conflict-wl*: ‹*extract-shorter-conflict-wl'* S
$\leq \Downarrow \{(U,\ U').\ U = U' \wedge \mathit{equality\text{-}except\text{-}conflict\text{-}wl}\ U\ S \wedge \mathit{get\text{-}conflict\text{-}wl}\ U \neq \mathit{None} \wedge$
$\mathit{the}\ (\mathit{get\text{-}conflict\text{-}wl}\ U) \subseteq\# \mathit{the}\ (\mathit{get\text{-}conflict\text{-}wl}\ S) \wedge$
$-\mathit{lit\text{-}of}\ (\mathit{hd}\ (\mathit{get\text{-}trail\text{-}wl}\ S)) \in\# \mathit{the}\ (\mathit{get\text{-}conflict\text{-}wl}\ U)$
$\}\ (\mathit{extract\text{-}shorter\text{-}conflict\text{-}wl}\ S)$›
(**is** ‹- ≤ ⇓ *?extract-shorter* -›)
unfolding *extract-shorter-conflict-wl'-def extract-shorter-conflict-wl-def*
by (*cases S*)
(*auto 5 5 simp*: *extract-shorter-conflict-wl'-def extract-shorter-conflict-wl-def*
intro!: *RES-refine*)

have *find-decomp-wl*: ‹*find-decomp-wl* (*lit-of* (*hd* (*get-trail-wl S*))) T
$\leq \Downarrow \{(U,\ U').\ U = U' \wedge \mathit{equality\text{-}except\text{-}trail\text{-}wl}\ U\ T\}$
(*find-decomp-wl* (*lit-of* (*hd* (*get-trail-wl S*))) T')›
(**is** ‹- ≤ ⇓ *?find-decomp* -›)
if ‹$(T,\ T') \in$ *?extract-shorter*›
for $T\ T'$
using *that* **unfolding** *find-decomp-wl-def*
by (*cases T*) (*auto 5 5 intro*!: *RES-refine*)

have *find-lit-of-max-level-wl*:
‹*find-lit-of-max-level-wl* U (*lit-of* (*hd* (*get-trail-wl S*)))
≤ ⇓ *Id* (*find-lit-of-max-level-wl* U' (*lit-of* (*hd* (*get-trail-wl S*))))›
if
‹$(U,\ U') \in$ *?find-decomp T*›
for $T\ U\ U'$
using *that* **unfolding** *find-lit-of-max-level-wl-def*
by (*cases T*) (*auto 5 5 intro*!: *RES-refine*)

have *find-lit-of-max-level-wl'*:
‹*find-lit-of-max-level-wl'* U (*lit-of* (*hd* (*get-trail-wl S*)))
$\leq \Downarrow\{(L,\ L').\ L = L' \wedge L \in\# \mathit{remove1\text{-}mset}\ (-\mathit{lit\text{-}of}\ (\mathit{hd}\ (\mathit{get\text{-}trail\text{-}wl}\ S)))\ (\mathit{the}\ (\mathit{get\text{-}conflict\text{-}wl}$

$U))\}$
    $(find\text{-}lit\text{-}of\text{-}max\text{-}level\text{-}wl\ U'\ (lit\text{-}of\ (hd\ (get\text{-}trail\text{-}wl\ S))))$›
  (**is** ‹- $\leq \Downarrow$ *?find-lit* -›)
 if
  ‹*backtrack-wl-inv S*› **and**
  ‹*backtrack-wl-D-inv S*› **and**
  ‹$(U, U') \in$ *?find-decomp T*› **and**
  ‹$1 <$ *size* (*the* (*get-conflict-wl U*))› **and**
  ‹$1 <$ *size* (*the* (*get-conflict-wl U'*))›
 for $U\ U'\ T$
 using *that* **unfolding** *find-lit-of-max-level-wl'-def find-lit-of-max-level-wl-def*
 by (*cases U*) (*auto 5 5 intro*!: *RES-refine*)

have *is-$\mathcal{L}_{all}$-add*: ‹*is-$\mathcal{L}_{all}$* $\mathcal{A}$ $(A + B) \longleftrightarrow$ *set-mset A* $\subseteq$ *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$)› **if** ‹*is-$\mathcal{L}_{all}$* $\mathcal{A}$ *B*› **for** *A B*
 using *that* **unfolding** *is-$\mathcal{L}_{all}$-def* **by** *auto*

have *propagate-bt-wl-D*: ‹*propagate-bt-wl-D* (*lit-of* (*hd* (*get-trail-wl S*))) *L U*
   $\leq \Downarrow \{(T', T).\ T = T' \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ \mathcal{A}\ T\}$
    (*propagate-bt-wl* (*lit-of* (*hd* (*get-trail-wl S*))) *L' U'*)›
 if
  ‹*backtrack-wl-inv S*› **and**
  bt: ‹*backtrack-wl-D-inv S*› **and**
  TT': ‹$(T, T') \in$ *?extract-shorter*› **and**
  UU': ‹$(U, U') \in$ *?find-decomp T*› **and**
  ‹$1 <$ *size* (*the* (*get-conflict-wl U*))› **and**
  ‹$1 <$ *size* (*the* (*get-conflict-wl U'*))› **and**
  LL': ‹$(L, L') \in$ *?find-lit U*›
 for $L\ L'\ T\ T'\ U\ U'$
proof –
 obtain *MS NS DS NES UES W Q* **where**
   S: ‹$S = (MS, NS, Some\ DS, NES, UES, Q, W)$›
  using *bt* **by** (*cases S*; *cases* ‹*get-conflict-wl S*›)
   (*auto simp*: *backtrack-wl-D-inv-def backtrack-wl-inv-def*
    backtrack-l-inv-def state-wl-l-def)
 then obtain *DT* **where**
  T: ‹$T = (MS, NS, Some\ DT, NES, UES, Q, W)$› **and** *DT*: ‹$DT \subseteq\# DS$›
  using *TT'* **by** (*cases T'*; *cases* ‹*get-conflict-wl T'*›) *auto*
 then obtain *MU* **where**
  U: ‹$U = (MU, NS, Some\ DT, NES, UES, Q, W)$› **and** *U'*: ‹$U' = U$›
  using *UU'* **by** (*cases U*) *auto*
 define *list-of-mset* **where**
  ‹*list-of-mset D L L'* = *?list-of-mset D L L'*› **for** *D* **and** *L L'* :: ‹*nat literal*›
 have [*simp*]: ‹*get-conflict-wl S* = *Some DS*›
  using *S* **by** *auto*
 obtain *T U* **where**
  dist: ‹*distinct-mset* (*the* (*get-conflict-wl S*))› **and**
  ST: ‹$(S, T) \in$ *state-wl-l None*› **and**
  TU: ‹$(T, U) \in$ *twl-st-l None*› **and**
  alien: ‹*cdcl$_W$-restart-mset.no-strange-atm* (*state$_W$-of U*)›
  using *bt* **unfolding** *backtrack-wl-D-inv-def backtrack-wl-inv-def backtrack-l-inv-def*
  twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def
  cdcl$_W$-restart-mset.distinct-cdcl$_W$-state-def
  apply –
  apply *normalize-goal+*
  by (*auto simp*: *twl-st-wl twl-st-l twl-st*)

then have ‹*distinct-mset DT*›
  using *DT* **unfolding** *S* **by** (*auto simp*: *distinct-mset-mono*)
then have [*simp*]: ‹$L \neq -$*lit-of* (*hd MS*)›
  using *LL'* **by** (*auto simp*: *U S dest*: *distinct-mem-diff-mset*)

have ‹$x \in\#$ *all-lits-of-m* (*the* (*get-conflict-wl S*)) $\Longrightarrow$
    $x \in\#$ *all-lits-of-mm* ({#*mset x*. $x \in\#$ *ran-mf* (*get-clauses-wl S*)#} + *get-unit-clauses-wl S*)›
  for *x*
  using *alien ST TU* **unfolding** $cdcl_W$*-restart-mset.no-strange-atm-def*
  all-clss-lf-ran-m[*symmetric*] *set-mset-union*
  by (*auto simp*: *twl-st-wl twl-st-l twl-st in-all-lits-of-m-ain-atms-of-iff*
    in-all-lits-of-mm-ain-atms-of-iff get-unit-clauses-wl-alt-def)
then have ‹$x \in\#$ *all-lits-of-m DS* $\Longrightarrow$
    $x \in\#$ *all-lits-of-mm* ({#*mset x*. $x \in\#$ *ran-mf NS*#} + (*NES* + *UES*))›
  for *x*
  by (*simp add*: *S*)
then have *H*: ‹$x \in\#$ *all-lits-of-m DT* $\Longrightarrow$
    $x \in\#$ *all-lits-of-mm* ({#*mset x*. $x \in\#$ *ran-mf NS*#} + (*NES* + *UES*))›
  for *x*
  using *DT all-lits-of-m-mono* **by** *blast*
have *propa-ref*: ‹((*Propagated* (− *lit-of* (*hd* (*get-trail-wl S*))) *i* # *MU*, *fmupd i* (*D*, *False*) *NS*,
  None, *NES*, *UES*, *unmark* (*hd* (*get-trail-wl S*)), *W*
  (− *lit-of* (*hd* (*get-trail-wl S*)) :=
     W (− *lit-of* (*hd* (*get-trail-wl S*))) @ [(*i*, *L*, *length D* = *2*)],
   L := *W L* @ [(*i*, −*lit-of* (*hd* (*get-trail-wl S*)), *length D* = *2*)])),
 Propagated (− *lit-of* (*hd* (*get-trail-wl S*))) *i'* # *MU*,
 fmupd i'
  ([− *lit-of* (*hd* (*get-trail-wl S*)), *L'*] @
   remove1 (− *lit-of* (*hd* (*get-trail-wl S*))) (*remove1 L' D'*),
   False)
  NS,
 None, *NES*, *UES*, *unmark* (*hd* (*get-trail-wl S*)), *W*
 (− *lit-of* (*hd* (*get-trail-wl S*)) :=
    W (− *lit-of* (*hd* (*get-trail-wl S*))) @ [(*i'*, *L'*,
    length
       ([− *lit-of* (*hd* (*get-trail-wl S*)), *L'*] @
        remove1 (− *lit-of* (*hd* (*get-trail-wl S*))) (*remove1 L' D'*)) =
       2)],
  L' := *W L'* @ [(*i'*, − *lit-of* (*hd* (*get-trail-wl S*)),
    length
       ([− *lit-of* (*hd* (*get-trail-wl S*)), *L'*] @
        remove1 (− *lit-of* (*hd* (*get-trail-wl S*))) (*remove1 L' D'*)) =
       2)]))
$\in$ {(*T'*, *T*). *T* = *T'* $\wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}›
  if
    DD': ‹(*D*, *D'*) $\in$ *list-of-mset* (*the* (*Some DT*)) (− *lit-of* (*hd* (*get-trail-wl S*))) *L*› **and**
    ii': ‹(*i*, *i'*) $\in$ {(*i*, *i'*). *i* = *i'* $\wedge$ *i* $\notin\#$ *dom-m NS*}›
  for *i i' D D'*
proof −
  have [*simp*]: ‹*i* = *i'*› ‹*L* = *L'*› **and** *i'-dom*: ‹*i'* $\notin\#$ *dom-m NS*›
    using *ii' LL'* **by** *auto*
  have
    D: ‹*D* = [− *lit-of* (*hd* (*get-trail-wl S*)), *L*] @
      remove1 (− *lit-of* (*hd* (*get-trail-wl S*))) (*remove1 L D'*)› **and**
    DT-D: ‹*DT* = *mset D*›
    using *DD'* **unfolding** *list-of-mset-def*

by *force+*
have ‹$L \in$ *set D*›
using *ii′ LL′* **by** (*auto simp*: *U DT-D dest*!: *in-diffD*)
have *K*: ‹$L \in$ *set D* $\Longrightarrow$ $L \in\#$ *all-lits-of-m* (*mset D*)› **for** *L*
unfolding *in-multiset-in-set*[*symmetric*]
apply (*drule multi-member-split*)
by (*auto simp*: *all-lits-of-m-add-mset*)
have [*simp*]: ‹$-$ *lit-of* (*hd* (*get-trail-wl S*)) # *L′* #
remove1 ($-$ *lit-of* (*hd* (*get-trail-wl S*))) (*remove1 L′ D′*) = *D*›
using *D* **by** *simp*
then have *1*[*simp*]: ‹$-$ *lit-of* (*hd MS*) # *L′* #
remove1 ($-$ *lit-of* (*hd MS*)) (*remove1 L′ D′*) = *D*›
using *D* **by** (*simp add*: *S*)
have ‹$-$ *lit-of* (*hd MS*) $\in$ *set D*›
apply (*subst 1*[*symmetric*])
unfolding *set-append list.sel*
by (*rule list.set-intros*)
have ‹*set-mset* (*all-lits-of-m* (*mset D*)) $\subseteq$
set-mset (*all-lits-of-mm* ({#*mset* (*fst x*). $x \in\#$ *ran-m NS*#} + (*NES* + *UES*)))›
by (*auto dest*!: *H*[*unfolded DT-D*])
then have [*simp*]: ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits* (*fmupd i′* (*D*, *False*) *NS*) (*NES* + *UES*)) =
is-$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits NS* (*NES* + *UES*))›
‹*set-mset* ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits* (*fmupd i′* (*D*, *False*) *NS*) (*NES* + *UES*))) =
set-mset ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits NS* (*NES* + *UES*)))›
using *i′-dom* **unfolding** *is-*$\mathcal{L}_{all}$*-def all-lits-def*
by (*auto 5 5 simp add*: *ran-m-mapsto-upd-notin all-lits-of-mm-add-mset*
in-$\mathcal{L}_{all}$*-atm-of-*$\mathcal{A}_{in}$ *atm-of-eq-atm-of*)

have ‹$x \in\#$ *all-lits-of-mm* ({#*mset* (*fst x*). $x \in\#$ *ran-m NS*#} + (*NES* + *UES*)) $\Longrightarrow$
$x \in\#$ $\mathcal{L}_{all}$ $\mathcal{A}$› **for** *x*
using *i′-dom* $\mathcal{A}_{in}$ **unfolding** *is-*$\mathcal{L}_{all}$*-def literals-are-*$\mathcal{L}_{in}$*-def*
by (*auto simp*: *S all-lits-def*)
then show *?thesis*
using *i′-dom* $\mathcal{A}_{in}$ *K*[*OF* ‹$L \in$ *set D*›] *K*[*OF* ‹$-$ *lit-of* (*hd MS*) $\in$ *set D*›]
unfolding *literals-are-*$\mathcal{L}_{in}$*-def*
by (*auto simp*: *ran-m-mapsto-upd-notin all-lits-of-mm-add-mset*
blits-in-$\mathcal{L}_{in}$*-def is-*$\mathcal{L}_{all}$*-add S dest*!: *H*[*unfolded DT-D*])
qed

define *get-fresh-index2* **where**
‹*get-fresh-index2 N NUE W* = *get-fresh-index-wl* (*N* :: *nat clauses-l*) (*NUE* :: *nat clauses*)
(*W*::*nat literal* ⇒ (*nat watcher*) *list*)›
for *N NUE W*
have *fresh*: ‹*get-fresh-index-wl N NUE W* $\le \Downarrow$ {(*i*, *i′*). $i = i' \wedge i \notin\#$ *dom-m N*} (*get-fresh-index2*
N′ NUE′ W′)›
if ‹*N* = *N′*› ‹*NUE* = *NUE′*› ‹*W*=*W′*›**for** *N N′ NUE NUE′ W W′*
using *that* **by** (*auto simp*: *get-fresh-index-wl-def get-fresh-index2-def intro*!: *RES-refine*)
show *?thesis*
unfolding *propagate-bt-wl-D-def propagate-bt-wl-def propagate-bt-wl-D-def U U′ S T*
apply (*subst* (*2*) *get-fresh-index2-def*[*symmetric*])
apply *clarify*
apply (*refine-rcg list-of-mset fresh*)
subgoal ..
subgoal using *TT′ T* **by** (*auto simp*: *U S*)
subgoal using *LL′* **by** (*auto simp*: *T U S dest*: *in-diffD*)
subgoal by *auto*

subgoal ..
subgoal ..
subgoal ..
subgoal for *D D′ i i′*
unfolding *list-of-mset-def*[*symmetric*] *U*[*symmetric*] *U′*[*symmetric*] *S*[*symmetric*] *T*[*symmetric*]
by (*rule propa-ref*)
done
qed

have *propagate-unit-bt-wl-D*: ‹*propagate-unit-bt-wl-D* (*lit-of* (*hd* (*get-trail-wl S*))) *U*
$\leq$ *SPEC* ($\lambda c.$ (*c*, *propagate-unit-bt-wl* (*lit-of* (*hd* (*get-trail-wl S*))) *U′*)
$\in \{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T\}$)›
if
‹*backtrack-wl-inv S*› **and**
bt: ‹*backtrack-wl-D-inv S*› **and**
TT′: ‹(*T*, *T′*) ∈ *?extract-shorter*› **and**
UU′: ‹(*U*, *U′*) ∈ *?find-decomp T*› **and**
‹¬*1* < *size* (*the* (*get-conflict-wl U*))› **and**
‹¬*1* < *size* (*the* (*get-conflict-wl U′*))›
for *L L′ T T′ U U′*
proof −
obtain *MS NS DS NES UES W Q* **where**
S: ‹*S* = (*MS*, *NS*, *Some DS*, *NES*, *UES*, *Q*, *W*)›
using *bt* **by** (*cases S*; *cases* ‹*get-conflict-wl S*›)
(*auto simp*: *backtrack-wl-D-inv-def backtrack-wl-inv-def*
backtrack-l-inv-def state-wl-l-def)
then obtain *DT* **where**
T: ‹*T* = (*MS*, *NS*, *Some DT*, *NES*, *UES*, *Q*, *W*)› **and** *DT*: ‹*DT* ⊆# *DS*›
using *TT′* **by** (*cases T′*; *cases* ‹*get-conflict-wl T′*›) *auto*
then obtain *MU* **where**
U: ‹*U* = (*MU*, *NS*, *Some DT*, *NES*, *UES*, *Q*, *W*)› **and** *U′*: ‹*U′* = *U*›
using *UU′* **by** (*cases U*) *auto*
define *list-of-mset* **where**
‹*list-of-mset D L L′* = *?list-of-mset D L L′*› **for** *D* **and** *L L′* :: ‹*nat literal*›
have [*simp*]: ‹*get-conflict-wl S* = *Some DS*›
using *S* **by** *auto*
obtain *T U* **where**
dist: ‹*distinct-mset* (*the* (*get-conflict-wl S*))› **and**
ST: ‹(*S*, *T*) ∈ *state-wl-l None*› **and**
TU: ‹(*T*, *U*) ∈ *twl-st-l None*› **and**
alien: ‹$cdcl_W$*-restart-mset.no-strange-atm* ($state_W$*-of U*)›
using *bt* **unfolding** *backtrack-wl-D-inv-def backtrack-wl-inv-def backtrack-l-inv-def*
twl-struct-invs-def $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def*
apply −
apply *normalize-goal+*
by (*auto simp*: *twl-st-wl twl-st-l twl-st*)

then have ‹*distinct-mset DT*›
using *DT* **unfolding** *S* **by** (*auto simp*: *distinct-mset-mono*)
have ‹*x* ∈# *all-lits-of-m* (*the* (*get-conflict-wl S*)) ⟹
x ∈# *all-lits-of-mm* ({#*mset x*. *x* ∈# *ran-mf* (*get-clauses-wl S*)#} + *get-unit-init-clss-wl S*)›
for *x*
using *alien ST TU* **unfolding** $cdcl_W$*-restart-mset.no-strange-atm-def*
all-clss-lf-ran-m[*symmetric*] *set-mset-union*
by (*auto simp*: *twl-st-wl twl-st-l twl-st in-all-lits-of-m-ain-atms-of-iff*

in-all-lits-of-mm-ain-atms-of-iff)
then have ‹x ∈# *all-lits-of-m DS* ⟹
x ∈# *all-lits-of-mm* ({#*mset* x. x ∈# *ran-mf NS*#} + *NES*)›
for x
by (*simp add*: *S*)
then have *H*: ‹x ∈# *all-lits-of-m DT* ⟹
x ∈# *all-lits-of-mm* ({#*mset* x. x ∈# *ran-mf NS*#} + *NES*)›
for x
using *DT all-lits-of-m-mono* **by** *blast*
then have $\mathcal{A}_{in}$-*D*: ‹*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}$ *DT*›
using *DT* $\mathcal{A}_{in}$ **unfolding** *literals-are-in-*$\mathcal{L}_{in}$*-def S is-*$\mathcal{L}_{all}$*-def literals-are-*$\mathcal{L}_{in}$*-def*
by (*auto simp*: *all-lits-of-mm-union all-lits-def*)
have [*simp*]: ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits NS* (*add-mset* {#*x*#} (*NES* + *UES*))) =
is-$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits NS* (*NES* + *UES*))›
‹*set-mset* ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits NS* (*add-mset* {#*x*#} (*NES* + *UES*)))) =
set-mset ($\mathcal{L}_{all}$ (*atm-of* ‘# *all-lits NS* (*NES* + *UES*)))›
if ‹*DT* = {#*x*#}›
for x
using *H*[*of x*] *H*[*of* ‹−*x*›] *that*
unfolding *is-*$\mathcal{L}_{all}$*-def all-lits-def*
by (*auto simp add*: *all-lits-of-mm-add-mset in-*$\mathcal{L}_{all}$*-atm-of-*$\mathcal{A}_{in}$ *atm-of-eq-atm-of*
all-lits-of-m-add-mset insert-absorb all-lits-of-mm-union)

show *?thesis*
unfolding *propagate-unit-bt-wl-D-def propagate-unit-bt-wl-def U U′ single-of-mset-def*
apply *clarify*
apply *refine-vcg*
using $\mathcal{A}_{in}$-*D* $\mathcal{A}_{in}$ **unfolding** *literals-are-*$\mathcal{L}_{in}$*-def*
by (*auto simp*: *clauses-def mset-take-mset-drop-mset mset-take-mset-drop-mset′*
all-lits-of-mm-add-mset is-$\mathcal{L}_{all}$*-add literals-are-in-*$\mathcal{L}_{in}$*-def S*
blits-in-$\mathcal{L}_{in}$*-def*)
qed
show *?thesis*
unfolding *backtrack-wl-D-def backtrack-wl-def find-lit-of-max-level-wl′-def*
apply (*subst extract-shorter-conflict-wl′-def*[*symmetric*])
apply (*subst find-lit-of-max-level-wl′-def*[*symmetric*])
supply [[*goals-limit=1*]]
apply (*refine-vcg extract-shorter-conflict-wl find-lit-of-max-level-wl find-decomp-wl*
find-lit-of-max-level-wl′ propagate-bt-wl-D propagate-unit-bt-wl-D)
subgoal using $\mathcal{A}_{in}$ **unfolding** *backtrack-wl-D-inv-def* **by** *fast*
subgoal by *auto*
by *assumption+*
qed

Decide or Skip

definition *find-unassigned-lit-wl-D*
:: ‹*nat twl-st-wl* ⇒ (*nat twl-st-wl* × *nat literal option*) *nres*›
where
‹*find-unassigned-lit-wl-D S* = (
SPEC(λ((*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*), *L*).
S = (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) ∧
(*L* ≠ *None* ⟶
undefined-lit M (*the L*) ∧ *the L* ∈# $\mathcal{L}_{all}$ (*all-atms N NE*) ∧
atm-of (*the L*) ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf N* + *NE*)) ∧
(*L* = *None* ⟶ (∄ *L′*. *undefined-lit M L′* ∧

$atm\text{-}of\ L' \in atms\text{-}of\text{-}mm\ (clause\ `\#\ twl\text{-}clause\text{-}of\ `\#\ init\text{-}clss\text{-}lf\ N + NE)))))$
›

definition *decide-wl-or-skip-D-pre* :: ‹*nat twl-st-wl* ⇒ *bool*› **where**
‹*decide-wl-or-skip-D-pre S* ⟷
  decide-wl-or-skip-pre S ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*›

definition *decide-wl-or-skip-D*
  :: ‹*nat twl-st-wl* ⇒ (*bool* × *nat twl-st-wl*) *nres*›
where
  ‹*decide-wl-or-skip-D S* = (*do* {
    ASSERT(*decide-wl-or-skip-D-pre S*);
    (*S*, *L*) ← *find-unassigned-lit-wl-D S*;
    case L of
      None ⇒ *RETURN* (*True*, *S*)
    | *Some L* ⇒ *RETURN* (*False*, *decide-lit-wl L S*)
  })
›

theorem *decide-wl-or-skip-D-spec*:
  assumes ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
  shows ‹*decide-wl-or-skip-D S*
    ≤ ⇓ {((*b'*, *T'*), *b*, *T*). *b* = *b'* ∧ *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*} (*decide-wl-or-skip S*)›
proof −
  have *H*: ‹*find-unassigned-lit-wl-D S* ≤ ⇓ {((*S'*, *L'*), *L*). *S'* = *S* ∧ *L* = *L'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S* ∧
        (*L* ≠ *None* ⟶
          undefined-lit (*get-trail-wl S*) (*the L*) ∧
          atm-of (*the L*) ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf* (*get-clauses-wl S*)
              + *get-unit-init-clss-wl S*)) ∧
        (*L* = *None* ⟶ (∄ *L'*. *undefined-lit* (*get-trail-wl S*) *L'* ∧
          atm-of L' ∈ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *init-clss-lf* (*get-clauses-wl S*)
              + *get-unit-init-clss-wl S*)))}
      (*find-unassigned-lit-wl S'*)›
    (**is** ‹- ≤ ⇓ *?find* -›)
    if ‹*S* = *S'*› **and** ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
    for *S S'* :: ‹*nat twl-st-wl*›
    using *that*(*2*) **unfolding** *find-unassigned-lit-wl-def find-unassigned-lit-wl-D-def that*(*1*)
    by (*cases S'*) (*auto intro*!: *RES-refine simp*: *mset-take-mset-drop-mset'*)
  have [*refine*]: ‹*x* = *x'* ⟹ (*x*, *x'*) ∈ ⟨*Id*⟩ *option-rel*›
    for *x x'* **by** *auto*
  have *decide-lit-wl*: ‹((*False*, *decide-lit-wl L T*), *False*, *decide-lit-wl L' S'*)
      ∈ {((*b'*, *T'*), *b*, *T*).
          b = *b'* ∧ *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}›
    if
      SS': ‹(*S*, *S'*) ∈ {(*T'*, *T*). *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}› **and**
      ‹*decide-wl-or-skip-pre S'*› **and**
      pre: ‹*decide-wl-or-skip-D-pre S*› **and**
      LT-L': ‹(*LT*, *bL'*) ∈ *?find S*› **and**
      LT: ‹*LT* = (*T*, *bL*)› **and**
      ‹*bL'* = *Some L'*› **and**
      ‹*bL* = *Some L*› **and**
      LL': ‹(*L*, *L'*) ∈ *Id*›
    for *S S' L L' LT bL bL' T*
  proof −
    have $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*› **and** [*simp*]: ‹*T* = *S*›

using *LT-L′ pre* **unfolding** *LT decide-wl-or-skip-D-pre-def*
by *fast+*
have [*simp*]: ‹$S' = S$› ‹$L = L'$›
using *SS′ LL′* **by** *simp-all*
have ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$* (*decide-lit-wl L′ S*)›
using $\mathcal{A}_{in}$
by (*cases S*) (*auto simp*: *decide-lit-wl-def clauses-def blits-in-$\mathcal{L}_{in}$-def literals-are-$\mathcal{L}_{in}$-def*)
then show *?thesis*
by *auto*
qed

have ‹(*decide-wl-or-skip-D*, *decide-wl-or-skip*) $\in \{(T', T).\ T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ T*$\} \rightarrow_f$ $\langle\{((b', T'), (b, T)).\ b = b' \wedge T = T' \wedge$ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ T*$\}\rangle$ *nres-rel*›
unfolding *decide-wl-or-skip-D-def decide-wl-or-skip-def*
apply (*intro frefI*)
apply (*refine-vcg H*)
subgoal unfolding *decide-wl-or-skip-D-pre-def* **by** *blast*
subgoal by *simp*
subgoal by *auto*
subgoal by *simp*
subgoal unfolding *decide-wl-or-skip-D-pre-def* **by** *fast*
subgoal by (*rule decide-lit-wl*) *assumption+*
done
then show *?thesis*
using *assms* **by** (*cases S*) (*auto simp*: *fref-def nres-rel-def*)
qed

Backtrack, Skip, Resolve or Decide

definition *cdcl-twl-o-prog-wl-D-pre* **where**
‹*cdcl-twl-o-prog-wl-D-pre S* $\longleftrightarrow$ *cdcl-twl-o-prog-wl-pre S* $\wedge$ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*›

definition *cdcl-twl-o-prog-wl-D*
:: ‹*nat twl-st-wl* $\Rightarrow$ (*bool* $\times$ *nat twl-st-wl*) *nres*›
where
‹*cdcl-twl-o-prog-wl-D S* =
do {
ASSERT(*cdcl-twl-o-prog-wl-D-pre S*);
if get-conflict-wl S = *None*
then decide-wl-or-skip-D S
else do {
if count-decided (*get-trail-wl S*) > *0*
then do {
T ← *skip-and-resolve-loop-wl-D S*;
ASSERT(*get-conflict-wl T* ≠ *None* ∧ *get-clauses-wl S* = *get-clauses-wl T*);
U ← *backtrack-wl-D T*;
RETURN (*False*, *U*)
}
else RETURN (*True*, *S*)
}
}
›

theorem *cdcl-twl-o-prog-wl-D-spec*:
assumes ‹*literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ S*›

shows ‹*cdcl-twl-o-prog-wl-D S* ≤ ⇓ {((*b′*, *T′*), (*b*, *T*)). *b* = *b′* ∧ *T* = *T′* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
(*cdcl-twl-o-prog-wl S*)›
proof −
have *1*: ‹*backtrack-wl-D S* ≤
⇓ {(*T′*, *T*). *T* = *T′* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
(*backtrack-wl T*)› **if** ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› **and** ‹*get-conflict-wl S* ≠ *None*› **and** ‹*S* = *T*›
for *S T*
using *backtrack-wl-D-spec*[*of* $\mathcal{A}$ *S*] *that* **by** *fast*
have *2*: ‹*skip-and-resolve-loop-wl-D S* ≤
⇓ {(*T′*, *T*). *T* = *T′* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T* ∧ *get-clauses-wl T* = *get-clauses-wl S*}
(*skip-and-resolve-loop-wl T*)›
if $\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› ‹*S* = *T*›
for *S T*
using *skip-and-resolve-loop-wl-D-spec*[*of* $\mathcal{A}$ *S*] *that* **by** *fast*
show *?thesis*
using *assms*
unfolding *cdcl-twl-o-prog-wl-D-def cdcl-twl-o-prog-wl-def*
apply (*refine-vcg decide-wl-or-skip-D-spec 1 2*)
subgoal unfolding *cdcl-twl-o-prog-wl-D-pre-def* **by** *auto*
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *simp*
subgoal by *auto*
subgoal by *auto*
done
qed

theorem *cdcl-twl-o-prog-wl-D-spec′*:
‹(*cdcl-twl-o-prog-wl-D*, *cdcl-twl-o-prog-wl*) ∈
{(*S*,*S′*). (*S*,*S′*) ∈ *Id* ∧*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*} $\rightarrow_f$
⟨*bool-rel* $\times_r$ {(*T′*, *T*). *T* = *T′* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}⟩ *nres-rel*›
apply (*intro frefI nres-relI*)
subgoal for *x y*
apply (*rule order-trans*)
apply (*rule cdcl-twl-o-prog-wl-D-spec*[*of* $\mathcal{A}$ *x*])
apply (*auto simp*: *prod-rel-def intro*: *conc-fun-R-mono*)
done
done

Full Strategy

definition *cdcl-twl-stgy-prog-wl-D*
:: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*›
where
‹*cdcl-twl-stgy-prog-wl-D* S_0 =
do {
do {
(*brk*, *T*) ← $WHILE_T$$^{\lambda(brk,\ T).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}inv\ S_0\ (brk,\ T)\ \wedge\ literals\text{-}are\text{-}\mathcal{L}_{in}\ (all\text{-}atms\text{-}st\ T)\ T}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*).
do {

$T \leftarrow$ *unit-propagation-outer-loop-wl-D S*;
cdcl-twl-o-prog-wl-D T
})
(*False*, S_0);
RETURN T
}
}
›

theorem *cdcl-twl-stgy-prog-wl-D-spec*:
assumes ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
shows ‹*cdcl-twl-stgy-prog-wl-D S* $\leq \Downarrow$ {(*T'*, *T*). *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
(*cdcl-twl-stgy-prog-wl S*)›
proof −
have *1*: ‹((*False*, *S*), *False*, *S*) ∈ {((*brk'*, *T'*), *brk*, *T*). *brk* = *brk'* ∧ *T* = *T'* ∧
literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}›
using *assms* **by** *fast*
have *2*: ‹*unit-propagation-outer-loop-wl-D S* $\leq \Downarrow$ {(*T'*, *T*). *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
(*unit-propagation-outer-loop-wl T*)› **if** ‹*S* = *T*› ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› **for** *S T*
using *unit-propagation-outer-loop-wl-D-spec*[*of* $\mathcal{A}$ *S*] *that* **by** *fast*
have *3*: ‹*cdcl-twl-o-prog-wl-D S* $\leq \Downarrow$ {((*b'*, *T'*), *b*, *T*). *b* = *b'* ∧ *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
(*cdcl-twl-o-prog-wl T*)› **if** ‹*S* = *T*› ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› **for** *S T*
using *cdcl-twl-o-prog-wl-D-spec*[*of* $\mathcal{A}$ *S*] *that* **by** *fast*
show *?thesis*
unfolding *cdcl-twl-stgy-prog-wl-D-def cdcl-twl-stgy-prog-wl-def*
apply (*refine-vcg 1 2 3*)
subgoal by *auto*
subgoal by *auto*
subgoal by *fast*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

lemma *cdcl-twl-stgy-prog-wl-D-spec'*:
‹(*cdcl-twl-stgy-prog-wl-D*, *cdcl-twl-stgy-prog-wl*) ∈
{(*S*,*S'*). (*S*,*S'*) ∈ *Id* ∧*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*} $\rightarrow_f$
⟨{(*T'*, *T*). *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}⟩ *nres-rel*›
by (*intro frefI nres-relI*)
(*auto intro*: *cdcl-twl-stgy-prog-wl-D-spec*)

definition *cdcl-twl-stgy-prog-wl-D-pre* **where**
‹*cdcl-twl-stgy-prog-wl-D-pre S U* ⟷
(*cdcl-twl-stgy-prog-wl-pre S U* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*)›

lemma *cdcl-twl-stgy-prog-wl-D-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-D-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-wl-D S* $\leq \Downarrow$ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S'*)›
proof −
have *T*: ‹*cdcl-twl-stgy-prog-wl-pre S S'* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*›
using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-D-pre-def* **by** *blast*

show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-wl-D-spec*[*of* ‹*all-atms-st S*›]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step'*)
apply (*rule cdcl-twl-stgy-prog-wl-spec-final*[*of - S'*])
subgoal using *T* **by** *fast*
subgoal unfolding *conc-fun-chain* **by** (*rule conc-fun-R-mono*) *blast*
done
done
qed

definition *cdcl-twl-stgy-prog-break-wl-D* :: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*›
where
‹*cdcl-twl-stgy-prog-break-wl-D* S_0 =
do {
b ← *SPEC* (λ-. *True*);
(b, *brk*, T) ← $WHILE_T{}^{\lambda(b,\ brk,\ T).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}wl\text{-}inv\ S_0\ (brk,\ T)\ \wedge}$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* T) T
(λ(b, *brk*, -). b ∧ ¬*brk*)
(λ(b, *brk*, S).
do {
ASSERT(b);
T ← *unit-propagation-outer-loop-wl-D* S;
(*brk*, T) ← *cdcl-twl-o-prog-wl-D* T;
b ← *SPEC* (λ-. *True*);
RETURN(b, *brk*, T)
})
(b, *False*, S_0);
if brk then RETURN T
else cdcl-twl-stgy-prog-wl-D T
}›

theorem *cdcl-twl-stgy-prog-break-wl-D-spec*:
assumes ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ S›
shows ‹*cdcl-twl-stgy-prog-break-wl-D* S ≤ ⇓ {(T', T). $T = T'$ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}
(*cdcl-twl-stgy-prog-break-wl* S)›
proof −
define f **where** ‹f ≡ *SPEC* (λ-::*bool*. *True*)›
have *1*: ‹((b, *False*, S), b, *False*, S) ∈ {((b', brk', T'), b, *brk*, T). $b = b'$ ∧ $brk = brk'$ ∧
$T = T'$ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}›
for b
using *assms* **by** *fast*
have *1*: ‹((b, *False*, S), b', *False*, S) ∈ {((b', brk', T'), b, *brk*, T). $b = b'$ ∧ $brk = brk'$ ∧
$T = T'$ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}›
if ‹(b, b') ∈ *bool-rel*›
for b b'
using *assms that* **by** *fast*

have *2*: ‹*unit-propagation-outer-loop-wl-D* S ≤ ⇓ {(T', T). $T = T'$ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}
(*unit-propagation-outer-loop-wl* T)› **if** ‹$S = T$› ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ S› **for** S T
using *unit-propagation-outer-loop-wl-D-spec*[*of* $\mathcal{A}$ S] *that* **by** *fast*
have *3*: ‹*cdcl-twl-o-prog-wl-D* S ≤ ⇓ {((b', T'), b, T). $b = b'$ ∧ $T = T'$ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ T}
(*cdcl-twl-o-prog-wl* T)› **if** ‹$S = T$› ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ S› **for** S T
using *cdcl-twl-o-prog-wl-D-spec*[*of* $\mathcal{A}$ S] *that* **by** *fast*

show *?thesis*
unfolding *cdcl-twl-stgy-prog-break-wl-D-def cdcl-twl-stgy-prog-break-wl-def f-def*[*symmetric*]
apply (*refine-vcg 1 2 3*)
subgoal by *auto*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by *fast*
subgoal by (*fast intro*!: *cdcl-twl-stgy-prog-wl-D-spec*)
done
qed

lemma *cdcl-twl-stgy-prog-break-wl-D-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-D-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-break-wl-D S* $\leq \Downarrow$ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S'*)›
proof −
have *T*: ‹*cdcl-twl-stgy-prog-wl-pre S S'* $\wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*›
using *assms* **unfolding** *cdcl-twl-stgy-prog-wl-D-pre-def* **by** *blast*
show *?thesis*
apply (*rule order-trans*[*OF cdcl-twl-stgy-prog-break-wl-D-spec*[*of* ‹*all-atms-st S*›]])
subgoal using *T* **by** *auto*
subgoal
apply (*rule order-trans*)
apply (*rule ref-two-step'*)
apply (*rule cdcl-twl-stgy-prog-break-wl-spec-final*[*of - S'*])
subgoal using *T* **by** *fast*
subgoal unfolding *conc-fun-chain* **by** (*rule conc-fun-R-mono*) *blast*
done
done
qed

The definition is here to be shared later.

definition *get-propagation-reason* :: ‹(*'v*, *'mark*) *ann-lits* $\Rightarrow$ *'v literal* $\Rightarrow$ *'mark option nres*› **where**
‹*get-propagation-reason M L* = *SPEC*(λC. $C \neq$ *None* $\longrightarrow$ *Propagated L* (*the C*) $\in$ *set M*)›

end
theory *Watched-Literals-Watch-List-Domain-Restart*
imports *Watched-Literals-Watch-List-Domain Watched-Literals-Watch-List-Restart*
begin

lemma *cdcl-twl-restart-get-all-init-clss*:
assumes ‹*cdcl-twl-restart S T*›
shows ‹*get-all-init-clss T* = *get-all-init-clss S*›
using *assms* **by** (*induction rule*: *cdcl-twl-restart.induct*) *auto*

lemma *rtranclp-cdcl-twl-restart-get-all-init-clss*:
assumes ‹*cdcl-twl-restart*** *S T*›
shows ‹*get-all-init-clss T* = *get-all-init-clss S*›

using *assms* **by** (*induction rule*: *rtranclp-induct*) (*auto simp*: *cdcl-twl-restart-get-all-init-clss*)

As we have a specialised version of *correct-watching*, we defined a special version for the inclusion of the domain:

definition *all-init-lits* :: ‹(*nat*, *'v literal list* × *bool*) *fmap* ⇒ *'v literal multiset multiset* ⇒ *'v literal multiset*› **where**
‹*all-init-lits S NUE* = *all-lits-of-mm* ((λ*C*. *mset C*) '# *init-clss-lf S* + *NUE*)›

abbreviation *all-init-lits-st* :: ‹*'v twl-st-wl* ⇒ *'v literal multiset*› **where**
‹*all-init-lits-st S* ≡ *all-init-lits* (*get-clauses-wl S*) (*get-unit-init-clss-wl S*)›

definition *all-init-atms* :: ‹- ⇒ - ⇒ *'v multiset*› **where**
‹*all-init-atms N NUE* = *atm-of* '# *all-init-lits N NUE*›

declare *all-init-atms-def*[*symmetric*, *simp*]

lemma *all-init-atms-alt-def*:
‹*set-mset* (*all-init-atms N NE*) = *atms-of-mm* (*mset* '# *init-clss-lf N*) ∪ *atms-of-mm NE*›
unfolding *all-init-atms-def all-init-lits-def*
by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff*
all-lits-of-mm-def atms-of-ms-def image-UN
atms-of-def
dest!: *multi-member-split*[*of* ‹(-, -)› ‹*ran-m N*›]
dest: *multi-member-split atm-of-lit-in-atms-of*
simp del: *set-image-mset*)

abbreviation *all-init-atms-st* :: ‹*'v twl-st-wl* ⇒ *'v multiset*› **where**
‹*all-init-atms-st S* ≡ *atm-of* '# *all-init-lits-st S*›

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

definition *literals-are-*$\mathcal{L}_{in}$*'* :: ‹*nat multiset* ⇒ *nat twl-st-wl* ⇒ *bool*› **where**
‹*literals-are-*$\mathcal{L}_{in}$*'* $\mathcal{A}$ *S* ≡
is-$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*mset* '# *init-clss-lf* (*get-clauses-wl S*)
+ *get-unit-init-clss-wl S*)) ∧
blits-in-$\mathcal{L}_{in}$*' S*›

lemma $\mathcal{L}_{all}$*-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *set-mset* ($\mathcal{L}_{all}$ $\mathcal{B}$)›
unfolding *literals-are-*$\mathcal{L}_{in}$*'-def blits-in-*$\mathcal{L}_{in}$*'-def* $\mathcal{L}_{all}$*-def*
by *auto*

lemma *literals-are-*$\mathcal{L}_{in}$*'-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *literals-are-*$\mathcal{L}_{in}$*'* $\mathcal{A}$ *S* = *literals-are-*$\mathcal{L}_{in}$*'* $\mathcal{B}$ *S*›
using $\mathcal{L}_{all}$*-cong*[*of* $\mathcal{A}$ $\mathcal{B}$]
unfolding *literals-are-*$\mathcal{L}_{in}$*'-def blits-in-*$\mathcal{L}_{in}$*'-def is-*$\mathcal{L}_{all}$*-def*
by *auto*

lemma *literals-are-*$\mathcal{L}_{in}$*-cong*:
‹*set-mset* $\mathcal{A}$ = *set-mset* $\mathcal{B}$ ⟹ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S* = *literals-are-*$\mathcal{L}_{in}$ $\mathcal{B}$ *S*›
using $\mathcal{L}_{all}$*-cong*[*of* $\mathcal{A}$ $\mathcal{B}$]
unfolding *literals-are-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*-def is-*$\mathcal{L}_{all}$*-def*
by *auto*

lemma *literals-are-$\mathcal{L}_{in}$'-literals-are-$\mathcal{L}_{in}$-iff*:
  assumes
    Sx: ‹$(S, x) \in$ *state-wl-l None*› **and**
    x-xa: ‹$(x, xa) \in$ *twl-st-l None*› **and**
    struct-invs: ‹*twl-struct-invs xa*›
  shows
    ‹*literals-are-$\mathcal{L}_{in}$' $\mathcal{A}$ S* $\longleftrightarrow$ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ S*› (**is** *?A*)
    ‹*literals-are-$\mathcal{L}_{in}$'* (*all-init-atms-st S*) *S* $\longleftrightarrow$ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*› (**is** *?B*)
    ‹*set-mset* (*all-init-atms-st S*) = *set-mset* (*all-atms-st S*)› (**is** *?C*)
proof −
  have ‹*cdcl$_W$-restart-mset.no-strange-atm* (*state$_W$-of xa*)›
    using *struct-invs* **unfolding** *twl-struct-invs-def cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv-def*
    by *fast+*
  then have ‹$\bigwedge L$. $L \in$ *atm-of* ‘ *lits-of-l* (*get-trail-wl S*) $\Longrightarrow L \in$ *atms-of-ms*
      ((λx. *mset* (*fst x*)) ‘ {*a*. *a* $\in\#$ *ran-m* (*get-clauses-wl S*) $\wedge$ *snd a*}) $\cup$
      atms-of-mm (*get-unit-init-clss-wl S*)› **and**
    alien-learned: ‹*atms-of-mm* (*learned-clss* (*state$_W$-of xa*))
      $\subseteq$ *atms-of-mm* (*init-clss* (*state$_W$-of xa*))›
    using *Sx x-xa* **unfolding** *cdcl$_W$-restart-mset.no-strange-atm-def*
    by (*auto simp add*: *twl-st twl-st-l twl-st-wl*)
  have *all-init-lits-alt-def*: ‹*all-lits-of-mm*
      ({#*mset* (*fst C*). *C* $\in\#$ *init-clss-l* (*get-clauses-wl S*)#} +
       get-unit-init-clss-wl S) = *all-init-lits-st S*›
    by (*auto simp*: *all-init-lits-def*)

  have *H*: ‹*set-mset*
     (*all-lits-of-mm*
       ({#*mset* (*fst C*). *C* $\in\#$ *init-clss-l* (*get-clauses-wl S*)#} +
        get-unit-init-clss-wl S)) = *set-mset*
     (*all-lits-of-mm*
       ({#*mset* (*fst C*). *C* $\in\#$ *ran-m* (*get-clauses-wl S*)#} +
        get-unit-clauses-wl S))›
    apply (*subst* (*2*) *all-clss-l-ran-m*[*symmetric*])
    using *alien-learned Sx x-xa*
    unfolding *image-mset-union all-lits-of-mm-union*
    by (*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff get-unit-clauses-wl-alt-def*
      twl-st twl-st-l twl-st-wl get-learned-clss-wl-def)
  show *A*: ‹*literals-are-$\mathcal{L}_{in}$' $\mathcal{A}$ S* $\longleftrightarrow$ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ S*› **for** $\mathcal{A}$
  proof −
    have ‹*is-$\mathcal{L}_{all}$ $\mathcal{A}$*
       (*all-lits-of-mm*
  ({#*mset C*. *C* $\in\#$ *init-clss-lf* (*get-clauses-wl S*)#} +
   get-unit-init-clss-wl S)) $\longleftrightarrow$
      is-$\mathcal{L}_{all}$ $\mathcal{A}$
       (*all-lits-of-mm*
  ({#*mset* (*fst C*). *C* $\in\#$ *ran-m* (*get-clauses-wl S*)#} +
   get-unit-clauses-wl S))›
      using *H* **unfolding** *is-$\mathcal{L}_{all}$-def* **by** *auto*
    moreover have ‹*set-mset* $\mathcal{A}'$ = *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$)› **if** ‹*is-$\mathcal{L}_{all}$ $\mathcal{A}$ $\mathcal{A}'$*› **for** $\mathcal{A}'$
      unfolding *that*[*unfolded is-$\mathcal{L}_{all}$-def*]
      by *auto*
    moreover have ‹*set-mset* ($\mathcal{L}_{all}$ (*all-atms-st S*)) = *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$)› **if** ‹*is-$\mathcal{L}_{all}$ $\mathcal{A}$* (*all-lits-st S*)›
      for $\mathcal{A}$ *S*
      unfolding *that*[*unfolded is-$\mathcal{L}_{all}$-def*]
      using ‹*set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *set-mset* (*all-lits-st S*)› *is-$\mathcal{L}_{all}$-all-lits-st-$\mathcal{L}_{all}$*(*1*) *that* **by** *blast*

moreover have ‹*set-mset* ($\mathcal{L}_{all}$ (*all-init-atms-st S*)) = *set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$)›
if ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-init-lits-st S*)› **for** $\mathcal{A}$ *S*
unfolding *that*[*unfolded is-*$\mathcal{L}_{all}$*-def*]
by (*metis* ‹*set-mset* ($\mathcal{L}_{all}$ $\mathcal{A}$) = *set-mset* (*all-init-lits-st S*)› *all-init-lits-def is-*$\mathcal{L}_{all}$*-*$\mathcal{L}_{all}$*-rewrite that*)
ultimately show *?thesis*
using *Sx x-xa* **unfolding** *cdcl*$_W$*-restart-mset.no-strange-atm-def literals-are-*$\mathcal{L}_{in}$*'-def*
literals-are-$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*'-def*
all-init-lits-def[*symmetric*] *all-lits-def*[*symmetric*] *all-init-lits-alt-def*
by (*auto 5 5 dest*: *multi-member-split*)
qed
show *C*: *?C*
unfolding *cdcl*$_W$*-restart-mset.no-strange-atm-def literals-are-*$\mathcal{L}_{in}$*'-def*
literals-are-$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*'-def all-atms-def all-init-atms-def*
all-init-lits-def all-lits-def all-init-lits-alt-def
by (*auto simp*: *H*)

show *?B*
by (*subst A*)
(*rule literals-are-*$\mathcal{L}_{in}$*-cong*[*OF C*])
qed

lemma *GC-remap-all-init-atmsD*:
‹*GC-remap* (*N*, *x*, *m*) (*N'*, *x'*, *m'*) $\Longrightarrow$ *all-init-atms N NE* + *all-init-atms m NE* = *all-init-atms N' NE* + *all-init-atms m' NE*›
by (*induction rule*: *GC-remap.induct*[*split-format*(*complete*)])
(*auto simp*: *all-init-atms-def all-init-lits-def init-clss-l-fmdrop-if*
init-clss-l-fmupd-if image-mset-remove1-mset-if
simp del: *all-init-atms-def*[*symmetric*]
simp flip: *image-mset-union all-lits-of-mm-add-mset all-lits-of-mm-union*)

lemma *rtranclp-GC-remap-all-init-atmsD*:
‹*GC-remap*** (*N*, *x*, *m*) (*N'*, *x'*, *m'*) $\Longrightarrow$ *all-init-atms N NE* + *all-init-atms m NE* = *all-init-atms N' NE* + *all-init-atms m' NE*›
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*: *GC-remap-all-init-atmsD*)

lemma *rtranclp-GC-remap-all-init-atms*:
‹*GC-remap*** (*x1a*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *x1ad*) $\Longrightarrow$ *all-init-atms x1ad NE* = *all-init-atms x1a NE*›
by (*auto dest*!: *rtranclp-GC-remap-all-init-atmsD*[*of - - - - - - NE*])

lemma *GC-remap-all-init-lits*:
‹*GC-remap* (*N*, *m*, *new*) (*N'*, *m'*, *new'*) $\Longrightarrow$ *all-init-lits N NE* + *all-init-lits new NE* = *all-init-lits N' NE* + *all-init-lits new' NE*›
by (*induction rule*: *GC-remap.induct*[*split-format*(*complete*)])
(*case-tac* ‹*irred N C*› ; *auto simp*: *all-init-lits-def init-clss-l-fmupd-if image-mset-remove1-mset-if*
simp flip: *all-lits-of-mm-union*)

lemma *rtranclp-GC-remap-all-init-lits*:
‹*GC-remap*** (*N*, *m*, *new*) (*N'*, *m'*, *new'*) $\Longrightarrow$ *all-init-lits N NE* + *all-init-lits new NE* = *all-init-lits N' NE* + *all-init-lits new' NE*›
by (*induction rule*: *rtranclp-induct*[*of r* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
(*auto dest*: *GC-remap-all-init-lits*)

lemma *cdcl-twl-restart-is-*$\mathcal{L}_{all}$:

assumes
ST: ‹*cdcl-twl-restart*** *S T*› **and**
struct-invs-S: ‹*twl-struct-invs S*› **and**
L: ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*clauses* (*get-clauses S*) + *unit-clss S*))›
shows ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*clauses* (*get-clauses T*) + *unit-clss T*))›
proof −
have ‹*twl-struct-invs T*›
using *rtranclp-cdcl-twl-restart-twl-struct-invs*[*OF ST struct-invs-S*] **.**
then have ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of T*)›
unfolding *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
by *fast+*
then have ‹*?thesis* ⟷ *is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*get-all-init-clss T*))›
unfolding *cdcl*$_W$*-restart-mset.no-strange-atm-def is-*$\mathcal{L}_{all}$*-alt-def*
by (*cases T*)
(*auto simp*: *cdcl*$_W$*-restart-mset-state*)
moreover have ‹*get-all-init-clss T* = *get-all-init-clss S*›
using *rtranclp-cdcl-twl-restart-get-all-init-clss*[*OF ST*] **.**
moreover {
have ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of S*)›
using *struct-invs-S*
unfolding *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
by *fast+*
then have ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*get-all-init-clss S*))›
using *L*
unfolding *cdcl*$_W$*-restart-mset.no-strange-atm-def is-*$\mathcal{L}_{all}$*-alt-def*
by (*cases S*)
(*auto simp*: *cdcl*$_W$*-restart-mset-state*)
}
ultimately show *?thesis*
by *argo*
qed

lemma *cdcl-twl-restart-is-*$\mathcal{L}_{all}$*'*:
assumes
ST: ‹*cdcl-twl-restart*** *S T*› **and**
struct-invs-S: ‹*twl-struct-invs S*› **and**
L: ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*get-all-init-clss S*))›
shows ‹*is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*get-all-init-clss T*))›
proof −
have ‹*twl-struct-invs T*›
using *rtranclp-cdcl-twl-restart-twl-struct-invs*[*OF ST struct-invs-S*] **.**
then have ‹*cdcl*$_W$*-restart-mset.no-strange-atm* (*state*$_W$*-of T*)›
unfolding *twl-struct-invs-def cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
by *fast+*
then have ‹*?thesis* ⟷ *is-*$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-lits-of-mm* (*get-all-init-clss T*))›
unfolding *cdcl*$_W$*-restart-mset.no-strange-atm-def is-*$\mathcal{L}_{all}$*-alt-def*
by (*cases T*)
(*auto simp*: *cdcl*$_W$*-restart-mset-state*)
moreover have ‹*get-all-init-clss T* = *get-all-init-clss S*›
using *rtranclp-cdcl-twl-restart-get-all-init-clss*[*OF ST*] **.**
then show *?thesis*
using *L*
by *argo*
qed

definition *remove-all-annot-true-clause-imp-wl-D-inv*
:: ⟨*nat twl-st-wl* ⇒ - ⇒ *nat* × *nat twl-st-wl* ⇒ *bool*⟩
where
⟨*remove-all-annot-true-clause-imp-wl-D-inv S xs* = (λ(*i*, *T*).
remove-all-annot-true-clause-imp-wl-inv S xs (*i*, *T*) ∧
literals-are-$\mathcal{L}_{in}$′ (*all-init-atms-st T*) *T* ∧
all-init-atms-st S = *all-init-atms-st T*)⟩

definition *remove-all-annot-true-clause-imp-wl-D-pre*
:: ⟨*nat multiset* ⇒ *nat literal* ⇒ *nat twl-st-wl* ⇒ *bool*⟩
where
⟨*remove-all-annot-true-clause-imp-wl-D-pre* $\mathcal{A}$ *L S* ⟷ (*L* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$ ∧ *literals-are-*$\mathcal{L}_{in}$′ $\mathcal{A}$ *S*)⟩

definition *remove-all-annot-true-clause-imp-wl-D*
:: ⟨*nat literal* ⇒ *nat twl-st-wl* ⇒ (*nat twl-st-wl*) *nres*⟩
where
⟨*remove-all-annot-true-clause-imp-wl-D* = (λ*L S*. *do* {
ASSERT(*remove-all-annot-true-clause-imp-wl-D-pre* (*all-init-atms-st S*)
L S);
let xs = *get-watched-wl S L*;
(-, *T*) ← *WHILE*$_T$$^{\lambda(i,\ T).}$ *remove-all-annot-true-clause-imp-wl-D-inv S xs* (*i*, *T*)
(λ(*i*, *T*). *i* < *length xs*)
(λ(*i*, *T*). *do* {
ASSERT(*i* < *length xs*);
let (*C*, - , -) = *xs* ! *i*;
if C ∈# *dom-m* (*get-clauses-wl T*) ∧ *length* ((*get-clauses-wl T*) ∝ *C*) ≠ *2*
then do {
T ← *remove-all-annot-true-clause-one-imp-wl* (*C*, *T*);
RETURN (*i*+*1*, *T*)
}
else
RETURN (*i*+*1*, *T*)
})
(*0*, *S*);
RETURN T
})⟩

lemma *is-*$\mathcal{L}_{all}$*-init-itself*[*iff*]:
⟨*is-*$\mathcal{L}_{all}$ (*all-init-atms x1h x2h*) (*all-init-lits x1h x2h*)⟩
unfolding *is-*$\mathcal{L}_{all}$*-def*
by (*auto simp*: *all-init-lits-def in-*$\mathcal{L}_{all}$*-atm-of-*$\mathcal{A}_{in}$
in-all-lits-of-mm-ain-atms-of-iff all-init-atms-def)

lemma *literals-are-*$\mathcal{L}_{in}$′*-alt-def*: ⟨*literals-are-*$\mathcal{L}_{in}$′ $\mathcal{A}$ *S* ⟷
is-$\mathcal{L}_{all}$ $\mathcal{A}$ (*all-init-lits* (*get-clauses-wl S*) (*get-unit-init-clss-wl S*)) ∧
blits-in-$\mathcal{L}_{in}$′ *S*⟩
unfolding *literals-are-*$\mathcal{L}_{in}$′*-def all-init-lits-def*
by *auto*

lemma *remove-all-annot-true-clause-imp-wl-remove-all-annot-true-clause-imp*:
⟨(*uncurry remove-all-annot-true-clause-imp-wl-D*, *uncurry remove-all-annot-true-clause-imp-wl*) ∈
{(*L*, *L*′). *L* = *L*′ ∧ *L* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$} $\times_f$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$′ $\mathcal{A}$ *S* ∧
$\mathcal{A}$ = *all-init-atms-st S*} $\to_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$′ $\mathcal{A}$ *S*}⟩*nres-rel*⟩
(**is** ⟨- ∈ - $\to_f$ ⟨*?R*⟩*nres-rel*⟩)
proof −

have [*refine0*]:
‹*remove-all-annot-true-clause-one-imp-wl* $(C, S) \leq$
$\Downarrow \{(S', T). (S', T) \in ?R \wedge$ *all-init-atms-st* $S' =$ *all-init-atms-st* $S\}$
(*remove-all-annot-true-clause-one-imp-wl* (C', S'))›
if ‹$(S, S') \in ?R$› **and** ‹$(C, C') \in Id$›
for $S\ S'\ C\ C'$
using *that* **unfolding** *remove-all-annot-true-clause-one-imp-def*
by (*cases S*)
(*fastforce simp*: *init-clss-l-fmdrop-irrelev init-clss-l-fmdrop*
image-mset-remove1-mset-if all-init-lits-def
remove-all-annot-true-clause-one-imp-wl-def
drop-clause-add-move-init-def
literals-are-$\mathcal{L}_{in}$*'-def*
blits-in-$\mathcal{L}_{in}$*'-def drop-clause-def*
all-init-atms-def
dest!: *multi-member-split*[*of* - ‹$\mathcal{L}_{all}\ \mathcal{A}$›])

show *?thesis*
supply [[*goals-limit=1*]]
unfolding *uncurry-def remove-all-annot-true-clause-imp-wl-D-def*
remove-all-annot-true-clause-imp-wl-def
apply (*intro frefI nres-relI*)
subgoal for *x y*
apply (*refine-vcg*
WHILEIT-refine[**where** $R =$ ‹$\{((i, S), (i', S')).\ i = i' \wedge (S, S') \in Id \wedge$
literals-are-$\mathcal{L}_{in}$*'* $\mathcal{A}\ S' \wedge$ *all-init-atms-st* (*snd x*) $=$ *all-init-atms-st* $S\}$›])
subgoal unfolding *remove-all-annot-true-clause-imp-wl-D-pre-def*
by (*auto simp flip*: *all-init-atms-def*)
subgoal by *auto*
subgoal
unfolding *remove-all-annot-true-clause-imp-wl-D-inv-def all-init-atms-def*
by (*auto simp flip*: *all-atms-def simp*: *literals-are-*$\mathcal{L}_{in}$*'-alt-def*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *remove-all-annot-true-clause-imp-wl-D-inv-def literals-are-*$\mathcal{L}_{in}$*'-alt-def*
blits-in-$\mathcal{L}_{in}$*-def*
by (*auto simp flip*: *all-atms-def simp*: *blits-in-*$\mathcal{L}_{in}$*'-def*)
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *remove-all-annot-true-clause-imp-wl-D-pre-def* **by** *auto*
done
done
qed

definition *remove-one-annot-true-clause-one-imp-wl-D-pre* **where**
‹*remove-one-annot-true-clause-one-imp-wl-D-pre* $i\ T \longleftrightarrow$
remove-one-annot-true-clause-one-imp-wl-pre $i\ T \wedge$
literals-are-$\mathcal{L}_{in}$*'* (*all-init-atms-st* T) T›

definition *remove-one-annot-true-clause-one-imp-wl-D*
:: ‹*nat* $\Rightarrow$ *nat twl-st-wl* $\Rightarrow$ (*nat* $\times$ *nat twl-st-wl*) *nres*›
where
‹*remove-one-annot-true-clause-one-imp-wl-D* $=$ ($\lambda i\ S.$ *do* {
ASSERT(*remove-one-annot-true-clause-one-imp-wl-D-pre* $i\ S$);

$ASSERT(is\text{-}proped\ (rev\ (get\text{-}trail\text{-}wl\ S)\ !\ i));$
$(L,\ C) \leftarrow SPEC(\lambda(L,\ C).\ (rev\ (get\text{-}trail\text{-}wl\ S))!i = Propagated\ L\ C);$
$ASSERT(Propagated\ L\ C \in set\ (get\text{-}trail\text{-}wl\ S));$
$ASSERT(atm\text{-}of\ L \in\#\ all\text{-}init\text{-}atms\text{-}st\ S);$
if $C = 0$ *then* $RETURN\ (i+1,\ S)$
else do {
$ASSERT(C \in\#\ dom\text{-}m\ (get\text{-}clauses\text{-}wl\ S));$
$T \leftarrow replace\text{-}annot\text{-}l\ L\ C\ S;$
$ASSERT(get\text{-}clauses\text{-}wl\ S = get\text{-}clauses\text{-}wl\ T);$
$T \leftarrow remove\text{-}and\text{-}add\text{-}cls\text{-}l\ C\ T;$
— $S \leftarrow remove\text{-}all\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}wl\ L\ S;$
$RETURN\ (i+1,\ T)$
}
})›

lemma *remove-one-annot-true-clause-one-imp-wl-pre-in-trail-in-all-init-atms-st*:
assumes
inv: ‹*remove-one-annot-true-clause-one-imp-wl-D-pre K S*› **and**
LC-tr: ‹$Propagated\ L\ C \in set\ (get\text{-}trail\text{-}wl\ S)$›
shows ‹$atm\text{-}of\ L \in\#\ all\text{-}init\text{-}atms\text{-}st\ S$›
proof −
obtain *x xa* **where**
Sx: ‹$(S,\ x) \in state\text{-}wl\text{-}l\ None$› **and**
‹$correct\text{-}watching''\ S$› **and**
‹*twl-list-invs x*› **and**
‹$K < length\ (get\text{-}trail\text{-}l\ x)$› **and**
‹*twl-list-invs x*› **and**
‹$get\text{-}conflict\text{-}l\ x = None$› **and**
‹$clauses\text{-}to\text{-}update\text{-}l\ x = \{\#\}$› **and**
x-xa: ‹$(x,\ xa) \in twl\text{-}st\text{-}l\ None$› **and**
struct: ‹*twl-struct-invs xa*›
using *inv*
unfolding *remove-one-annot-true-clause-one-imp-wl-pre-def*
remove-one-annot-true-clause-one-imp-pre-def
remove-one-annot-true-clause-one-imp-wl-D-pre-def
by *blast*
have ‹$L \in lits\text{-}of\text{-}l\ (trail\ (state_W\text{-}of\ xa))$›
using *LC-tr Sx x-xa*
by (*force simp*: *twl-st twl-st-l twl-st-wl lits-of-def*)
moreover have ‹$cdcl_W\text{-}restart\text{-}mset.no\text{-}strange\text{-}atm\ (state_W\text{-}of\ xa)$›
using *struct* **unfolding** *twl-struct-invs-def*
$cdcl_W\text{-}restart\text{-}mset.cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$
by *fast*
ultimately have ‹$atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (init\text{-}clss\ (state_W\text{-}of\ xa))$›
unfolding $cdcl_W\text{-}restart\text{-}mset.no\text{-}strange\text{-}atm\text{-}def$
by (*auto simp*: *twl-st twl-st-l twl-st-wl*)
then have ‹$atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (mset\ `\#\ (init\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}wl\ S)) +$
$get\text{-}unit\text{-}init\text{-}clss\text{-}wl\ S)$›
using *Sx x-xa*
unfolding $cdcl_W\text{-}restart\text{-}mset.no\text{-}strange\text{-}atm\text{-}def$
by (*auto simp*: *twl-st twl-st-l twl-st-wl*)
then show *?thesis*
by (*auto simp*: *all-init-atms-alt-def*)
qed

lemma *remove-one-annot-true-clause-one-imp-wl-D-remove-one-annot-true-clause-one-imp-wl*:
‹(*uncurry remove-one-annot-true-clause-one-imp-wl-D*,
uncurry remove-one-annot-true-clause-one-imp-wl) ∈
nat-rel $\times_f$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$′ (*all-init-atms-st S*) *S*} $\to_f$
⟨*nat-rel* $\times_f$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$′ (*all-init-atms-st S*) *S*}⟩*nres-rel*›
(**is** ‹- ∈ - $\times_f$ *?A* $\to_f$ -›)
proof −
have [*refine0*]: ‹*replace-annot-l L C S* ≤
⇓ {(*S*′, *T*′). (*S*′, *T*′) ∈ *?A* ∧ *get-clauses-wl S*′ = *get-clauses-wl S*} (*replace-annot-l L*′ *C*′ *T*′)›
if ‹(*L*, *L*′) ∈ *Id*› **and** ‹(*S*, *T*′) ∈ *?A*› **and** ‹(*C*, *C*′) ∈ *Id*› **for** *L L*′ *S T*′ *C C*′
using *that*
by (*cases S*; *cases T*′)
(*fastforce simp*: *replace-annot-l-def state-wl-l-def*
literals-are-$\mathcal{L}_{in}$′*-def blits-in-*$\mathcal{L}_{in}$′*-def*
intro: *RES-refine*)
have [*simp*]: ‹*all-init-atms* (*fmdrop C*′ *x1a*) (*add-mset* (*mset* (*x1a* ∝ *C*′)) *x1c*) =
all-init-atms x1a x1c›
if ‹*irred x1a C*′› **and** ‹*C*′ ∈# *dom-m x1a*›
for *C*′ *x1a x1c*
using *that*
by (*auto simp*: *all-init-atms-def all-init-lits-def*
image-mset-remove1-mset-if)
have [*simp*]: ‹*all-init-atms* (*fmdrop C*′ *x1a*) *x1c* =
all-init-atms x1a x1c›
if ‹¬*irred x1a C*′›
for *C*′ *x1a x1c*
using *that*
by (*auto simp*: *all-init-atms-def all-init-lits-def*
image-mset-remove1-mset-if)
have [*refine0*]: ‹*remove-and-add-cls-l C S* ≤⇓ *?A* (*remove-and-add-cls-l C*′ *S*′)›
if ‹(*C*, *C*′) ∈ *Id*› **and** ‹(*S*, *S*′) ∈ *?A*› **and**
‹*C* ∈# *dom-m* (*get-clauses-wl S*)›
for *C C*′ *S S*′
using *that* **unfolding** *remove-and-add-cls-l-def*
by *refine-rcg*
(*auto intro*!: *RES-refine simp*: *state-wl-l-def init-clss-l-fmdrop*
literals-are-$\mathcal{L}_{in}$′*-def blits-in-*$\mathcal{L}_{in}$′*-def image-mset-remove1-mset-if*
init-clss-l-fmdrop-irrelev)
show *?thesis*
supply [[*goals-limit=1*]]
unfolding *uncurry-def remove-one-annot-true-clause-one-imp-wl-def*
remove-one-annot-true-clause-one-imp-wl-D-def
apply (*intro frefI nres-relI*)
subgoal for *x y*
apply (*refine-vcg*
remove-all-annot-true-clause-imp-wl-remove-all-annot-true-clause-imp[
of ‹*all-init-atms-st* (*snd x*)›,
THEN fref-to-Down-curry])
subgoal unfolding *remove-one-annot-true-clause-one-imp-wl-D-pre-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *K S K*′ *S*′ *LC LC*′ *L C L*′ *C*′
by (*rule remove-one-annot-true-clause-one-imp-wl-pre-in-trail-in-all-init-atms-st*)
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
done
qed

definition *remove-one-annot-true-clause-imp-wl-D-inv* **where**
‹*remove-one-annot-true-clause-imp-wl-D-inv* $S = (\lambda(i,\ T).$
remove-one-annot-true-clause-imp-wl-inv $S\ (i,\ T) \wedge$
literals-are-$\mathcal{L}_{in}'$ (*all-init-atms-st* T) T)›

definition *remove-one-annot-true-clause-imp-wl-D* :: ‹*nat twl-st-wl* ⇒ (*nat twl-st-wl*) *nres*›
where
‹*remove-one-annot-true-clause-imp-wl-D* = (λS. *do* {
$k \leftarrow$ *SPEC*(λk. ($\exists$ *M1 M2 K*. (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*get-trail-wl* S)) ∧
count-decided M1 = *0* ∧ k = *length M1*)
∨ (*count-decided* (*get-trail-wl* S) = *0* ∧ k = *length* (*get-trail-wl* S)));
(-, S) ← $\mathit{WHILE}_T{}^{\mathit{remove\text{-}one\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}wl\text{-}D\text{-}inv}\ S}$
($\lambda(i,\ S).\ i < k$)
($\lambda(i,\ S)$. *remove-one-annot-true-clause-one-imp-wl-D* i S)
(*0*, S);
RETURN S
})›

lemma *remove-one-annot-true-clause-imp-wl-D-remove-one-annot-true-clause-imp-wl*:
‹(*remove-one-annot-true-clause-imp-wl-D*, *remove-one-annot-true-clause-imp-wl*) ∈
{(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st* S) S} $\rightarrow_f$
⟨{(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st* S) S}⟩*nres-rel*›
proof −
show *?thesis*
unfolding *uncurry-def remove-one-annot-true-clause-imp-wl-D-def*
remove-one-annot-true-clause-imp-wl-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
WHILEIT-refine[**where** R=‹*nat-rel* $\times_r$ {(S, T). (S, T) ∈ *Id* ∧
literals-are-$\mathcal{L}_{in}'$ (*all-init-atms-st* S) S}›]
remove-one-annot-true-clause-one-imp-wl-D-remove-one-annot-true-clause-one-imp-wl[*THEN fref-to-Down-curry*])
subgoal by *auto*
subgoal by *auto*
subgoal for $S\ S'\ k\ k'\ T\ T'$
by (*cases* T') (*auto simp*: *remove-one-annot-true-clause-imp-wl-D-inv-def*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *mark-to-delete-clauses-wl-D-pre* **where**
‹*mark-to-delete-clauses-wl-D-pre S* ⟷
mark-to-delete-clauses-wl-pre S ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*›

definition *mark-to-delete-clauses-wl-D-inv* **where**
‹*mark-to-delete-clauses-wl-D-inv* = (λ*S xs0* (*i*, *T*, *xs*).
mark-to-delete-clauses-wl-inv S xs0 (*i*, *T*, *xs*) ∧
literals-are-$\mathcal{L}_{in}$*'* (*all-init-atms-st T*) *T*)›

definition *mark-to-delete-clauses-wl-D* :: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*› **where**
‹*mark-to-delete-clauses-wl-D* = (λ*S*. *do* {
  ASSERT(*mark-to-delete-clauses-wl-D-pre S*);
  xs ← *collect-valid-indices-wl S*;
  l ← *SPEC*(λ-::*nat*. *True*);
  (-, *S*, *xs*) ← *WHILE*$_T$$^{mark\text{-}to\text{-}delete\text{-}clauses\text{-}wl\text{-}D\text{-}inv\ S\ xs}$
    (λ(*i*, -, *xs*). *i* < *length xs*)
    (λ(*i*, *T*, *xs*). *do* {
      if(*xs*!*i* ∉# *dom-m* (*get-clauses-wl T*)) *then RETURN* (*i*, *T*, *delete-index-and-swap xs i*)
      else do {
        ASSERT(*0* < *length* (*get-clauses-wl T*∝(*xs*!*i*)));
        ASSERT(*get-clauses-wl T*∝(*xs*!*i*)!*0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T*));
        can-del ← *SPEC*(λ*b*. *b* ⟶
          (*Propagated* (*get-clauses-wl T*∝(*xs*!*i*)!*0*) (*xs*!*i*) ∉ *set* (*get-trail-wl T*)) ∧
          ¬*irred* (*get-clauses-wl T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl T*∝(*xs*!*i*)) ≠ *2*);
        ASSERT(*i* < *length xs*);
        if can-del
        then
          RETURN (*i*, *mark-garbage-wl* (*xs*!*i*) *T*, *delete-index-and-swap xs i*)
        else
          RETURN (*i*+*1*, *T*, *xs*)
      }
    })
    (*l*, *S*, *xs*);
  RETURN S
})›

lemma *mark-to-delete-clauses-wl-D-mark-to-delete-clauses-wl*:
‹(*mark-to-delete-clauses-wl-D*, *mark-to-delete-clauses-wl*) ∈
{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*} →$_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*}⟩*nres-rel*›
proof −
have [*refine0*]: ‹*collect-valid-indices-wl S* ≤ ⇓ *Id* (*collect-valid-indices-wl T*)›
if ‹(*S*, *T*) ∈ *Id*› **for** *S T*
using *that* **by** *auto*
have [*iff*]: ‹(∀ (*x*::*bool*) *xa*. *P x xa*) ⟷ (∀ *xa*.(*P True xa* ∧ *P False xa*))› **for** *P*
by *metis*
have *in-Lit*: ‹*get-clauses-wl T'* ∝ (*xs* ! *j*) ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T'*)›
if
‹(*l*, *l'*) ∈ *nat-rel*› **and**
rel: ‹(*st*, *st'*) ∈ *nat-rel* ×$_r$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*} ×$_r$
Id› **and**
inv-x: ‹*mark-to-delete-clauses-wl-D-inv S ys st*› **and**
‹*mark-to-delete-clauses-wl-inv S' ys' st'*› **and**
dom: ‹¬ *xs* ! *j* ∉# *dom-m* (*get-clauses-wl T'*)› **and**
‹¬ *xs'* ! *i* ∉# *dom-m* (*get-clauses-wl T*)› **and**
assert: ‹*0* < *length* (*get-clauses-wl T* ∝ (*xs'* ! *i*))› **and**

st: ‹*st′* = (*i*, *sT*)› ‹*sT* = (*T*, *xs*)› ‹*st* = (*j*, *sT′*)› ‹*sT′* = (*T′*, *xs′*)› **and**
le: ‹*case st of* (*i*, *T*, *xs*) ⇒ *i* < *length xs*› **and**
‹*case st′ of* (*i*, *S*, *xs′*) ⇒ *i* < *length xs′*› **and**
‹*0* < *length* (*get-clauses-wl T′* ∝ (*xs* ! *j*))›
for *S S′ xs′ l l′ st st′ i T j T′ sT xs sT′ ys ys′*
proof −

have *lits-T′*: ‹*literals-are-*$\mathcal{L}_{in}$*′* (*all-init-atms-st T′*) *T′*›
using *inv-x* **unfolding** *mark-to-delete-clauses-wl-D-inv-def prod.simps st*
by *fast*
have ‹*literals-are-*$\mathcal{L}_{in}$ (*all-init-atms-st T*) *T*›
proof −
obtain *x xa xb* **where**
lits-T′: ‹*literals-are-*$\mathcal{L}_{in}$*′* (*all-init-atms-st T′*) *T′*› **and**
Ta-x: ‹(*S*, *x*) ∈ *state-wl-l None*› **and**
Ta-y: ‹(*T′*, *xa*) ∈ *state-wl-l None*› **and**
‹*correct-watching′ S*› **and**
rem: ‹*remove-one-annot-true-clause*** *x xa*› **and**
list: ‹*twl-list-invs x*› **and**
x-xb: ‹(*x*, *xb*) ∈ *twl-st-l None*› **and**
struct: ‹*twl-struct-invs xb*› **and**
confl: ‹*get-conflict-l x* = *None*› **and**
upd: ‹*clauses-to-update-l x* = {#}›
using *inv-x* **unfolding** *mark-to-delete-clauses-wl-D-inv-def st prod.case mark-to-delete-clauses-wl-inv-def mark-to-delete-clauses-l-inv-def*
by *blast*

obtain *y* **where**
Ta-y′: ‹(*xa*, *y*) ∈ *twl-st-l None*› **and**
struct′: ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF rem list confl upd x-xb struct*] **by** *blast*

have ‹*literals-are-*$\mathcal{L}_{in}$ (*all-init-atms-st T′*) *T′*›
by (*rule literals-are-*$\mathcal{L}_{in}$*′-literals-are-*$\mathcal{L}_{in}$*-iff*(*1*)[*THEN iffD1*, *OF Ta-y Ta-y′ struct′ lits-T′*])
then show *?thesis*
using *rel* **by** (*auto simp*: *st*)
qed
then have ‹*literals-are-in-*$\mathcal{L}_{in}$ (*all-init-atms-st T′*) (*mset* (*get-clauses-wl T′* ∝ (*xs* ! *j*)))›
using *literals-are-in-*$\mathcal{L}_{in}$*-nth*[*of* ‹*xs*!*i*› ‹*T*›] *rel dom*
by (*auto simp*: *st*)
then show *?thesis*
by (*rule literals-are-in-*$\mathcal{L}_{in}$*-in-*$\mathcal{L}_{all}$) (*use le assert rel dom* **in** ‹*auto simp*: *st*›)
qed
have *final-rel-del*:
‹((*j*, *mark-garbage-wl* (*xs* ! *j*) *T′*, *delete-index-and-swap xs j*),
i, *mark-garbage-wl* (*xs′* ! *i*) *T*, *delete-index-and-swap xs′ i*)
∈ *nat-rel* $\times_r$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*′* (*all-init-atms-st S*) *S*}$\times_r$ *Id*›
if
rel: ‹(*st*, *st′*) ∈ *nat-rel* $\times_r$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*′* (*all-init-atms-st T*) *S*} $\times_r$ *Id*› **and**
‹*case st of* (*i*, *T*, *xs*) ⇒ *i* < *length xs*› **and**
‹*case st′ of* (*i*, *S*, *xs′*) ⇒ *i* < *length xs′*› **and**
inv: ‹*mark-to-delete-clauses-wl-D-inv S ys st*› **and**

⟨*mark-to-delete-clauses-wl-inv* S' *ys'* *st'*⟩ **and**
st: ⟨$st' = (i, sT)$⟩ ⟨$sT = (T, xs)$⟩ ⟨$st = (j, sT')$⟩ ⟨$sT' = (T', xs')$⟩ **and**
dom: ⟨¬ *xs* ! *j* ∉# *dom-m* (*get-clauses-wl* T')⟩ **and**
⟨¬ *xs'* ! *i* ∉# *dom-m* (*get-clauses-wl* T)⟩ **and**
le: ⟨*0* < *length* (*get-clauses-wl* T ∝ (*xs'* ! *i*))⟩ **and**
⟨*0* < *length* (*get-clauses-wl* T' ∝ (*xs* ! *j*))⟩ **and**
⟨*get-clauses-wl* T' ∝ (*xs* ! *j*) ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st* T')⟩ **and**
⟨(*can-del*, *can-del'*) ∈ *bool-rel*⟩ **and**
can-del: ⟨*can-del*
∈ {*b*. *b* ⟶
Propagated (*get-clauses-wl* T' ∝ (*xs* ! *j*) ! *0*) (*xs* ! *j*)
∉ *set* (*get-trail-wl* T') ∧
¬ *irred* (*get-clauses-wl* T') (*xs* ! *j*)}⟩ **and**
⟨*can-del'*
∈ {*b*. *b* ⟶
Propagated (*get-clauses-wl* T ∝ (*xs'* ! *i*) ! *0*) (*xs'* ! *i*)
∉ *set* (*get-trail-wl* T) ∧
¬ *irred* (*get-clauses-wl* T) (*xs'* ! *i*)}⟩ **and**
i-le: ⟨*i* < *length xs'*⟩ **and**
⟨*j* < *length xs*⟩ **and**
[*simp*]: ⟨*can-del*⟩ **and**
[*simp*]: ⟨*can-del'*⟩
for S S' *xs xs'* *l l'* *st st'* *i* T *j* T' *can-del can-del'* *sT sT'* *ys ys'*
proof −
have ⟨*literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st* (*mark-garbage-wl* (*xs'* ! *i*) T)) (*mark-garbage-wl* (*xs'* ! *i*) T)⟩
using *can-del inv rel* **unfolding** *mark-to-delete-clauses-wl-D-inv-def st mark-garbage-wl-def*
by (*cases* T)
(*auto simp*: *literals-are-*$\mathcal{L}_{in}$*'-def init-clss-l-fmdrop-irrelev mset-take-mset-drop-mset'*
blits-in-$\mathcal{L}_{in}$*'-def all-init-lits-def*)
then show *?thesis*
using *inv rel* **unfolding** *st*
by *auto*
qed

show *?thesis*
unfolding *uncurry-def mark-to-delete-clauses-wl-D-def mark-to-delete-clauses-wl-def*
collect-valid-indices-wl-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
WHILEIT-refine[**where**
R = ⟨*nat-rel* $\times_r$ {(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st* S) S} $\times_r$ *Id*⟩])
subgoal
unfolding *mark-to-delete-clauses-wl-D-pre-def* **by** *auto*
subgoal by *auto*
subgoal for *x y xs xsa l la xa x'*
unfolding *mark-to-delete-clauses-wl-D-inv-def* **by** (*cases x'*) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for S S' *xs xs'* *l l'* *st st'* *i* T *j* T'
by (*rule in-Lit*; *assumption?*) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal for $S\ S'\ xs\ xs'\ l\ l'\ st\ st'\ i\ T\ j\ T'$ *can-del can-del'*
by (*rule final-rel-del*; *assumption?*) *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *mark-to-delete-clauses-wl-D-post* **where**
‹*mark-to-delete-clauses-wl-D-post S T* ⟷
(*mark-to-delete-clauses-wl-post S T* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st S*) *S*)›

definition *cdcl-twl-full-restart-wl-prog-D* :: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*› **where**
‹*cdcl-twl-full-restart-wl-prog-D S = do* {
— *S* ← *remove-one-annot-true-clause-imp-wl-D S*;
ASSERT(*mark-to-delete-clauses-wl-D-pre S*);
T ← *mark-to-delete-clauses-wl-D S*;
ASSERT (*mark-to-delete-clauses-wl-post S T*);
RETURN T
}›

lemma *cdcl-twl-full-restart-wl-prog-D-final-rel*:
assumes
‹(*S*, *Sa*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*}› **and**
‹*mark-to-delete-clauses-wl-D-pre S*› **and**
‹(*T*, *Ta*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st S*) *S*}› **and**
post: ‹*mark-to-delete-clauses-wl-post Sa Ta*› **and**
‹*mark-to-delete-clauses-wl-post S T*›
shows ‹(*T*, *Ta*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*}›
proof −
have *lits-T*: ‹*literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st Ta*) *Ta*› **and** *T*: ‹*T = Ta*›
using *assms* **by** *auto*
obtain *x xa xb* **where**
Sa-x: ‹(*Sa*, *x*) ∈ *state-wl-l None*› **and**
Ta-x: ‹(*Ta*, *xa*) ∈ *state-wl-l None*› **and**
corr-S: ‹*correct-watching Sa*› **and**
corr-T: ‹*correct-watching Ta*› **and**
x-xb: ‹(*x*, *xb*) ∈ *twl-st-l None*› **and**
rem: ‹*remove-one-annot-true-clause*** *x xa*› **and**
list: ‹*twl-list-invs x*› **and**
struct: ‹*twl-struct-invs xb*› **and**
confl: ‹*get-conflict-l x = None*› **and**
upd: ‹*clauses-to-update-l x* = {#}›
using *post* **unfolding** *mark-to-delete-clauses-wl-post-def mark-to-delete-clauses-l-post-def*
by *blast*
obtain *y* **where**
‹*cdcl-twl-restart-l*** *x xa*› **and**
Ta-y: ‹(*xa*, *y*) ∈ *twl-st-l None*› **and**
‹*cdcl-twl-restart*** *xb y*› **and**
struct: ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF rem list confl upd x-xb struct*]
by *blast*

have ‹*literals-are-*$\mathcal{L}_{in}$ (*all-atms-st Ta*) *Ta*›
by (*rule literals-are-*$\mathcal{L}_{in}'$*-literals-are-*$\mathcal{L}_{in}$*-iff*(*2*)[*THEN iffD1*,

OF Ta-x Ta-y struct lits-T])
then show *?thesis*
using *lits-T T* **by** *auto*
qed

lemma *mark-to-delete-clauses-wl-pre-lits'*:
‹$(S, T) \in \{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ *(all-atms-st S) S*$\} \Longrightarrow$
mark-to-delete-clauses-wl-pre T $\Longrightarrow$ *mark-to-delete-clauses-wl-D-pre S*›
unfolding *mark-to-delete-clauses-wl-D-pre-def mark-to-delete-clauses-wl-pre-def*
apply *normalize-goal+*
apply (*intro conjI*)
subgoal for *U*
by (*rule exI*[*of - U*]) *auto*
subgoal for *U*
unfolding *mark-to-delete-clauses-l-pre-def*
apply *normalize-goal+*
by (*subst literals-are-*$\mathcal{L}_{in}$*'-literals-are-*$\mathcal{L}_{in}$*-iff*(*2*)[*of - U*]) *auto*
done

lemma *cdcl-twl-full-restart-wl-prog-D-cdcl-twl-restart-wl-prog*:
‹(*cdcl-twl-full-restart-wl-prog-D*, *cdcl-twl-full-restart-wl-prog*) $\in$
$\{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*$\} \rightarrow_f$
$\langle\{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*$\}\rangle$*nres-rel*›
proof −
show *?thesis*
unfolding *uncurry-def cdcl-twl-full-restart-wl-prog-D-def cdcl-twl-full-restart-wl-prog-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
mark-to-delete-clauses-wl-D-mark-to-delete-clauses-wl[*THEN fref-to-Down*])
subgoal for *S T*
by (*rule mark-to-delete-clauses-wl-pre-lits'*)
subgoal for *S T*
unfolding *mark-to-delete-clauses-wl-D-pre-def* **by** *blast*
subgoal by *auto*
subgoal for *x y S Sa*
by (*rule cdcl-twl-full-restart-wl-prog-D-final-rel*)
done
qed

definition *restart-abs-wl-D-pre* :: ‹*nat twl-st-wl* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-abs-wl-D-pre S brk* ⟷
(*restart-abs-wl-pre S brk* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*)›

definition *cdcl-twl-local-restart-wl-D-spec*
:: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*›
where
‹*cdcl-twl-local-restart-wl-D-spec* = (λ(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
ASSERT(*restart-abs-wl-D-pre* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) *False*);
(*M*, *Q'*) ← *SPEC*(λ(*M'*, *Q'*). (∃ *K M2*. (*Decided K* # *M'*, *M2*) ∈ *set* (*get-all-ann-decomposition*
M) ∧
Q' = {#}) ∨ (*M'* = *M* ∧ *Q'* = *Q*));
RETURN (*M*, *N*, *D*, *NE*, *UE*, *Q'*, *W*)
})›

lemma *cdcl-twl-local-restart-wl-D-spec-cdcl-twl-local-restart-wl-spec*:
‹(*cdcl-twl-local-restart-wl-D-spec*, *cdcl-twl-local-restart-wl-spec*)

$\in [\lambda S.\ restart\text{-}abs\text{-}wl\text{-}D\text{-}pre\ S\ False]_f\ \{(S,\ T).\ (S,\ T) \in Id \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ (all\text{-}atms\text{-}st\ S)\ S\} \rightarrow$
$\langle\{(S,\ T).\ (S,\ T) \in Id \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ (all\text{-}atms\text{-}st\ S)\ S\}\rangle nres\text{-}rel\rangle$
proof –
show *?thesis*
unfolding *cdcl-twl-local-restart-wl-D-spec-def cdcl-twl-local-restart-wl-spec-def rewatch-clauses-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def correct-watching.simps clause-to-update-def literals-are-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*-def all-atms-def*[*symmetric*])
done
qed

definition *cdcl-twl-restart-wl-D-prog* **where**
‹*cdcl-twl-restart-wl-D-prog S* = *do* {
b ← *SPEC*(λ-. *True*);
if b then cdcl-twl-local-restart-wl-D-spec S else cdcl-twl-full-restart-wl-prog-D S
}›

lemma *cdcl-twl-restart-wl-D-prog-final-rel*:
assumes
post: ‹*restart-abs-wl-D-pre Sa b*› **and**
‹$(S,\ Sa) \in \{(S,\ T).\ (S,\ T) \in Id \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ (all\text{-}atms\text{-}st\ S)\ S\}$›
shows ‹$(S,\ Sa) \in \{(S,\ T).\ (S,\ T) \in Id \wedge literals\text{-}are\text{-}\mathcal{L}_{in}{}'\ (all\text{-}init\text{-}atms\text{-}st\ S)\ S\}$›
proof –
have *lits-T*: ‹*literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*› **and** *T*: ‹*S* = *Sa*›
using *assms* **by** *auto*
obtain *x xa* **where**
‹*literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st S*) *S*› **and**
S-x: ‹(*S*, *x*) ∈ *state-wl-l None*› **and**
‹*correct-watching S*› **and**
x-xa: ‹(*x*, *xa*) ∈ *twl-st-l None*› **and**
struct: ‹*twl-struct-invs xa*› **and**
list: ‹*twl-list-invs x*› **and**
‹*clauses-to-update-l x* = {#}› **and**
‹*twl-stgy-invs xa*› **and**
confl: ‹¬*b* ⟶ *get-conflict xa* = *None*›
using *post* **unfolding** *restart-abs-wl-D-pre-def restart-abs-wl-pre-def restart-abs-l-pre-def restart-prog-pre-def T* **by** *blast*

have ‹*literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st S*) *S*›
by (*rule literals-are-*$\mathcal{L}_{in}{}'$*-literals-are-*$\mathcal{L}_{in}$*-iff*(*2*)[*THEN iffD2*, *OF S-x x-xa struct lits-T*])
then show *?thesis*
using *T* **by** *auto*
qed

lemma *cdcl-twl-restart-wl-D-prog-cdcl-twl-restart-wl-prog*:
‹(*cdcl-twl-restart-wl-D-prog*, *cdcl-twl-restart-wl-prog*)
$\in [\lambda S.\ restart\text{-}abs\text{-}wl\text{-}D\text{-}pre\ S\ False]_f\ \{(S,\ T).\ (S,\ T) \in Id \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ (all\text{-}atms\text{-}st\ S)\ S\} \rightarrow$
$\langle\{(S,\ T).\ (S,\ T) \in Id \wedge literals\text{-}are\text{-}\mathcal{L}_{in}\ (all\text{-}atms\text{-}st\ S)\ S\}\rangle nres\text{-}rel\rangle$
unfolding *cdcl-twl-restart-wl-D-prog-def cdcl-twl-restart-wl-prog-def rewatch-clauses-def*
apply (*intro frefI nres-relI*)

apply (*refine-vcg*
cdcl-twl-local-restart-wl-D-spec-cdcl-twl-local-restart-wl-spec[*THEN fref-to-Down*]
cdcl-twl-full-restart-wl-prog-D-cdcl-twl-restart-wl-prog[*THEN fref-to-Down*])
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal for $x\ y\ b\ b'$
by *auto*
done

context *twl-restart-ops*
begin

definition *mark-to-delete-clauses-wl2-D-inv* **where**
‹*mark-to-delete-clauses-wl2-D-inv* = (λ*S xs0* (*i*, *T*, *xs*).
mark-to-delete-clauses-wl2-inv S xs0 (*i*, *T*, *xs*) ∧
literals-are-$\mathcal{L}_{in}{}'$ (*all-init-atms-st T*) *T*)›

definition *mark-to-delete-clauses-wl2-D* :: ‹*nat twl-st-wl* ⇒ *nat twl-st-wl nres*› **where**
‹*mark-to-delete-clauses-wl2-D* = (λ*S*. *do* {
ASSERT(*mark-to-delete-clauses-wl-D-pre S*);
xs ← *collect-valid-indices-wl S*;
l ← *SPEC*(λ-::*nat*. *True*);
(-, *S*, *xs*) ← $WHILE_T{}^{mark\text{-}to\text{-}delete\text{-}clauses\text{-}wl2\text{-}D\text{-}inv\ S\ xs}$
(λ(*i*, -, *xs*). *i* < *length xs*)
(λ(*i*, *T*, *xs*). *do* {
if(*xs*!*i* ∉# *dom-m* (*get-clauses-wl T*)) *then RETURN* (*i*, *T*, *delete-index-and-swap xs i*)
else do {
ASSERT(*0* < *length* (*get-clauses-wl T*∝(*xs*!*i*)));
ASSERT(*get-clauses-wl T*∝(*xs*!*i*)!*0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T*));
can-del ← *SPEC*(λ*b*. *b* ⟶
(*Propagated* (*get-clauses-wl T*∝(*xs*!*i*)!*0*) (*xs*!*i*) ∉ *set* (*get-trail-wl T*)) ∧
¬*irred* (*get-clauses-wl T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl T*∝(*xs*!*i*)) ≠ *2*);
ASSERT(*i* < *length xs*);
if can-del
then
RETURN (*i*, *mark-garbage-wl* (*xs*!*i*) *T*, *delete-index-and-swap xs i*)
else
RETURN (*i*+*1*, *T*, *xs*)
}
})
(*l*, *S*, *xs*);
RETURN S
})›

lemma *mark-to-delete-clauses-wl-D-mark-to-delete-clauses-wl2*:
‹(*mark-to-delete-clauses-wl2-D*, *mark-to-delete-clauses-wl2*) ∈
{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st S*) *S*} $\rightarrow_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st S*) *S*}⟩*nres-rel*›
proof −
have [*refine0*]: ‹*collect-valid-indices-wl S* ≤ ⇓ *Id* (*collect-valid-indices-wl T*)›
if ‹(*S*, *T*) ∈ *Id*› **for** *S T*
using *that* **by** *auto*
have [*iff*]: ‹(∀ (*x*::*bool*) *xa*. *P x xa*) ⟷ (∀ *xa*.(*P True xa* ∧ *P False xa*))› **for** *P*
by *metis*
have *in-Lit*: ‹*get-clauses-wl T′* ∝ (*xs* ! *j*) ! *0* ∈# $\mathcal{L}_{all}$ (*all-init-atms-st T′*)›
if
‹(*l*, *l′*) ∈ *nat-rel*› **and**

rel: ‹$(st, st') \in$ *nat-rel* $\times_r$ {(S, T). $(S, T) \in$ *Id* $\wedge$ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*} $\times_r$ *Id*› **and**
inv-x: ‹*mark-to-delete-clauses-wl2-D-inv S ys st*› **and**
‹*mark-to-delete-clauses-wl2-inv S' ys' st'*› **and**
dom: ‹$\neg$ *xs* ! *j* $\notin\#$ *dom-m* (*get-clauses-wl T'*)› **and**
‹$\neg$ *xs'* ! *i* $\notin\#$ *dom-m* (*get-clauses-wl T*)› **and**
assert: ‹*0* < *length* (*get-clauses-wl T* $\propto$ (*xs'* ! *i*))› **and**
st: ‹*st'* = (*i*, *sT*)› ‹*sT* = (*T*, *xs*)› ‹*st* = (*j*, *sT'*)› ‹*sT'* = (*T'*, *xs'*)› **and**
le: ‹*case st of* (*i*, *T*, *xs*) $\Rightarrow$ *i* < *length xs*› **and**
‹*case st' of* (*i*, *S*, *xs'*) $\Rightarrow$ *i* < *length xs'*› **and**
‹*0* < *length* (*get-clauses-wl T'* $\propto$ (*xs* ! *j*))›
for *S S' xs' l l' st st' i T j T' sT xs sT' ys ys'*
proof −

have *lits-T'*: ‹*literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st T'*) *T'*›
using *inv-x* **unfolding** *mark-to-delete-clauses-wl2-D-inv-def prod.simps st*
by *fast*
have ‹*literals-are-*$\mathcal{L}_{in}$ (*all-init-atms-st T*) *T*›
proof −
obtain *x xa xb* **where**
lits-T': ‹*literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st T'*) *T'*› **and**
Ta-x: ‹(*S*, *x*) $\in$ *state-wl-l None*› **and**
Ta-y: ‹(*T'*, *xa*) $\in$ *state-wl-l None*› **and**
‹*correct-watching'' S*› **and**
rem: ‹*remove-one-annot-true-clause*** *x xa*› **and**
list: ‹*twl-list-invs x*› **and**
x-xb: ‹(*x*, *xb*) $\in$ *twl-st-l None*› **and**
struct: ‹*twl-struct-invs xb*› **and**
confl: ‹*get-conflict-l x* = *None*› **and**
upd: ‹*clauses-to-update-l x* = {#}›
using *inv-x* **unfolding** *mark-to-delete-clauses-wl2-D-inv-def st prod.case mark-to-delete-clauses-wl2-inv-def mark-to-delete-clauses-l-inv-def*
by *blast*

obtain *y* **where**
Ta-y': ‹(*xa*, *y*) $\in$ *twl-st-l None*› **and**
struct': ‹*twl-struct-invs y*›
using *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*[*OF rem list confl upd x-xb struct*] **by** *blast*

have ‹*literals-are-*$\mathcal{L}_{in}$ (*all-init-atms-st T'*) *T'*›
by (*rule literals-are-*$\mathcal{L}_{in}$*'-literals-are-*$\mathcal{L}_{in}$*-iff*(*1*)[*THEN iffD1*, *OF Ta-y Ta-y' struct' lits-T'*])
then show *?thesis*
using *rel* **by** (*auto simp*: *st*)
qed
then have ‹*literals-are-in-*$\mathcal{L}_{in}$ (*all-init-atms-st T'*) (*mset* (*get-clauses-wl T'* $\propto$ (*xs* ! *j*)))›
using *literals-are-in-*$\mathcal{L}_{in}$*-nth*[*of* ‹*xs*!*i*› ‹*T*›] *rel dom*
by (*auto simp*: *st*)
then show *?thesis*
by (*rule literals-are-in-*$\mathcal{L}_{in}$*-in-*$\mathcal{L}_{all}$) (*use le assert rel dom* **in** ‹*auto simp*: *st*›)
qed
have *final-rel-del*:
‹((*j*, *mark-garbage-wl* (*xs* ! *j*) *T'*, *delete-index-and-swap xs j*),
i, *mark-garbage-wl* (*xs'* ! *i*) *T*, *delete-index-and-swap xs' i*)

$\in$ *nat-rel* $\times_r$ $\{(S, T). (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st* S) $S\}\times_r$ *Id*⟩
if
rel: ⟨$(st, st') \in$ *nat-rel* $\times_r$ $\{(S, T). (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st* T) $S\}$ $\times_r$
Id⟩ **and**
⟨*case st of* $(i, T, xs) \Rightarrow i <$ *length xs*⟩ **and**
⟨*case st′ of* $(i, S, xs') \Rightarrow i <$ *length xs′*⟩ **and**
inv: ⟨*mark-to-delete-clauses-wl2-D-inv S ys st*⟩ **and**
⟨*mark-to-delete-clauses-wl2-inv S′ ys′ st′*⟩ **and**
st: ⟨$st' = (i, sT)$⟩ ⟨$sT = (T, xs)$⟩ ⟨$st = (j, sT')$⟩ ⟨$sT' = (T', xs')$⟩ **and**
dom: ⟨$\neg$ *xs* ! *j* $\notin\#$ *dom-m* (*get-clauses-wl* T')⟩ **and**
⟨$\neg$ *xs′* ! *i* $\notin\#$ *dom-m* (*get-clauses-wl* T)⟩ **and**
le: ⟨$0 <$ *length* (*get-clauses-wl* $T \propto (xs' ! i)$)⟩ **and**
⟨$0 <$ *length* (*get-clauses-wl* $T' \propto (xs ! j)$)⟩ **and**
⟨*get-clauses-wl* $T' \propto (xs ! j) ! 0 \in\# \mathcal{L}_{all}$ (*all-init-atms-st* T')⟩ **and**
⟨(*can-del*, *can-del′*) $\in$ *bool-rel*⟩ **and**
can-del: ⟨*can-del*
$\in \{b. b \longrightarrow$
Propagated (*get-clauses-wl* $T' \propto (xs ! j) ! 0$) $(xs ! j)$
$\notin$ *set* (*get-trail-wl* T') $\wedge$
$\neg$ *irred* (*get-clauses-wl* T') $(xs ! j)\}$⟩ **and**
⟨*can-del′*
$\in \{b. b \longrightarrow$
Propagated (*get-clauses-wl* $T \propto (xs' ! i) ! 0$) $(xs' ! i)$
$\notin$ *set* (*get-trail-wl* T) $\wedge$
$\neg$ *irred* (*get-clauses-wl* T) $(xs' ! i)\}$⟩ **and**
i-le: ⟨$i <$ *length xs′*⟩ **and**
⟨$j <$ *length xs*⟩ **and**
[*simp*]: ⟨*can-del*⟩ **and**
[*simp*]: ⟨*can-del′*⟩
for *S S′ xs xs′ l l′ st st′ i T j T′ can-del can-del′ sT sT′ ys ys′*
proof –
have ⟨*literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st* (*mark-garbage-wl* $(xs' ! i)$ T)) (*mark-garbage-wl* $(xs' ! i)$ T)⟩
using *can-del inv rel* **unfolding** *mark-to-delete-clauses-wl2-D-inv-def st mark-garbage-wl-def*
by (*cases T*)
(*auto simp*: *literals-are-*$\mathcal{L}_{in}{}'$*-def init-clss-l-fmdrop-irrelev mset-take-mset-drop-mset′*
blits-in-$\mathcal{L}_{in}{}'$*-def all-init-lits-def*)
then show *?thesis*
using *inv rel* **unfolding** *st*
by *auto*
qed

show *?thesis*
unfolding *uncurry-def mark-to-delete-clauses-wl2-D-def mark-to-delete-clauses-wl2-def*
collect-valid-indices-wl-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
WHILEIT-refine[**where**
$R =$ ⟨*nat-rel* $\times_r$ $\{(S, T). (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st* S) $S\}$ $\times_r$ *Id*⟩])
subgoal
unfolding *mark-to-delete-clauses-wl-D-pre-def* **by** *auto*
subgoal by *auto*
subgoal for *x y xs xsa l la xa x′*
unfolding *mark-to-delete-clauses-wl2-D-inv-def* **by** (*cases x′*) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal for *S S′ xs xs′ l l′ st st′ i T j T′*
by (*rule in-Lit*; *assumption?*) *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal for *S S′ xs xs′ l l′ st st′ i T j T′ can-del can-del′*
by (*rule final-rel-del*; *assumption?*) *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *cdcl-GC-clauses-prog-copy-wl-entry*
:: ‹′*v clauses-l* ⇒ ′*v watched* ⇒ ′*v literal* ⇒
′*v clauses-l* ⇒ (′*v clauses-l* × ′*v clauses-l*) *nres*›
where
‹*cdcl-GC-clauses-prog-copy-wl-entry* = (λ*N W A N′*. *do* {
let le = *length W*;
(*i*, *N*, *N′*) ← $WHILE_T$
(λ(*i*, *N*, *N′*). *i* < *le*)
(λ(*i*, *N*, *N′*). *do* {
ASSERT(*i* < *length W*);
let C = *fst* (*W* ! *i*);
if C ∈# *dom-m N then do* {
D ← *SPEC*(λ*D*. *D* ∉# *dom-m N′* ∧ *D* ≠ *0*);
RETURN (*i*+*1*, *fmdrop C N*, *fmupd D* (*N* ∝ *C*, *irred N C*) *N′*)
} *else RETURN* (*i*+*1*, *N*, *N′*)
}) (*0*, *N*, *N′*);
RETURN (*N*, *N′*)
})›

definition *clauses-pointed-to* :: ‹′*v literal set* ⇒ (′*v literal* ⇒ ′*v watched*) ⇒ *nat set*›
where
‹*clauses-pointed-to* 𝒜 *W* ≡ ⋃(((‘) *fst*) ‘ *set* ‘ *W* ‘ 𝒜)›

lemma *clauses-pointed-to-insert*[*simp*]:
‹*clauses-pointed-to* (*insert A* 𝒜) *W* =
fst ‘ *set* (*W A*) ∪
clauses-pointed-to 𝒜 *W*› **and**
clauses-pointed-to-empty[*simp*]:
‹*clauses-pointed-to* {} *W* = {}›
by (*auto simp*: *clauses-pointed-to-def*)

lemma *cdcl-GC-clauses-prog-copy-wl-entry*:
fixes *A* :: ‹′*v literal*› **and** *WS* :: ‹′*v literal* ⇒ ′*v watched*›
defines [*simp*]: ‹*W* ≡ *WS A*›
assumes ‹
ran m0 ⊆ *set-mset* (*dom-m N0′*) ∧
(∀ *L*∈*dom m0*. *L* ∉# (*dom-m N0*)) ∧
set-mset (*dom-m N0*) ⊆ *clauses-pointed-to* (*set-mset* 𝒜) *WS* ∧
0 ∉# *dom-m N0′*›
shows
‹*cdcl-GC-clauses-prog-copy-wl-entry N0 W A N0′* ≤

SPEC(λ(*N*, *N'*). (∃ *m*. *GC-remap*** (*N0*, *m0*, *N0'*) (*N*, *m*, *N'*) ∧
ran m ⊆ *set-mset* (*dom-m N'*) ∧
(∀ *L*∈*dom m*. *L* ∉# (*dom-m N*)) ∧
set-mset (*dom-m N*) ⊆ *clauses-pointed-to* (*set-mset* (*remove1-mset A 𝒜*)) *WS*) ∧
(∀ *L* ∈ *set W*. *fst L* ∉# *dom-m N*) ∧
0 ∉# *dom-m N'*)›

proof −
have [*simp*]:
‹*x* ∈# *remove1-mset a* (*dom-m aaa*) ⟷ *x* ≠ *a* ∧ *x* ∈# *dom-m aaa*› **for** *x a aaa*
using *distinct-mset-dom*[*of aaa*]
by (*cases* ‹*a* ∈# *dom-m aaa*›)
(*auto dest*!: *multi-member-split simp*: *add-mset-eq-add-mset*)

show *?thesis*
unfolding *cdcl-GC-clauses-prog-copy-wl-entry-def*
apply (*refine-vcg*
WHILET-rule[**where** *I* = ‹λ(*i*, *N*, *N'*). ∃ *m*. *GC-remap*** (*N0*, *m0*, *N0'*) (*N*, *m*, *N'*) ∧
ran m ⊆ *set-mset* (*dom-m N'*) ∧
(∀ *L*∈*dom m*. *L* ∉# (*dom-m N*)) ∧
set-mset (*dom-m N*) ⊆ *clauses-pointed-to* (*set-mset* (*remove1-mset A 𝒜*)) *WS* ∪
(*fst*) ‘ *set* (*drop i W*) ∧
(∀ *L* ∈ *set* (*take i W*). *fst L* ∉# *dom-m N*) ∧
0 ∉# *dom-m N'*› **and**
R = ‹*measure* (λ(*i*, *N*, *N'*). *length W* −*i*)›])
subgoal by *auto*
subgoal
using *assms*
by (*cases* ‹*A* ∈# *𝒜*›) (*auto dest*!: *multi-member-split*)
subgoal by *auto*
subgoal for *s aa ba aaa baa x x1 x2 x1a x2a*
apply *clarify*
apply (*subgoal-tac* ‹(∃ *m'*. *GC-remap* (*aaa*, *m*, *baa*) (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*, *m'*,
fmupd x (*the* (*fmlookup aaa* (*fst* (*W* ! *aa*)))) *baa*) ∧
ran m' ⊆ *set-mset* (*dom-m* (*fmupd x* (*the* (*fmlookup aaa* (*fst* (*W* ! *aa*)))) *baa*)) ∧
(∀ *L*∈*dom m'*. *L* ∉# (*dom-m* (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*)))) ∧
set-mset (*dom-m* (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*)) ⊆
clauses-pointed-to (*set-mset* (*remove1-mset A 𝒜*)) *WS* ∪
fst ‘ *set* (*drop* (*Suc aa*) *W*) ∧
(∀ *L* ∈ *set* (*take* (*Suc aa*) *W*). *fst L* ∉# *dom-m* (*fmdrop* (*fst* (*W* ! *aa*)) *aaa*))›)
apply (*auto intro*: *rtranclp.rtrancl-into-rtrancl*)[]
apply (*auto simp*: *GC-remap.simps Cons-nth-drop-Suc*[*symmetric*]
take-Suc-conv-app-nth
dest: *multi-member-split*)
apply (*rule-tac x*= ‹*m*(*fst* (*W* ! *aa*) ↦ *x*)› **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*= ‹*fst* (*W* ! *aa*)› **in** *exI*)
apply (*force dest*: *rtranclp-GC-remap-ran-m-no-lost*)
apply *auto*
by (*smt basic-trans-rules*(*31*) *fun-upd-apply mem-Collect-eq option.simps*(*1*) *ran-def*)
subgoal by *auto*
subgoal by (*auto 5 5 simp*: *GC-remap.simps Cons-nth-drop-Suc*[*symmetric*]
take-Suc-conv-app-nth
dest: *multi-member-split*)
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
done
qed

definition *cdcl-GC-clauses-prog-single-wl*
:: ‹$'v$ *clauses-l* ⇒ ($'v$ *literal* ⇒ $'v$ *watched*) ⇒ $'v$ ⇒
$'v$ *clauses-l* ⇒ ($'v$ *clauses-l* × $'v$ *clauses-l* × ($'v$ *literal* ⇒ $'v$ *watched*)) *nres*›
where
‹*cdcl-GC-clauses-prog-single-wl* = (λ*N WS A N'*. *do* {
L ← *RES* {*Pos A*, *Neg A*};
(*N*, *N'*) ← *cdcl-GC-clauses-prog-copy-wl-entry N* (*WS L*) *L N'*;
let WS = *WS*(*L* := []);
(*N*, *N'*) ← *cdcl-GC-clauses-prog-copy-wl-entry N* (*WS* (−*L*)) (−*L*) *N'*;
let WS = *WS*(−*L* := []);
RETURN (*N*, *N'*, *WS*)
})›

lemma *clauses-pointed-to-remove1-if*:
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹ *xa* ∈# *dom-m aa* ⟹
xa ∈ *clauses-pointed-to* (*set-mset* (*remove1-mset L* 𝒜))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
xa ∈ *clauses-pointed-to* (*set-mset* (*remove1-mset L* 𝒜)) *W*›
by (*cases* ‹*L* ∈# 𝒜›)
(*fastforce simp*: *clauses-pointed-to-def*
dest!: *multi-member-split*)+

lemma *clauses-pointed-to-remove1-if2*:
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹ *xa* ∈# *dom-m aa* ⟹
xa ∈ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L*, *L'*#}))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
xa ∈ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L*, *L'*#})) *W*›
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹ *xa* ∈# *dom-m aa* ⟹
xa ∈ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L'*, *L*#}))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
xa ∈ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L'*, *L*#})) *W*›
by (*cases* ‹*L* ∈# 𝒜›; *fastforce simp*: *clauses-pointed-to-def*
dest!: *multi-member-split*)+

lemma *clauses-pointed-to-remove1-if2-eq*:
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹
set-mset (*dom-m aa*) ⊆ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L*, *L'*#}))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
set-mset (*dom-m aa*) ⊆ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L*, *L'*#})) *W*›
‹∀ *L*∈*set* (*W L*). *fst L* ∉# *dom-m aa* ⟹
set-mset (*dom-m aa*) ⊆ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L'*, *L*#}))
(λ*a*. *if a* = *L then* [] *else W a*) ⟷
set-mset (*dom-m aa*) ⊆ *clauses-pointed-to* (*set-mset* (𝒜 − {#*L'*, *L*#})) *W*›
by (*auto simp*: *clauses-pointed-to-remove1-if2*)

lemma *negs-remove-Neg*: ‹*A* ∉# 𝒜 ⟹ *negs* 𝒜 + *poss* 𝒜 − {#*Neg A*, *Pos A*#} =
negs 𝒜 + *poss* 𝒜›
by (*induction* 𝒜) *auto*
lemma *poss-remove-Pos*: ‹*A* ∉# 𝒜 ⟹ *negs* 𝒜 + *poss* 𝒜 − {#*Pos A*, *Neg A*#} =
negs 𝒜 + *poss* 𝒜›
by (*induction* 𝒜) *auto*

lemma *cdcl-GC-clauses-prog-single-wl-removed*:
‹$\forall L{\in}set\ (W\ (Pos\ A)).\ fst\ L \notin\#\ dom\text{-}m\ aaa \Longrightarrow$
$\forall L{\in}set\ (W\ (Neg\ A)).\ fst\ L \notin\#\ dom\text{-}m\ a \Longrightarrow$
$GC\text{-}remap^{**}\ (aaa,\ ma,\ baa)\ (a,\ mb,\ b) \Longrightarrow$
$set\text{-}mset\ (dom\text{-}m\ a) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (negs\ \mathcal{A} + poss\ \mathcal{A} - \{\#Neg\ A,\ Pos\ A\#\}))\ W$
$\Longrightarrow$
$xa \in\#\ dom\text{-}m\ a \Longrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (Neg\ `\ set\text{-}mset\ (remove1\text{-}mset\ A\ \mathcal{A}) \cup Pos\ `\ set\text{-}mset\ (remove1\text{-}mset\ A$
$\mathcal{A}))$
$(W(Pos\ A := [],\ Neg\ A := []))$›
‹$\forall L{\in}set\ (W\ (Neg\ A)).\ fst\ L \notin\#\ dom\text{-}m\ aaa \Longrightarrow$
$\forall L{\in}set\ (W\ (Pos\ A)).\ fst\ L \notin\#\ dom\text{-}m\ a \Longrightarrow$
$GC\text{-}remap^{**}\ (aaa,\ ma,\ baa)\ (a,\ mb,\ b) \Longrightarrow$
$set\text{-}mset\ (dom\text{-}m\ a) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (negs\ \mathcal{A} + poss\ \mathcal{A} - \{\#Pos\ A,\ Neg\ A\#\}))\ W$
$\Longrightarrow$
$xa \in\#\ dom\text{-}m\ a \Longrightarrow$
$xa \in clauses\text{-}pointed\text{-}to$
$(Neg\ `\ set\text{-}mset\ (remove1\text{-}mset\ A\ \mathcal{A}) \cup Pos\ `\ set\text{-}mset\ (remove1\text{-}mset\ A\ \mathcal{A}))$
$(W(Neg\ A := [],\ Pos\ A := []))$›
supply *poss-remove-Pos*[*simp*] *negs-remove-Neg*[*simp*]
by (*case-tac* [!] ‹$A \in\#\ \mathcal{A}$›)
(*fastforce simp*: *clauses-pointed-to-def*
dest!: *multi-member-split*
dest: *rtranclp-GC-remap-ran-m-no-lost*)+

lemma *cdcl-GC-clauses-prog-single-wl*:
fixes A :: ‹$'v$› **and** WS :: ‹$'v\ literal \Rightarrow 'v\ watched$› **and**
$N0$ :: ‹$'v\ clauses\text{-}l$›
assumes ‹$ran\ m \subseteq set\text{-}mset\ (dom\text{-}m\ N0') \wedge$
$(\forall L{\in}dom\ m.\ L \notin\#\ (dom\text{-}m\ N0)) \wedge$
$set\text{-}mset\ (dom\text{-}m\ N0) \subseteq$
$clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (negs\ \mathcal{A} + poss\ \mathcal{A}))\ W \wedge$
$0 \notin\#\ dom\text{-}m\ N0'$›
shows
‹$cdcl\text{-}GC\text{-}clauses\text{-}prog\text{-}single\text{-}wl\ N0\ W\ A\ N0' \le$
$SPEC(\lambda(N,\ N',\ WS').\ \exists m'.\ GC\text{-}remap^{**}\ (N0,\ m,\ N0')\ (N,\ m',\ N') \wedge$
$ran\ m' \subseteq set\text{-}mset\ (dom\text{-}m\ N') \wedge$
$(\forall L{\in}dom\ m'.\ L \notin\#\ dom\text{-}m\ N) \wedge$
$WS'\ (Pos\ A) = [] \wedge WS'\ (Neg\ A) = [] \wedge$
$(\forall L.\ L \neq Pos\ A \longrightarrow L \neq Neg\ A \longrightarrow W\ L = WS'\ L) \wedge$
$set\text{-}mset\ (dom\text{-}m\ N) \subseteq$
$clauses\text{-}pointed\text{-}to$
$(set\text{-}mset\ (negs\ (remove1\text{-}mset\ A\ \mathcal{A}) + poss\ (remove1\text{-}mset\ A\ \mathcal{A})))\ WS' \wedge$
$0 \notin\#\ dom\text{-}m\ N'$
)›
proof –
have [*simp*]: ‹$A \notin\#\ \mathcal{A} \Longrightarrow negs\ \mathcal{A} + poss\ \mathcal{A} - \{\#Neg\ A,\ Pos\ A\#\} =$
$negs\ \mathcal{A} + poss\ \mathcal{A}$›
by (*induction* $\mathcal{A}$) *auto*
have [*simp*]: ‹$A \notin\#\ \mathcal{A} \Longrightarrow negs\ \mathcal{A} + poss\ \mathcal{A} - \{\#Pos\ A,\ Neg\ A\#\} =$
$negs\ \mathcal{A} + poss\ \mathcal{A}$›
by (*induction* $\mathcal{A}$) *auto*
show *?thesis*
unfolding *cdcl-GC-clauses-prog-single-wl-def*
apply (*refine-vcg*)

subgoal for *x*
  apply (*rule order-trans*, *rule cdcl-GC-clauses-prog-copy-wl-entry*[*of* - - -
    ⟨*negs* $\mathcal{A}$ + *poss* $\mathcal{A}$⟩])
   apply(*solves* ⟨*use assms in auto*⟩)
  apply (*rule RES-rule*)
  apply (*refine-vcg*)
  apply *clarify*
  subgoal for *aa ba aaa baa ma*
    apply (*rule order-trans*,
      rule cdcl-GC-clauses-prog-copy-wl-entry[*of ma* - -
        ⟨*remove1-mset x* (*negs* $\mathcal{A}$ + *poss* $\mathcal{A}$)⟩])
     apply (*solves* ⟨*auto simp*: *clauses-pointed-to-remove1-if*⟩)[]
    unfolding *Let-def*
    apply (*rule RES-rule*)
    apply *clarsimp*
    apply (*simp add*: *eq-commute*[*of* ⟨*Neg* -⟩]
      uminus-lit-swap clauses-pointed-to-remove1-if)
    apply *auto*
     apply (*rule-tac x=mb* **in** *exI*)
     apply (*auto dest*!:
      simp: *clauses-pointed-to-remove1-if*
      clauses-pointed-to-remove1-if2
      clauses-pointed-to-remove1-if2-eq)
     apply (*subst* (*asm*) *clauses-pointed-to-remove1-if2-eq*)
      apply (*force dest*: *rtranclp-GC-remap-ran-m-no-lost*)
     apply (*auto intro*!: *cdcl-GC-clauses-prog-single-wl-removed*)[]
     apply (*rule-tac x=mb* **in** *exI*)
     apply (*auto dest*: *multi-member-split*[*of A*]
      simp: *clauses-pointed-to-remove1-if*
      clauses-pointed-to-remove1-if2
      clauses-pointed-to-remove1-if2-eq)
     apply (*subst* (*asm*) *clauses-pointed-to-remove1-if2-eq*)
      apply (*force dest*: *rtranclp-GC-remap-ran-m-no-lost*)
    apply (*auto intro*!: *cdcl-GC-clauses-prog-single-wl-removed*)[]
    done
  done
  done
qed

definition *cdcl-GC-clauses-prog-wl-inv*
  :: ⟨$'v$ *multiset* ⇒ $'v$ *clauses-l* ⇒
    $'v$ *multiset* × ($'v$ *clauses-l* × $'v$ *clauses-l* × ($'v$ *literal* ⇒ $'v$ *watched*)) ⇒ *bool*⟩
where
⟨*cdcl-GC-clauses-prog-wl-inv* $\mathcal{A}$ *N0* = (λ($\mathcal{B}$, (*N*, *N'*, *WS*)). $\mathcal{B}$ ⊆# $\mathcal{A}$ ∧
  (∀ *A* ∈ *set-mset* $\mathcal{A}$ − *set-mset* $\mathcal{B}$. (*WS* (*Pos A*) = []) ∧ *WS* (*Neg A*) = []) ∧
  0 ∉# *dom-m N'* ∧
  (∃ *m*. *GC-remap*** (*N0*, (λ-. *None*), *fmempty*) (*N*, *m*, *N'*)∧
    ran m ⊆ *set-mset* (*dom-m N'*) ∧
    (∀ *L*∈*dom m*. *L* ∉# *dom-m N*) ∧
    set-mset (*dom-m N*) ⊆ *clauses-pointed-to* (*Neg* ‘ *set-mset* $\mathcal{B}$ ∪ *Pos* ‘ *set-mset* $\mathcal{B}$) *WS*))⟩

definition *cdcl-GC-clauses-prog-wl* :: ⟨$'v$ *twl-st-wl* ⇒ $'v$ *twl-st-wl nres*⟩ **where**
  ⟨*cdcl-GC-clauses-prog-wl* = (λ(*M*, *N0*, *D*, *NE*, *UE*, *Q*, *WS*). *do* {
    ASSERT(*cdcl-GC-clauses-pre-wl* (*M*, *N0*, *D*, *NE*, *UE*, *Q*, *WS*));
    $\mathcal{A}$ ← *SPEC*(λ$\mathcal{A}$. *set-mset* $\mathcal{A}$ = *set-mset* (*all-init-atms N0 NE*));

```
    (-, (N, N', WS)) ← WHILE_T^{cdcl-GC-clauses-prog-wl-inv 𝒜 N0}
      (λ(ℬ, -). ℬ ≠ {#})
      (λ(ℬ, (N, N', WS)). do {
        ASSERT(ℬ ≠ {#});
        A ← SPEC (λA. A ∈# ℬ);
        (N, N', WS) ← cdcl-GC-clauses-prog-single-wl N WS A N';
        RETURN (remove1-mset A ℬ, (N, N', WS))
      })
      (𝒜, (N0, fmempty, WS));
    RETURN (M, N', D, NE, UE, Q, WS)
  })›
```

```
lemma cdcl-GC-clauses-prog-wl:
  assumes ‹((M, N0, D, NE, UE, Q, WS), S) ∈ state-wl-l None ∧
    correct-watching'' (M, N0, D, NE, UE, Q, WS) ∧ cdcl-GC-clauses-pre S ∧
   set-mset (dom-m N0) ⊆ clauses-pointed-to
      (Neg ‘ set-mset (all-init-atms N0 NE) ∪ Pos ‘ set-mset (all-init-atms N0 NE)) WS›
  shows
   ‹cdcl-GC-clauses-prog-wl (M, N0, D, NE, UE, Q, WS) ≤
      (SPEC(λ(M', N', D', NE', UE', Q', WS'). (M', D', NE', UE', Q') = (M, D, NE, UE, Q) ∧
         (∃ m. GC-remap** (N0, (λ-. None), fmempty) (fmempty, m, N')) ∧
         0 ∉# dom-m N' ∧ (∀ L ∈# all-init-lits N0 NE. WS' L = [])))›
proof −
  show ?thesis
    supply[[goals-limit=1]]
    unfolding cdcl-GC-clauses-prog-wl-def
    apply (refine-vcg
      WHILEIT-rule[where R = ‹measure (λ(𝒜::'v multiset, (-::'v clauses-l, - ::'v clauses-l,
          -:: 'v literal ⇒ 'v watched)). size 𝒜)›])
    subgoal
      using assms
      unfolding cdcl-GC-clauses-pre-wl-def
      by blast
    subgoal by auto
    subgoal using assms unfolding cdcl-GC-clauses-prog-wl-inv-def by auto
    subgoal by auto
    subgoal for a b aa ba ab bb ac bc ad bd ae be x s af bf ag bg ah bh xa
      unfolding cdcl-GC-clauses-prog-wl-inv-def
      apply clarify
      apply (rule order-trans,
         rule-tac m=m and 𝒜=af in cdcl-GC-clauses-prog-single-wl)
      subgoal by auto
      subgoal
        apply (rule RES-rule)
        apply clarify
        apply (rule RETURN-rule)
        apply clarify
        apply (intro conjI)
             apply (solves auto)
            apply (solves ‹auto dest!: multi-member-split›)
           apply (solves auto)
          apply (rule-tac x=m' in exI)
          apply (solves auto)[]
         apply (simp-all add: size-Diff1-less)[]
        done
```

done
subgoal
unfolding *cdcl-GC-clauses-prog-wl-inv-def*
by *auto*
subgoal
unfolding *cdcl-GC-clauses-prog-wl-inv-def*
by *auto*
subgoal
unfolding *cdcl-GC-clauses-prog-wl-inv-def*
by *auto*
subgoal
unfolding *cdcl-GC-clauses-prog-wl-inv-def*
by (*intro ballI*, *rename-tac L*, *case-tac L*)
(*auto simp*: *in-all-lits-of-mm-ain-atms-of-iff all-init-atms-def*
simp del: *all-init-atms-def*[*symmetric*]
dest!: *multi-member-split*)
done
qed

lemma *all-init-atms-fmdrop-add-mset-unit*:
‹$C \in\# \textit{dom-m baa} \Longrightarrow \textit{irred baa } C \Longrightarrow$
all-init-atms (*fmdrop C baa*) (*add-mset* (*mset* (*baa* $\propto$ *C*)) *da*) =
all-init-atms baa da›
‹$C \in\# \textit{dom-m baa} \Longrightarrow \neg\textit{irred baa } C \Longrightarrow$
all-init-atms (*fmdrop C baa*) *da* =
all-init-atms baa da›
by (*auto simp del*: *all-init-atms-def*[*symmetric*]
simp: *all-init-atms-def all-init-lits-def*
init-clss-l-fmdrop-irrelev image-mset-remove1-mset-if)

lemma *cdcl-GC-clauses-prog-wl2*:
assumes ‹((*M*, *N0*, *D*, *NE*, *UE*, *Q*, *WS*), *S*) $\in$ *state-wl-l None* $\wedge$
correct-watching'' (*M*, *N0*, *D*, *NE*, *UE*, *Q*, *WS*) $\wedge$ *cdcl-GC-clauses-pre S* $\wedge$
set-mset (*dom-m N0*) $\subseteq$ *clauses-pointed-to*
(*Neg* ‘ *set-mset* (*all-init-atms N0 NE*) $\cup$ *Pos* ‘ *set-mset* (*all-init-atms N0 NE*)) *WS*› **and**
‹*N0* = *N0'*›
shows
‹*cdcl-GC-clauses-prog-wl* (*M*, *N0*, *D*, *NE*, *UE*, *Q*, *WS*) $\leq$
$\Downarrow$ {((*M'*, *N''*, *D'*, *NE'*, *UE'*, *Q'*, *WS'*), (*N*, *N'*)). (*M'*, *D'*, *NE'*, *UE'*, *Q'*) = (*M*, *D*, *NE*, *UE*, *Q*) $\wedge$
N'' = *N* $\wedge$ ($\forall$ *L*$\in\#$*all-init-lits N0 NE*. *WS' L* = [])$\wedge$
all-init-lits N0 NE = *all-init-lits N NE'* $\wedge$
($\exists$ *m*. *GC-remap*** (*N0*, (λ-. *None*), *fmempty*) (*fmempty*, *m*, *N*))}
(*SPEC*(λ(*N'*::(*nat*, *'a literal list* $\times$ *bool*) *fmap*, *m*).
GC-remap** (*N0'*, (λ-. *None*), *fmempty*) (*fmempty*, *m*, *N'*) $\wedge$
0 $\notin\#$ *dom-m N'*))›
proof −
show *?thesis*
unfolding ‹*N0* = *N0'*›[*symmetric*]
apply (*rule order-trans*[*OF cdcl-GC-clauses-prog-wl*[*OF assms*(*1*)]])
apply (*rule RES-refine*)
apply (*fastforce dest*: *rtranclp-GC-remap-all-init-lits*)

done
qed

definition *cdcl-twl-stgy-restart-abs-wl-D-inv* **where**
⟨*cdcl-twl-stgy-restart-abs-wl-D-inv S0 brk T n* ⟷
cdcl-twl-stgy-restart-abs-wl-inv S0 brk T n ∧
literals-are-$\mathcal{L}_{in}$ (*all-atms-st T*) *T*⟩

definition *cdcl-GC-clauses-pre-wl-D* :: ⟨*nat twl-st-wl* ⇒ *bool*⟩ **where**
⟨*cdcl-GC-clauses-pre-wl-D S* ⟷ (
∃ *T*. (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st S*) *S* ∧
cdcl-GC-clauses-pre-wl T
)⟩

definition *cdcl-twl-full-restart-wl-D-GC-prog-post* :: ⟨*'v twl-st-wl* ⇒ *'v twl-st-wl* ⇒ *bool*⟩ **where**
⟨*cdcl-twl-full-restart-wl-D-GC-prog-post S T* ⟷
(∃ *S' T'*. (*S*, *S'*) ∈ *Id* ∧ (*T*, *T'*) ∈ *Id* ∧
cdcl-twl-full-restart-wl-GC-prog-post S' T')⟩

definition *cdcl-GC-clauses-wl-D* :: ⟨*nat twl-st-wl* ⇒ *nat twl-st-wl nres*⟩ **where**
⟨*cdcl-GC-clauses-wl-D* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*). *do* {
ASSERT(*cdcl-GC-clauses-pre-wl-D* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*));
let b = *True*;
if b then do {
(*N'*, -) ← *SPEC* (λ(*N''*, *m*). *GC-remap*** (*N*, *Map.empty*, *fmempty*) (*fmempty*, *m*, *N''*) ∧
0 ∉# *dom-m N''*);
Q ← *SPEC*(λ*Q*. *correct-watching'* (*M*, *N'*, *D*, *NE*, *UE*, *WS*, *Q*) ∧
blits-in-$\mathcal{L}_{in}'$ (*M*, *N'*, *D*, *NE*, *UE*, *WS*, *Q*));
RETURN (*M*, *N'*, *D*, *NE*, *UE*, *WS*, *Q*)
}
else RETURN (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*)})⟩

lemma *cdcl-GC-clauses-wl-D-cdcl-GC-clauses-wl*:
⟨(*cdcl-GC-clauses-wl-D*, *cdcl-GC-clauses-wl*) ∈ {(*S*::*nat twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st S*) *S*} $\rightarrow_f$ ⟨{(*S*::*nat twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}'$ (*all-init-atms-st S*) *S*}⟩*nres-rel*⟩
unfolding *cdcl-GC-clauses-wl-D-def cdcl-GC-clauses-wl-def*
apply (*intro frefI nres-relI*)
apply *refine-vcg*
subgoal unfolding *cdcl-GC-clauses-pre-wl-D-def* **by** *blast*
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by *auto*
subgoal by (*auto simp*: *state-wl-l-def*)
subgoal by (*auto simp*: *state-wl-l-def literals-are-*$\mathcal{L}_{in}'$*-def is-*$\mathcal{L}_{all}$*-def*
all-init-atms-def all-init-lits-def
dest: *rtranclp-GC-remap-init-clss-l-old-new*)
subgoal by (*auto simp*: *state-wl-l-def*)
done

definition *cdcl-twl-full-restart-wl-D-GC-prog* **where**
⟨*cdcl-twl-full-restart-wl-D-GC-prog S* = *do* {
ASSERT(*cdcl-twl-full-restart-wl-GC-prog-pre S*);
S' ← *cdcl-twl-local-restart-wl-spec0 S*;
T ← *remove-one-annot-true-clause-imp-wl-D S'*;

ASSERT(*mark-to-delete-clauses-wl-D-pre T*);
U ← *mark-to-delete-clauses-wl2-D T*;
V ← *cdcl-GC-clauses-wl-D U*;
ASSERT(*cdcl-twl-full-restart-wl-D-GC-prog-post S V*);
RETURN V
}›

lemma $\mathcal{L}_{all}$-*all-init-atms-all-init-lits*:
‹*set-mset* ($\mathcal{L}_{all}$ (*all-init-atms N NE*)) = *set-mset* (*all-init-lits N NE*)›
using *is-*$\mathcal{L}_{all}$*-def* **by** *blast*

lemma $\mathcal{L}_{all}$-*all-atms-all-lits*:
‹*set-mset* ($\mathcal{L}_{all}$ (*all-atms N NE*)) = *set-mset* (*all-lits N NE*)›
by (*simp add*: $\mathcal{L}_{all}$*-atm-of-all-lits-of-mm all-atms-def all-lits-def*)

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

lemma *cdcl-twl-full-restart-wl-D-GC-prog*:
‹(*cdcl-twl-full-restart-wl-D-GC-prog*, *cdcl-twl-full-restart-wl-GC-prog*) ∈
{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$′ (*all-init-atms-st S*) *S*} $\rightarrow_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-init-atms-st S*) *S*}⟩*nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ -›)
proof −
have [*refine0*]: ‹*cdcl-twl-local-restart-wl-spec0 x*
≤ ⇓ *?R* (*cdcl-twl-local-restart-wl-spec0 y*)›
if ‹(*x*, *y*) ∈ *?R*›
for *x y*
using *that* **apply** (*case-tac x*; *case-tac y*)
by (*auto 5 1 simp*: *cdcl-twl-local-restart-wl-spec0-def image-iff*
RES-RES-RETURN-RES2 intro!: *RES-refine*)
(*auto simp*: *literals-are-*$\mathcal{L}_{in}$′*-def blits-in-*$\mathcal{L}_{in}$′*-def*)
show *?thesis*
unfolding *cdcl-twl-full-restart-wl-D-GC-prog-def cdcl-twl-full-restart-wl-GC-prog-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
remove-one-annot-true-clause-imp-wl-D-remove-one-annot-true-clause-imp-wl[*THEN fref-to-Down*]
mark-to-delete-clauses-wl-D-mark-to-delete-clauses-wl2[*THEN fref-to-Down*]
cdcl-GC-clauses-wl-D-cdcl-GC-clauses-wl[*THEN fref-to-Down*])
subgoal by *fast*
subgoal unfolding *mark-to-delete-clauses-wl-D-pre-def* **by** *fast*
subgoal unfolding *cdcl-twl-full-restart-wl-D-GC-prog-post-def* **by** *fast*
subgoal unfolding *cdcl-twl-full-restart-wl-GC-prog-post-def*
literals-are-$\mathcal{L}_{in}$*-def literals-are-*$\mathcal{L}_{in}$′*-def*
is-$\mathcal{L}_{all}$*-def blits-in-*$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$′*-def*
$\mathcal{L}_{all}$*-all-init-atms-all-init-lits*
all-atms-def[*symmetric*]
all-init-atms-def[*symmetric*]
all-lits-alt-def[*symmetric*]
all-init-lits-def[*symmetric*]
$\mathcal{L}_{all}$*-all-atms-all-lits*
by *fastforce*
done
qed

definition *restart-prog-wl-D* :: *nat twl-st-wl* ⇒ *nat* ⇒ *bool* ⇒ (*nat twl-st-wl* × *nat*) *nres* **where**
‹*restart-prog-wl-D S n brk* = *do* {
ASSERT(*restart-abs-wl-D-pre S brk*);
b ← *restart-required-wl S n*;
b2 ← *SPEC*(λ-. *True*);
if b2 ∧ *b* ∧ ¬*brk then do* {
T ← *cdcl-twl-full-restart-wl-D-GC-prog S*;
RETURN (*T*, *n* + *1*)
}
else if b ∧ ¬*brk then do* {
T ← *cdcl-twl-restart-wl-D-prog S*;
RETURN (*T*, *n* + *1*)
}
else
RETURN (*S*, *n*)
}›

lemma *restart-abs-wl-D-pre-literals-are-*$\mathcal{L}_{in}$*'*:
assumes
‹(*x*, *y*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} $\times_f$
nat-rel $\times_f$
bool-rel› **and**
‹*x1* = (*x1a*, *x2*)› **and**
‹*y* = (*x1*, *x2a*)› **and**
‹*x1b* = (*x1c*, *x2b*)› **and**
‹*x* = (*x1b*, *x2c*)› **and**
pre: ‹*restart-abs-wl-D-pre x1c x2c*› **and**
‹*b2* ∧ *b* ∧ ¬ *x2c*› **and**
‹*b2a* ∧ *ba* ∧ ¬ *x2a*›
shows ‹(*x1c*, *x1a*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*}›
proof −
have *y*: ‹*y* = ((*x1a*, *x2*), *x2a*)› **and**
x-y: ‹*x* = *y*› **and**
[*simp*]: ‹*x1c* = *x1a*›
using *assms* **by** *auto*
obtain *x xa* **where**
lits: ‹*literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st x1c*) *x1c*› **and**
x1c-x: ‹(*x1c*, *x*) ∈ *state-wl-l None*› **and**
‹*correct-watching x1c*› **and**
x-xa: ‹(*x*, *xa*) ∈ *twl-st-l None*› **and**
‹*restart-prog-pre xa x2c*› **and**
list-invs: ‹*twl-list-invs x*› **and**
struct-invs: ‹*twl-struct-invs xa*› **and**
‹*clauses-to-update-l x* = {#}›
using *pre* **unfolding** *restart-abs-wl-D-pre-def restart-abs-wl-pre-def*
restart-abs-l-pre-def restart-prog-pre-def **by** *blast*
have ‹*set-mset* (*all-init-atms-st x1a*) = *set-mset* (*all-atms-st x1a*)›
using *literals-are-*$\mathcal{L}_{in}$*'-literals-are-*$\mathcal{L}_{in}$*-iff*(*3*)[*OF x1c-x x-xa struct-invs*]
lits
by *auto*
with $\mathcal{L}_{all}$*-cong*[*OF this*] **have** ‹*literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st x1a*) *x1a*›
using *assms*(*1*)
unfolding *literals-are-*$\mathcal{L}_{in}$*'-def literals-are-*$\mathcal{L}_{in}$*-def*

all-init-lits-def[*symmetric*] *y x-y*
blits-in-$\mathcal{L}_{in}$*-def blits-in-*$\mathcal{L}_{in}$*'-def*
by *auto*
then show *?thesis*
using *x-y* **by** *auto*
qed

lemma *restart-prog-wl-D-restart-prog-wl*:
‹(*uncurry2 restart-prog-wl-D*, *uncurry2 restart-prog-wl*) ∈
{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} $\times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} $\times_r$ *nat-rel*⟩*nres-rel*›
proof −
have [*refine0*]: ‹*restart-required-wl x1c x2b* ≤ ⇓ *Id* (*restart-required-wl x1a x2*)›
if ‹(*x1c*, *x1a*) ∈ *Id*› ‹(*x2b*, *x2*) ∈ *Id*›
for *x1c x1a x2b x2*
using *that* **by** *auto*
have *restart-abs-wl-D-pre*: ‹*restart-abs-wl-D-pre x1c x2c*›
if
‹(*x*, *y*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} $\times_f$ *nat-rel* $\times_f$ *bool-rel*› **and**
‹*x1* = (*x1a*, *x2*)› **and**
‹*y* = (*x1*, *x2a*)› **and**
‹*x1b* = (*x1c*, *x2b*)› **and**
‹*x* = (*x1b*, *x2c*)› **and**
pre: ‹*restart-abs-wl-pre x1a x2a*›
for *x y x1 x1a x2 x2a x1b x1c x2b x2c*
proof −
have ‹*restart-abs-wl-pre x1a x2c*› **and** *lits-T*: ‹*literals-are-*$\mathcal{L}_{in}$ (*all-atms-st x1a*) *x1a*›
using *pre that*
unfolding *restart-abs-wl-D-pre-def*
by *auto*
then obtain *xa x* **where**
S-x: ‹(*x1a*, *x*) ∈ *state-wl-l None*› **and**
‹*correct-watching x1a*› **and**
x-xa: ‹(*x*, *xa*) ∈ *twl-st-l None*› **and**
struct: ‹*twl-struct-invs xa*› **and**
list: ‹*twl-list-invs x*› **and**
‹*clauses-to-update-l x* = {#}› **and**
‹*twl-stgy-invs xa*› **and**
‹¬ *x2c* ⟶ *get-conflict xa* = *None*›
unfolding *restart-abs-wl-pre-def restart-abs-l-pre-def restart-prog-pre-def* **by** *blast*

show *?thesis*
using *pre that literals-are-*$\mathcal{L}_{in}$*'-literals-are-*$\mathcal{L}_{in}$*-iff*(*1*,*2*)[*THEN iffD2*,
OF S-x x-xa struct lits-T]
unfolding *restart-abs-wl-D-pre-def*
by *auto*
qed

show *?thesis*
unfolding *uncurry-def restart-prog-wl-D-def restart-prog-wl-def*
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
cdcl-twl-restart-wl-D-prog-cdcl-twl-restart-wl-prog[*THEN fref-to-Down*]
cdcl-twl-full-restart-wl-D-GC-prog[*THEN fref-to-Down*])
subgoal by (*rule restart-abs-wl-D-pre*)
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by (*rule restart-abs-wl-D-pre-literals-are-*$\mathcal{L}_{in}$′)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

definition *cdcl-twl-stgy-restart-prog-wl-D*
:: *nat twl-st-wl* ⇒ *nat twl-st-wl nres*
where
⟨*cdcl-twl-stgy-restart-prog-wl-D* S_0 =
do {
(*brk*, *T*, -) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*brk*, -). ¬*brk*)
(λ(*brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop-wl-D S*;
(*brk*, *T*) ← *cdcl-twl-o-prog-wl-D T*;
(*T*, *n*) ← *restart-prog-wl-D T n brk*;
RETURN (*brk*, *T*, *n*)
})
(*False*, S_0::*nat twl-st-wl*, *0*);
RETURN T
}⟩

theorem *cdcl-twl-o-prog-wl-D-spec*′:
⟨(*cdcl-twl-o-prog-wl-D*, *cdcl-twl-o-prog-wl*) ∈
{(*S*,*S*′). (*S*,*S*′) ∈ *Id* ∧*literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} $\rightarrow_f$
⟨*bool-rel* $\times_r$ {(*T*′, *T*). *T* = *T*′ ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st T*) *T*}⟩ *nres-rel*⟩
apply (*intro frefI nres-relI*)
subgoal for *x y*
apply (*rule order-trans*)
apply (*rule cdcl-twl-o-prog-wl-D-spec*[*of all-atms-st x x*])
apply (*auto simp*: *prod-rel-def intro*: *conc-fun-R-mono*)
done
done

lemma *unit-propagation-outer-loop-wl-D-spec*′:
shows ⟨(*unit-propagation-outer-loop-wl-D*, *unit-propagation-outer-loop-wl*) ∈
{(*T*′, *T*). *T* = *T*′ ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st T*) *T*} $\rightarrow_f$
⟨{(*T*′, *T*). *T* = *T*′ ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st T*) *T*}⟩*nres-rel*⟩
apply (*intro frefI nres-relI*)
subgoal for *x y*
apply (*rule order-trans*)
apply (*rule unit-propagation-outer-loop-wl-D-spec*[*of all-atms-st x x*])
apply (*auto simp*: *prod-rel-def intro*: *conc-fun-R-mono*)
done
done

lemma *cdcl-twl-stgy-restart-prog-wl-D-cdcl-twl-stgy-restart-prog-wl*:

‹(*cdcl-twl-stgy-restart-prog-wl-D*, *cdcl-twl-stgy-restart-prog-wl*) ∈
  $\{(S, T). (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) $S\} \rightarrow_f$
  $\langle\{(S, T). (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) $S\}\rangle$*nres-rel*›
unfolding *uncurry-def cdcl-twl-stgy-restart-prog-wl-D-def*
  cdcl-twl-stgy-restart-prog-wl-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
  restart-prog-wl-D-restart-prog-wl[*THEN fref-to-Down-curry2*]
  cdcl-twl-o-prog-wl-D-spec′[*THEN fref-to-Down*]
  unit-propagation-outer-loop-wl-D-spec′[*THEN fref-to-Down*]
  WHILEIT-refine[**where** *R*=‹*bool-rel* $\times_r$ $\{(S, T). (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) $S\}$
$\times_r$ *nat-rel*›])
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-D-inv-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

definition *cdcl-twl-stgy-restart-prog-early-wl-D*
  :: *nat twl-st-wl* ⇒ *nat twl-st-wl nres*
where
  ‹*cdcl-twl-stgy-restart-prog-early-wl-D* S_0 = *do* {
    ebrk ← *RES UNIV*;
    (*ebrk*, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(\text{-},\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ T\ n}$
      (λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
      (λ(-, *brk*, *S*, *n*).
      do {
        T ← *unit-propagation-outer-loop-wl-D S*;
        (*brk*, *T*) ← *cdcl-twl-o-prog-wl-D T*;
        (*T*, *n*) ← *restart-prog-wl-D T n brk*;
        ebrk ← *RES UNIV*;
        RETURN (*ebrk*, *brk*, *T*, *n*)
      })
      (*ebrk*, *False*, S_0::*nat twl-st-wl*, *0*);
    if ¬*brk then do* {
      (*brk*, *T*, -) ← $WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ T\ n}$
 (λ(*brk*, -). ¬*brk*)
 (λ(*brk*, *S*, *n*).
 do {
   T ← *unit-propagation-outer-loop-wl-D S*;
   (*brk*, *T*) ← *cdcl-twl-o-prog-wl-D T*;
   (*T*, *n*) ← *restart-prog-wl-D T n brk*;
   RETURN (*brk*, *T*, *n*)
 })
 (*False*, *T*::*nat twl-st-wl*, *n*);
      RETURN T
    }
    else RETURN T
  }›

lemma *cdcl-twl-stgy-restart-prog-early-wl-D-cdcl-twl-stgy-restart-prog-early-wl*:
‹(*cdcl-twl-stgy-restart-prog-early-wl-D*, *cdcl-twl-stgy-restart-prog-early-wl*) ∈
{(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) S} $\rightarrow_f$
⟨{(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) S}⟩*nres-rel*›
unfolding *uncurry-def cdcl-twl-stgy-restart-prog-early-wl-D-def*
cdcl-twl-stgy-restart-prog-early-wl-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
restart-prog-wl-D-restart-prog-wl[*THEN fref-to-Down-curry2*]
cdcl-twl-o-prog-wl-D-spec'[*THEN fref-to-Down*]
unit-propagation-outer-loop-wl-D-spec'[*THEN fref-to-Down*]
WHILEIT-refine[**where** R=‹*bool-rel* $\times_r$ *bool-rel* $\times_r$ {(S, T). (S, T) ∈ *Id* ∧
literals-are-$\mathcal{L}_{in}$ (*all-atms-st* S) S} $\times_r$ *nat-rel*›]
WHILEIT-refine[**where** R=‹*bool-rel* $\times_r$ {(S, T). (S, T) ∈ *Id* ∧
literals-are-$\mathcal{L}_{in}$ (*all-atms-st* S) S} $\times_r$ *nat-rel*›])
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-D-inv-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-D-inv-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

definition *cdcl-twl-stgy-restart-prog-bounded-wl-D*
:: *nat twl-st-wl* ⇒ (*bool* × *nat twl-st-wl*) *nres*
where
‹*cdcl-twl-stgy-restart-prog-bounded-wl-D* S_0 = *do* {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, T, n) ← $WHILE_T{}^{\lambda(\text{-},\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(-, *brk*, S, n).
do {
T ← *unit-propagation-outer-loop-wl-D* S;
(*brk*, T) ← *cdcl-twl-o-prog-wl-D* T;
(T, n) ← *restart-prog-wl-D* T n *brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk*, T, n)
})
(*ebrk*, *False*, S_0::*nat twl-st-wl*, *0*);
RETURN (*brk*, T)
}›

lemma *cdcl-twl-stgy-restart-prog-bounded-wl-D-cdcl-twl-stgy-restart-prog-bounded-wl*:
‹(*cdcl-twl-stgy-restart-prog-bounded-wl-D*, *cdcl-twl-stgy-restart-prog-bounded-wl*) ∈
{(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) S} $\rightarrow_f$
⟨*bool-rel* $\times_r$ {(S, T). (S, T) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st* S) S}⟩*nres-rel*›
unfolding *uncurry-def cdcl-twl-stgy-restart-prog-bounded-wl-D-def*
cdcl-twl-stgy-restart-prog-bounded-wl-def
apply (*intro frefI nres-relI*)
apply (*refine-vcg*
restart-prog-wl-D-restart-prog-wl[*THEN fref-to-Down-curry2*]
cdcl-twl-o-prog-wl-D-spec′[*THEN fref-to-Down*]
unit-propagation-outer-loop-wl-D-spec′[*THEN fref-to-Down*]
WHILEIT-refine[**where** *R*=‹*bool-rel* $\times_r$ *bool-rel* $\times_r$ {(S, T). (S, T) ∈ *Id* ∧
literals-are-$\mathcal{L}_{in}$ (*all-atms-st* S) S} $\times_r$ *nat-rel*›])
subgoal by *auto*
subgoal unfolding *cdcl-twl-stgy-restart-abs-wl-D-inv-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

end

end
theory *Watched-Literals-Initialisation*
imports *Watched-Literals-List*
begin

1.4.6 Initialise Data structure

type-synonym ′*v twl-st-init* = ‹′*v twl-st* × ′*v clauses*›

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

fun *get-conflict-init* :: ‹′*v twl-st-init* ⇒ ′*v cconflict*› **where**
‹*get-conflict-init* ((-, -, -, D, -, -, -, -), -) = D›

fun *literals-to-update-init* :: ‹′*v twl-st-init* ⇒ ′*v clause*› **where**
‹*literals-to-update-init* ((-, -, -, -, -, -, -, Q), -) = Q›

fun *get-init-clauses-init* :: ‹′*v twl-st-init* ⇒ ′*v twl-cls multiset*› **where**
‹*get-init-clauses-init* ((-, N, -, -, -, -, -, -), -) = N›

fun *get-learned-clauses-init* :: ‹′*v twl-st-init* ⇒ ′*v twl-cls multiset*› **where**
‹*get-learned-clauses-init* ((-, -, U, -, -, -, -, -), -) = U›

fun *get-unit-init-clauses-init* :: ‹′*v twl-st-init* ⇒ ′*v clauses*› **where**
‹*get-unit-init-clauses-init* ((-, -, -, -, NE, -, -, -), -) = NE›

fun *get-unit-learned-clauses-init* :: ‹′*v twl-st-init* ⇒ ′*v clauses*› **where**
‹*get-unit-learned-clauses-init* ((-, -, -, -, -, UE, -, -), -) = UE›

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

fun *other-clauses-init* :: ‹'v twl-st-init ⇒ 'v clauses› **where**
‹other-clauses-init ((-, -, -, -, -, -, -), OC) = OC›

fun *add-to-init-clauses* :: ‹'v clause-l ⇒ 'v twl-st-init ⇒ 'v twl-st-init› **where**
‹add-to-init-clauses C ((M, N, U, D, NE, UE, WS, Q), OC) =
((M, add-mset (twl-clause-of C) N, U, D, NE, UE, WS, Q), OC)›

fun *add-to-unit-init-clauses* :: ‹'v clause ⇒ 'v twl-st-init ⇒ 'v twl-st-init› **where**
‹add-to-unit-init-clauses C ((M, N, U, D, NE, UE, WS, Q), OC) =
((M, N, U, D, add-mset C NE, UE, WS, Q), OC)›

fun *set-conflict-init* :: ‹'v clause-l ⇒ 'v twl-st-init ⇒ 'v twl-st-init› **where**
‹set-conflict-init C ((M, N, U, -, NE, UE, WS, Q), OC) =
((M, N, U, Some (mset C), add-mset (mset C) NE, UE, {#}, {#}), OC)›

fun *propagate-unit-init* :: ‹'v literal ⇒ 'v twl-st-init ⇒ 'v twl-st-init› **where**
‹propagate-unit-init L ((M, N, U, D, NE, UE, WS, Q), OC) =
((Propagated L {#L#} # M, N, U, D, add-mset {#L#} NE, UE, WS, add-mset (−L) Q), OC)›

fun *add-empty-conflict-init* :: ‹'v twl-st-init ⇒ 'v twl-st-init› **where**
‹add-empty-conflict-init ((M, N, U, D, NE, UE, WS, Q), OC) =
((M, N, U, Some {#}, NE, UE, WS, {#}), add-mset {#} OC)›

fun *add-to-clauses-init* :: ‹'v clause-l ⇒ 'v twl-st-init ⇒ 'v twl-st-init› **where**
‹add-to-clauses-init C ((M, N, U, D, NE, UE, WS, Q), OC) =
((M, add-mset (twl-clause-of C) N, U, D, NE, UE, WS, Q), OC)›

type-synonym 'v twl-st-l-init = ‹'v twl-st-l × 'v clauses›

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

fun *get-conflict-l-init* :: ‹'v twl-st-l-init ⇒ 'v cconflict› **where**
‹get-conflict-l-init ((-, -, D, -, -, -, -), -) = D›

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

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

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

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

fun *clauses-to-update-l-init* :: ‹'v twl-st-l-init ⇒ 'v clauses-to-update-l› **where**
‹clauses-to-update-l-init ((-, -, -, -, -, WS, -), -) = WS›

fun *other-clauses-l-init* :: ‹'v twl-st-l-init ⇒ 'v clauses› **where**
‹other-clauses-l-init ((-, -, -, -, -, -, -), OC) = OC›

fun *state$_W$-of-init* :: *'v twl-st-init* ⇒ *'v cdcl$_W$-restart-mset* **where**
state$_W$-of-init ((*M*, *N*, *U*, *C*, *NE*, *UE*, *Q*), *OC*) =
  (*M*, *clause* '# *N* + *NE* + *OC*, *clause* '# *U* + *UE*, *C*)

named-theorems *twl-st-init* ‹*Convertion for inital theorems*›

lemma [*twl-st-init*]:
  ‹*get-conflict-init* (*S*, *QC*) = *get-conflict S*›
  ‹*get-trail-init* (*S*, *QC*) = *get-trail S*›
  ‹*clauses-to-update-init* (*S*, *QC*) = *clauses-to-update S*›
  ‹*literals-to-update-init* (*S*, *QC*) = *literals-to-update S*›
  by (*solves* ‹*cases S*; *auto*›)+

lemma [*twl-st-init*]:
  ‹*clauses-to-update-init* (*add-to-unit-init-clauses* (*mset C*) *T*) = *clauses-to-update-init T*›
  ‹*literals-to-update-init* (*add-to-unit-init-clauses* (*mset C*) *T*) = *literals-to-update-init T*›
  ‹*get-conflict-init* (*add-to-unit-init-clauses* (*mset C*) *T*) = *get-conflict-init T*›
  apply (*cases T*; *auto simp*: *twl-st-inv.simps*; *fail*)+
  done
lemma [*twl-st-init*]:
  ‹*twl-st-inv* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *twl-st-inv* (*fst T*)›
  ‹*valid-enqueued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *valid-enqueued* (*fst T*)›
  ‹*no-duplicate-queued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *no-duplicate-queued* (*fst T*)›
  ‹*distinct-queued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *distinct-queued* (*fst T*)›
  ‹*confl-cands-enqueued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *confl-cands-enqueued* (*fst T*)›
  ‹*propa-cands-enqueued* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *propa-cands-enqueued* (*fst T*)›
  ‹*twl-st-exception-inv* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*)) ⟷ *twl-st-exception-inv* (*fst T*)›
    apply (*cases T*; *auto simp*: *twl-st-inv.simps*; *fail*)+
  apply (*cases* ‹*get-conflict-init T*›; *cases T*;
      auto simp: *twl-st-inv.simps twl-exception-inv.simps*; *fail*)+
  done

lemma [*twl-st-init*]:
  ‹*trail* (*state$_W$-of-init T*) = *get-trail-init T*›
  ‹*get-trail* (*fst T*) = *get-trail-init* (*T*)›
  ‹*conflicting* (*state$_W$-of-init T*) = *get-conflict-init T*›
  ‹*init-clss* (*state$_W$-of-init T*) = *clauses* (*get-init-clauses-init T*) + *get-unit-init-clauses-init T*
    + *other-clauses-init T*›
  ‹*learned-clss* (*state$_W$-of-init T*) = *clauses* (*get-learned-clauses-init T*) +
     get-unit-learned-clauses-init T›
  ‹*conflicting* (*state$_W$-of* (*fst T*)) = *conflicting* (*state$_W$-of-init T*)›
  ‹*trail* (*state$_W$-of* (*fst T*)) = *trail* (*state$_W$-of-init T*)›
  ‹*clauses-to-update* (*fst T*) = *clauses-to-update-init T*›
  ‹*get-conflict* (*fst T*) = *get-conflict-init T*›
  ‹*literals-to-update* (*fst T*) = *literals-to-update-init T*›
  by (*cases T*; *auto simp*: *cdcl$_W$-restart-mset-state*; *fail*)+

definition *twl-st-l-init* :: ‹('*v twl-st-l-init* × '*v twl-st-init*) *set*› **where**
  ‹*twl-st-l-init* = {(((*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*), *OC*), ((*M'*, *N'*, *C'*, *NE'*, *UE'*, *WS'*, *Q'*), *OC'*)).
    (*M* , *M'*) ∈ *convert-lits-l N* (*NE*+*UE*) ∧
    ((*N'*, *C'*, *NE'*, *UE'*, *WS'*, *Q'*), *OC'*) =
      ((*twl-clause-of* '# *init-clss-lf N*, *twl-clause-of* '# *learned-clss-lf N*,
        C, *NE*, *UE*, {#}, *Q*), *OC*)}›

lemma *twl-st-l-init-alt-def*:

```
  ‹(S, T) ∈ twl-st-l-init ⟷
    (fst S, fst T) ∈ twl-st-l None ∧ other-clauses-l-init S = other-clauses-init T›
  by (cases S; cases T) (auto simp: twl-st-l-init-def twl-st-l-def)

lemma [twl-st-init]:
  assumes ‹(S, T) ∈ twl-st-l-init›
  shows
   ‹get-conflict-init T = get-conflict-l-init S›
   ‹get-conflict (fst T) = get-conflict-l-init S›
   ‹literals-to-update-init T = literals-to-update-l-init S›
   ‹clauses-to-update-init T = {#}›
   ‹other-clauses-init T = other-clauses-l-init S›
   ‹lits-of-l (get-trail-init T) = lits-of-l (get-trail-l-init S)›
   ‹lit-of `# mset (get-trail-init T) = lit-of `# mset (get-trail-l-init S)›
   by (use assms in ‹solves ‹cases S; auto simp: twl-st-l-init-def››)+

definition twl-struct-invs-init :: ‹'v twl-st-init ⇒ bool› where
  ‹twl-struct-invs-init S ⟷
    (twl-st-inv (fst S) ∧
    valid-enqueued (fst S) ∧
    cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of-init S) ∧
    cdcl_W-restart-mset.no-smaller-propa (state_W-of-init S) ∧
    twl-st-exception-inv (fst S) ∧
    no-duplicate-queued (fst S) ∧
    distinct-queued (fst S) ∧
    confl-cands-enqueued (fst S) ∧
    propa-cands-enqueued (fst S) ∧
    (get-conflict-init S ≠ None ⟶ clauses-to-update-init S = {#} ∧ literals-to-update-init S = {#}) ∧
    entailed-clss-inv (fst S) ∧
    clauses-to-update-inv (fst S) ∧
    past-invs (fst S))
  ›

lemma state_W-of-state_W-of-init:
  ‹other-clauses-init W = {#} ⟹ state_W-of (fst W) = state_W-of-init W›
  by (cases W) auto

lemma twl-struct-invs-init-twl-struct-invs:
  ‹other-clauses-init W = {#} ⟹ twl-struct-invs-init W ⟹ twl-struct-invs (fst W)›
  unfolding twl-struct-invs-def twl-struct-invs-init-def
  apply (subst state_W-of-state_W-of-init; assumption?)+
  apply (intro iffI impI conjI)
  by (clarsimp simp: twl-st-init)+

lemma twl-struct-invs-init-add-mset:
  assumes ‹twl-struct-invs-init (S, QC)› and [simp]: ‹distinct-mset C› and
    count-dec: ‹count-decided (trail (state_W-of S)) = 0›
  shows ‹twl-struct-invs-init (S, add-mset C QC)›
proof -
  have
    st-inv: ‹twl-st-inv S› and
    valid: ‹valid-enqueued S› and
    struct: ‹cdcl_W-restart-mset.cdcl_W-all-struct-inv (state_W-of-init (S, QC))› and
    smaller: ‹cdcl_W-restart-mset.no-smaller-propa (state_W-of-init (S, QC))› and
    excep: ‹twl-st-exception-inv S› and
    no-dup: ‹no-duplicate-queued S› and
```

dist: ⟨*distinct-queued S*⟩ **and**
cands-confl: ⟨*confl-cands-enqueued S*⟩ **and**
cands-propa: ⟨*propa-cands-enqueued S*⟩ **and**
confl: ⟨*get-conflict S* ≠ *None* ⟶ *clauses-to-update S* = {#} ∧ *literals-to-update S* = {#}⟩ **and**
unit: ⟨*entailed-clss-inv S*⟩ **and**
to-upd: ⟨*clauses-to-update-inv S*⟩ **and**
past: ⟨*past-invs S*⟩
using *assms* **unfolding** *twl-struct-invs-init-def fst-conv*
by (*auto simp add*: *twl-st-init*)

show *?thesis*
unfolding *twl-struct-invs-init-def fst-conv*
apply (*intro conjI*)
subgoal by (*rule st-inv*)
subgoal by (*rule valid*)
subgoal using *struct count-dec no-dup*
by (*cases S*)
(*auto 5 5 simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def clauses-def*
$cdcl_W$*-restart-mset-state* $cdcl_W$*-restart-mset.no-strange-atm-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def* $cdcl_W$*-restart-mset.reasons-in-clauses-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def all-decomposition-implies-def*)
subgoal using *smaller count-dec* **by** (*cases S*)(*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def clauses-def*
$cdcl_W$*-restart-mset-state*)
subgoal by (*rule excep*)
subgoal by (*rule no-dup*)
subgoal by (*rule dist*)
subgoal by (*rule cands-confl*)
subgoal by (*rule cands-propa*)
subgoal using *confl* **by** (*auto simp*: *twl-st-init*)
subgoal by (*rule unit*)
subgoal by (*rule to-upd*)
subgoal by (*rule past*)
done
qed

fun *add-empty-conflict-init-l* :: ⟨′*v twl-st-l-init* ⇒ ′*v twl-st-l-init*⟩ **where**
add-empty-conflict-init-l-def[*simp del*]:
⟨*add-empty-conflict-init-l* ((*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*), *OC*) =
((*M*, *N*, *Some* {#}, *NE*, *UE*, *WS*, {#}), *add-mset* {#} *OC*)⟩

fun *propagate-unit-init-l* :: ⟨′*v literal* ⇒ ′*v twl-st-l-init* ⇒ ′*v twl-st-l-init*⟩ **where**
propagate-unit-init-l-def[*simp del*]:
⟨*propagate-unit-init-l L* ((*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*), *OC*) =
((*Propagated L 0* # *M*, *N*, *D*, *add-mset* {#*L*#} *NE*, *UE*, *WS*, *add-mset* (−*L*) *Q*), *OC*)⟩

fun *already-propagated-unit-init-l* :: ⟨′*v clause* ⇒ ′*v twl-st-l-init* ⇒ ′*v twl-st-l-init*⟩ **where**
already-propagated-unit-init-l-def[*simp del*]:
⟨*already-propagated-unit-init-l C* ((*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*), *OC*) =
((*M*, *N*, *D*, *add-mset C NE*, *UE*, *WS*, *Q*), *OC*)⟩

fun *set-conflict-init-l* :: ‹'v clause-l ⇒ 'v twl-st-l-init ⇒ 'v twl-st-l-init› **where**
set-conflict-init-l-def[*simp del*]:
‹*set-conflict-init-l C ((M, N, -, NE, UE, WS, Q), OC) =*
((M, N, Some (mset C), add-mset (mset C) NE, UE, {#}, {#}), OC)›

fun *add-to-clauses-init-l* :: ‹'v clause-l ⇒ 'v twl-st-l-init ⇒ 'v twl-st-l-init nres› **where**
add-to-clauses-init-l-def[*simp del*]:
‹*add-to-clauses-init-l C ((M, N, -, NE, UE, WS, Q), OC) = do {*
i ← get-fresh-index N;
RETURN ((M, fmupd i (C, True) N, None, NE, UE, WS, Q), OC)
}›

fun *add-to-other-init* **where**
‹*add-to-other-init C (S, OC) = (S, add-mset (mset C) OC)*›

lemma *fst-add-to-other-init* [*simp*]: ‹*fst (add-to-other-init a T) = fst T*›
by (*cases T*) *auto*

definition *init-dt-step* :: ‹'v clause-l ⇒ 'v twl-st-l-init ⇒ 'v twl-st-l-init nres› **where**
‹*init-dt-step C S =*
(case get-conflict-l-init S of
None ⇒
if length C = 0
then RETURN (add-empty-conflict-init-l S)
else if length C = 1
then
let L = hd C in
if undefined-lit (get-trail-l-init S) L
then RETURN (propagate-unit-init-l L S)
else if L ∈ lits-of-l (get-trail-l-init S)
then RETURN (already-propagated-unit-init-l (mset C) S)
else RETURN (set-conflict-init-l C S)
else
add-to-clauses-init-l C S
| Some D ⇒
RETURN (add-to-other-init C S))›

definition *init-dt* :: ‹'v clause-l list ⇒ 'v twl-st-l-init ⇒ 'v twl-st-l-init nres› **where**
‹*init-dt CS S = nfoldli CS (λ-. True) init-dt-step S*›

thm *nfoldli.simps*

definition *init-dt-pre* **where**
‹*init-dt-pre CS SOC ⟷*
(∃ T. (SOC, T) ∈ twl-st-l-init ∧
(∀ C ∈ set CS. distinct C) ∧
twl-struct-invs-init T ∧
clauses-to-update-l-init SOC = {#} ∧
(∀ s∈set (get-trail-l-init SOC). ¬is-decided s) ∧
(get-conflict-l-init SOC = None ⟶
literals-to-update-l-init SOC = uminus ‘# lit-of ‘# mset (get-trail-l-init SOC)) ∧
twl-list-invs (fst SOC) ∧
twl-stgy-invs (fst T) ∧
(other-clauses-l-init SOC ≠ {#} ⟶ get-conflict-l-init SOC ≠ None))›

lemma *init-dt-pre-ConsD*: ‹*init-dt-pre* (*a* # *CS*) *SOC* ⟹ *init-dt-pre CS SOC* ∧ *distinct a*›
  unfolding *init-dt-pre-def*
  apply *normalize-goal+*
  by *fastforce*

definition *init-dt-spec* **where**
  ‹*init-dt-spec CS SOC SOC′* ⟷
    (∃ *T′*. (*SOC′*, *T′*) ∈ *twl-st-l-init* ∧
        twl-struct-invs-init T′ ∧
        clauses-to-update-l-init SOC′ = {#} ∧
        (∀ *s*∈*set* (*get-trail-l-init SOC′*). ¬*is-decided s*) ∧
        (*get-conflict-l-init SOC′* = *None* ⟶
          literals-to-update-l-init SOC′ = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init SOC′*)) ∧
        (*mset* ‘# *mset CS* + *mset* ‘# *ran-mf* (*get-clauses-l-init SOC*) + *other-clauses-l-init SOC* +
            get-unit-clauses-l-init SOC =
         mset ‘# *ran-mf* (*get-clauses-l-init SOC′*) + *other-clauses-l-init SOC′* +
            get-unit-clauses-l-init SOC′) ∧
        learned-clss-lf (*get-clauses-l-init SOC*) = *learned-clss-lf* (*get-clauses-l-init SOC′*) ∧
        get-learned-unit-clauses-l-init SOC′ = *get-learned-unit-clauses-l-init SOC* ∧
        twl-list-invs (*fst SOC′*) ∧
        twl-stgy-invs (*fst T′*) ∧
        (*other-clauses-l-init SOC′* ≠ {#} ⟶ *get-conflict-l-init SOC′* ≠ *None*) ∧
        ({#} ∈# *mset* ‘# *mset CS* ⟶ *get-conflict-l-init SOC′* ≠ *None*) ∧
        (*get-conflict-l-init SOC* ≠ *None* ⟶ *get-conflict-l-init SOC* = *get-conflict-l-init SOC′*))›

lemma *twl-struct-invs-init-add-to-other-init*:
  assumes
    dist: ‹*distinct a*› **and**
    lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
    invs: ‹*twl-struct-invs-init T*›
  shows
    ‹*twl-struct-invs-init* (*add-to-other-init a T*)›
      (**is** *?twl-struct-invs-init*)
proof −
  obtain *M N U D NE UE Q OC WS* **where**
    T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*), *OC*)›
    by (*cases T*) *auto*
  have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)›
    using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
  then have [*simp*]:
  ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, *add-mset* (*mset a*) (*clauses N* + *NE* + *OC*), *clauses U* + *UE*, *D*)›
    using *dist*
    by (*auto simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
      $cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
      $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
      $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def all-decomposition-implies-def*
      clauses-def $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*
      $cdcl_W$*-restart-mset.reasons-in-clauses-def*)

  have ‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)›
    using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
  then have [*simp*]:
    ‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *add-mset* (*mset a*) (*clauses N* + *NE* + *OC*),
      clauses U + *UE*, *D*)›

using *lev*
by (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def cdcl*$_W$*-restart-mset-state clauses-def T count-decided-0-iff*)
show *?twl-struct-invs-init*
using *invs*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv add-to-other-init.simps state*$_W$*-of-init.simps get-conflict.simps*
by *clarsimp*
qed

lemma *invariants-init-state*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
wf: ‹∀ *C* ∈# *get-clauses* (*fst T*). *struct-wf-twl-cls C*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* '# *lit-of* '# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
n-d: ‹*no-dup* (*get-trail-init T*)›
shows ‹*propa-cands-enqueued* (*fst T*)› **and** ‹*confl-cands-enqueued* (*fst T*)› **and** ‹*twl-st-inv* (*fst T*)›
‹*clauses-to-update-inv* (*fst T*)› **and** ‹*past-invs* (*fst T*)› **and** ‹*distinct-queued* (*fst T*)› **and**
‹*valid-enqueued* (*fst T*)› **and** ‹*twl-st-exception-inv* (*fst T*)› **and** ‹*no-duplicate-queued* (*fst T*)›
proof −
obtain *M N U NE UE OC D* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *uminus* '# *lit-of* '# *mset M*), *OC*)›
using *MQ WS* **by** (*cases T*) *auto*
let *?Q* = ‹*uminus* '# *lit-of* '# *mset M*›

have [*iff*]: ‹*M* = *M'* @ *Decided K* # *Ma* ⟷ *False*› **for** *M' K Ma*
using *lev* **by** (*auto simp*: *count-decided-0-iff T*)

have *struct*: ‹*struct-wf-twl-cls C*› **if** ‹*C* ∈# *N* + *U*› **for** *C*
using *wf that* **by** (*simp add*: *T twl-st-inv.simps*)
let *?T* = ‹*fst T*›
have [*simp*]: ‹*propa-cands-enqueued ?T*› **if** *D*: ‹*D* = *None*›
unfolding *propa-cands-enqueued.simps Ball-def T fst-conv D*
apply − **apply** (*intro conjI impI allI*)
subgoal for *x C*
using *struct*[*of C*]
apply (*case-tac C*; *auto simp*: *uminus-lit-swap lits-of-def size-2-iff true-annots-true-cls-def-iff-negation-in-model Ball-def remove1-mset-add-mset-If all-conj-distrib conj-disj-distribR ex-disj-distrib split*: *if-splits*)
done
done
then show ‹*propa-cands-enqueued ?T*›
by (*cases D*) (*auto simp*: *T*)

have [*simp*]: ‹*confl-cands-enqueued ?T*› **if** *D*: ‹*D* = *None*›
unfolding *confl-cands-enqueued.simps Ball-def T D fst-conv*
apply − **apply** (*intro conjI impI allI*)
subgoal for *x*
using *struct*[*of x*]
by (*case-tac x*; *case-tac* ‹*watched x*›; *auto simp*: *uminus-lit-swap lits-of-def*)
done
then show ‹*confl-cands-enqueued ?T*›
by (*cases D*) (*auto simp*: *T*)

have [*simp*]: ‹*get-level M L* = *0*› **for** *L*
using *lev* **by** (*auto simp*: *T count-decided-0-iff*)
show [*simp*]: ‹*twl-st-inv ?T*›
unfolding *T fst-conv twl-st-inv.simps Ball-def*
apply − **apply** (*intro conjI impI allI*)
subgoal using *wf* **by** (*auto simp*: *T*)
subgoal for *C*
by (*cases C*)
(*auto simp*: *T twl-st-inv.simps twl-lazy-update.simps twl-is-an-exception-def lits-of-def uminus-lit-swap*)
subgoal for *C*
using *lev* **by** (*cases C*)
(*auto simp*: *T twl-st-inv.simps twl-lazy-update.simps*)
done
have [*simp*]: ‹{#*C* ∈# *N*. *clauses-to-update-prop* {#− *lit-of x*. *x* ∈# *mset M*#} *M* (*L*, *C*)#} = {#}›
for *L N*
by (*auto simp*: *filter-mset-empty-conv clauses-to-update-prop.simps lits-of-def uminus-lit-swap*)
have ‹*clauses-to-update-inv ?T*› **if** *D*: ‹*D* = *None*›
unfolding *T D*
by (*auto simp*: *filter-mset-empty-conv lits-of-def uminus-lit-swap*)
then show ‹*clauses-to-update-inv* (*fst T*)›
by (*cases D*) (*auto simp*: *T*)

show ‹*past-invs ?T*›
by (*auto simp*: *T past-invs.simps*)

show ‹*distinct-queued ?T*›
using *WS n-d* **by** (*auto simp*: *T no-dup-distinct-uminus*)
show ‹*valid-enqueued ?T*›
using *lev* **by** (*auto simp*: *T lits-of-def*)

show ‹*twl-st-exception-inv* (*fst T*)›
unfolding *T fst-conv twl-st-exception-inv.simps Ball-def*
apply − **apply** (*intro conjI impI allI*)
apply (*case-tac x*; *cases D*)
by (*auto simp*: *T twl-exception-inv.simps lits-of-def uminus-lit-swap*)

show ‹*no-duplicate-queued* (*fst T*)›
by (*auto simp*: *T*)
qed

lemma *twl-struct-invs-init-init-state*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
wf: ‹∀ *C* ∈# *get-clauses* (*fst T*). *struct-wf-twl-cls C*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
struct-invs: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of-init T*)› **and**
‹$cdcl_W$*-restart-mset.no-smaller-propa* ($state_W$*-of-init T*)› **and**
‹*entailed-clss-inv* (*fst T*)› **and**
‹*get-conflict-init T* ≠ *None* ⟶ *clauses-to-update-init T* = {#} ∧ *literals-to-update-init T* = {#}›
shows ‹*twl-struct-invs-init T*›
proof −
have *n-d*: ‹*no-dup* (*get-trail-init T*)›
using *struct-invs* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*

$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* **by** (*cases T*) (*auto simp*: *trail.simps*)
then show *?thesis*
using *invariants-init-state*[*OF lev wf MQ WS n-d*] *assms* **unfolding** *twl-struct-invs-init-def*
by *fast+*
qed

lemma *twl-struct-invs-init-add-to-unit-init-clauses*:
assumes
dist: ⟨*distinct a*⟩ **and**
lev: ⟨*count-decided* (*get-trail* (*fst T*)) = *0*⟩ **and**
invs: ⟨*twl-struct-invs-init T*⟩ **and**
ex: ⟨∃ *L* ∈ *set a*. *L* ∈ *lits-of-l* (*get-trail-init T*)⟩
shows
⟨*twl-struct-invs-init* (*add-to-unit-init-clauses* (*mset a*) *T*)⟩
(**is** *?all-struct*)
proof −
obtain *M N U D NE UE Q OC WS* **where**
T: ⟨*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*), *OC*)⟩
by (*cases T*) *auto*
have ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)⟩
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, *add-mset* (*mset a*) (*clauses N* + *NE* + *OC*), *clauses U* + *UE*, *D*)⟩
using *twl-struct-invs-init-add-to-other-init*[*OF dist lev invs*]
unfolding *T twl-struct-invs-init-def*
by *simp*

have ⟨$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)⟩
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
⟨$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *add-mset* (*mset a*) (*clauses N* + *NE* + *OC*),
clauses U + *UE*, *D*)⟩
using *lev*
by (*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def* $cdcl_W$*-restart-mset-state*
clauses-def T count-decided-0-iff)
have [*simp*]: ⟨*confl-cands-enqueued* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) ⟷
confl-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
⟨*propa-cands-enqueued* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) ⟷
propa-cands-enqueued (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
⟨*twl-st-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) ⟷
twl-st-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
⟨⋀*x*. *twl-exception-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) *x* ⟷
twl-exception-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) *x*⟩
⟨*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) ⟷
clauses-to-update-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
⟨*past-invs* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) ⟷
past-invs (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
by (*cases D*; *auto simp*: *twl-st-inv.simps twl-exception-inv.simps past-invs.simps*; *fail*)+
have [*simp*]: ⟨*entailed-clss-inv* (*M*, *N*, *U*, *D*, *add-mset* (*mset a*) *NE*, *UE*, *WS*, *Q*) ⟷
entailed-clss-inv (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)⟩
using *ex count-decided-ge-get-level*[*of M*] *lev* **by** (*cases D*) (*auto simp*: *T*)
show *?all-struct*
using *invs ex*
unfolding *twl-struct-invs-init-def T*

unfolding *fst-conv add-to-other-init.simps* $state_W$*-of-init.simps get-conflict.simps*
by (*clarsimp simp del*: *entailed-clss-inv.simps*)
qed

lemma *twl-struct-invs-init-set-conflict-init*:
assumes
dist: ‹*distinct C*› **and**
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
ex: ‹$\forall L \in$ *set C*. $-L \in$ *lits-of-l* (*get-trail-init T*)› **and**
nempty: ‹$C \neq []$›
shows
‹*twl-struct-invs-init* (*set-conflict-init C T*)›
(**is** *?all-struct*)
proof −
obtain *M N U D NE UE Q OC WS* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*), *OC*)›
by (*cases T*) *auto*
have ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*M*, *add-mset* (*mset C*) (*clauses N* + *NE* + *OC*),
clauses U + *UE*, *Some* (*mset C*))›
using *dist ex*
unfolding *T twl-struct-invs-init-def*
by (*auto 5 5 simp*: $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def all-decomposition-implies-def*
clauses-def $cdcl_W$*-restart-mset*.$cdcl_W$*-learned-clause-alt-def*
true-annots-true-cls-def-iff-negation-in-model)

have ‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *add-mset* (*mset C*) (*clauses N* + *NE* + *OC*),
clauses U + *UE*, *Some* (*mset C*))›
using *lev*
by (*auto simp*: $cdcl_W$*-restart-mset.no-smaller-propa-def* $cdcl_W$*-restart-mset-state*
clauses-def T count-decided-0-iff)
let *?T* = ‹(*M*, *N*, *U*, *Some* (*mset C*), *add-mset* (*mset C*) *NE*, *UE*, {#}, {#})›

have [*simp*]: ‹*confl-cands-enqueued ?T*›
‹*propa-cands-enqueued ?T*›
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) $\Longrightarrow$ *twl-st-inv ?T*›
‹$\bigwedge x$. *twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) *x* $\Longrightarrow$ *twl-exception-inv ?T x*›
‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) $\Longrightarrow$ *clauses-to-update-inv ?T*›
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) $\Longrightarrow$ *past-invs ?T*›
by (*auto simp*: *twl-st-inv.simps twl-exception-inv.simps past-invs.simps*; *fail*)+
have [*simp*]: ‹*entailed-clss-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) $\Longrightarrow$ *entailed-clss-inv ?T*›
using *ex count-decided-ge-get-level*[*of M*] *lev nempty* **by** (*auto simp*: *T*)
show *?all-struct*
using *invs ex*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv add-to-other-init.simps* $state_W$*-of-init.simps get-conflict.simps*
by (*clarsimp simp del*: *entailed-clss-inv.simps*)

qed

lemma *twl-struct-invs-init-propagate-unit-init*:
assumes
lev: ⟨*count-decided* (*get-trail-init* *T*) = *0*⟩ **and**
invs: ⟨*twl-struct-invs-init* *T*⟩ **and**
undef: ⟨*undefined-lit* (*get-trail-init* *T*) *L*⟩ **and**
confl: ⟨*get-conflict-init* *T* = *None*⟩ **and**
MQ: ⟨*literals-to-update-init* *T* = *uminus* '# *lit-of* '# *mset* (*get-trail-init* *T*)⟩ **and**
WS: ⟨*clauses-to-update-init* *T* = {#}⟩
shows
⟨*twl-struct-invs-init* (*propagate-unit-init* *L* *T*)⟩
(**is** *?all-struct*)
proof −
obtain *M N U NE UE OC WS* **where**
T: ⟨*T* = ((*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *uminus* '# *lit-of* '# *mset* *M*), *OC*)⟩
using *confl MQ* **by** (*cases T*) *auto*
let *?Q* = ⟨*uminus* '# *lit-of* '# *mset* *M*⟩
have [*iff*]: ⟨− *L* ∈ *lits-of-l* *M* ⟷ *False*⟩
using *undef* **by** (*auto simp*: *T Decided-Propagated-in-iff-in-lits-of-l*)
have [*simp*]: ⟨*get-all-ann-decomposition* *M* = [([], *M*)]⟩
by (*rule no-decision-get-all-ann-decomposition*) (*use lev* **in** ⟨*auto simp*: *T count-decided-0-iff*⟩)
have *H*: ⟨*a* @ *Propagated* *L'* *mark'* # *b* = *Propagated* *L* *mark* # *M* ⟷
(*a* = [] ∧ *L* = *L'* ∧ *mark* = *mark'* ∧ *b* = *M*) ∨
(*a* ≠ [] ∧ *hd* *a* = *Propagated* *L* *mark* ∧ *tl* *a* @ *Propagated* *L'* *mark'* # *b* = *M*)⟩
for *a mark mark' L' b*
using *undef* **by** (*cases a*) (*auto simp*: *T atm-of-eq-atm-of*)
have ⟨$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*M*, *clauses* *N* + *NE* + *OC*, *clauses* *U* + *UE*, *None*)⟩
and
excep: ⟨*twl-st-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *?Q*)⟩ **and**
st-inv: ⟨*twl-st-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *?Q*)⟩
using *invs confl* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
⟨$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*M*, *add-mset* {#*L*#} (*clauses* *N* + *NE* + *OC*),
clauses *U* + *UE*, *None*)⟩ **and**
n-d: ⟨*no-dup* *M*⟩
by (*auto simp*: $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* *all-decomposition-implies-def*
clauses-def $cdcl_W$*-restart-mset*.$cdcl_W$*-learned-clause-alt-def*
$cdcl_W$*-restart-mset.reasons-in-clauses-def*)
then have [*simp*]:
⟨$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*Propagated* *L* {#*L*#} # *M*,
add-mset {#*L*#} (*clauses* *N* + *NE* + *OC*), *clauses* *U* + *UE*, *None*)⟩
using *undef* **by** (*auto simp*: $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def* *T H*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* *all-decomposition-implies-def*
clauses-def $cdcl_W$*-restart-mset*.$cdcl_W$*-learned-clause-alt-def*
consistent-interp-insert-iff)
have [*iff*]: ⟨*Propagated* *L* {#*L*#} # *M* = *M'* @ *Decided* *K* # *Ma* ⟷ *False*⟩ **for** *M' K Ma*
using *lev* **by** (*cases M'*) (*auto simp*: *count-decided-0-iff T*)
have ⟨$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *clauses* *N* + *NE* + *OC*, *clauses* *U* + *UE*, *None*)⟩
using *invs confl* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:

⟨$cdcl_W$-restart-mset.no-smaller-propa (Propagated L {#L#} # M, add-mset {#L#} (clauses N + NE + OC),
clauses U + UE, None)⟩
using *lev*
by (*auto simp*: $cdcl_W$-restart-mset.no-smaller-propa-def $cdcl_W$-restart-mset-state clauses-def T count-decided-0-iff)

have ⟨$cdcl_W$-restart-mset.no-smaller-propa (M, clauses N + NE + OC, clauses U + UE, None)⟩
using *invs confl* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
⟨$cdcl_W$-restart-mset.no-smaller-propa (Propagated L {#L#} # M, add-mset {#L#} (clauses N + NE + OC),
clauses U + UE, None)⟩
using *lev*
by (*auto simp*: $cdcl_W$-restart-mset.no-smaller-propa-def $cdcl_W$-restart-mset-state clauses-def T count-decided-0-iff)
let *?S* = ⟨(M, N, U, None, NE, UE, WS, ?Q)⟩
let *?T* = ⟨(Propagated L {#L#} # M, N, U, None, add-mset {#L#} NE, UE, WS, add-mset (−L) ?Q)⟩

have *struct*: ⟨struct-wf-twl-cls C⟩ **if** ⟨C ∈# N + U⟩ **for** *C*
using *st-inv that* **by** (*simp add*: *twl-st-inv.simps*)
have ⟨entailed-clss-inv (fst T)⟩
using *invs* **unfolding** *T twl-struct-invs-init-def fst-conv* **by** *fast*
then have *ent*: ⟨entailed-clss-inv (fst (propagate-unit-init L T))⟩
using *lev* **by** (*auto simp*: *T get-level-cons-if*)
show ⟨twl-struct-invs-init (propagate-unit-init L T)⟩
apply (*rule twl-struct-invs-init-init-state*)
subgoal using *lev* **by** (*auto simp*: *T*)
subgoal using *struct* **by** (*auto simp*: *T*)
subgoal using *MQ* **by** (*auto simp*: *T*)
subgoal using *WS* **by** (*auto simp*: *T*)
subgoal by (*simp add*: *T*)
subgoal by (*auto simp*: *T*)
subgoal by (*rule ent*)
subgoal by (*auto simp*: *T*)
done
qed

named-theorems *twl-st-l-init*
lemma [*twl-st-l-init*]:
⟨clauses-to-update-l-init (already-propagated-unit-init-l C S) = clauses-to-update-l-init S⟩
⟨get-trail-l-init (already-propagated-unit-init-l C S) = get-trail-l-init S⟩
⟨get-conflict-l-init (already-propagated-unit-init-l C S) = get-conflict-l-init S⟩
⟨other-clauses-l-init (already-propagated-unit-init-l C S) = other-clauses-l-init S⟩
⟨clauses-to-update-l-init (already-propagated-unit-init-l C S) = clauses-to-update-l-init S⟩
⟨literals-to-update-l-init (already-propagated-unit-init-l C S) = literals-to-update-l-init S⟩
⟨get-clauses-l-init (already-propagated-unit-init-l C S) = get-clauses-l-init S⟩
⟨get-unit-clauses-l-init (already-propagated-unit-init-l C S) = add-mset C (get-unit-clauses-l-init S)⟩
⟨get-learned-unit-clauses-l-init (already-propagated-unit-init-l C S) =
get-learned-unit-clauses-l-init S⟩
⟨get-conflict-l-init (T, OC) = get-conflict-l T⟩
by (*solves* ⟨cases S; cases T; auto simp: already-propagated-unit-init-l-def⟩)+

lemma [*twl-st-l-init*]:

‹(*V*, *W*) ∈ *twl-st-l-init* ⟹
count-decided (*get-trail-init W*) = *count-decided* (*get-trail-l-init V*)›
by (*auto simp*: *twl-st-l-init-def*)

lemma [*twl-st-l-init*]:
‹*get-conflict-l* (*fst T*) = *get-conflict-l-init T*›
‹*literals-to-update-l* (*fst T*) = *literals-to-update-l-init T*›
‹*clauses-to-update-l* (*fst T*) = *clauses-to-update-l-init T*›
by (*cases T*; *auto*; *fail*)+

lemma *entailed-clss-inv-add-to-unit-init-clauses*:
‹*count-decided* (*get-trail-init T*) = *0* ⟹ *C* ≠ [] ⟹ *hd C* ∈ *lits-of-l* (*get-trail-init T*) ⟹
entailed-clss-inv (*fst T*) ⟹ *entailed-clss-inv* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*))›
using *count-decided-ge-get-level*[*of* ‹*get-trail-init T*›]
by (*cases T*; *cases C*; *auto simp*: *twl-st-inv.simps twl-exception-inv.simps*)

lemma *convert-lits-l-no-decision-iff*: ‹(*S*, *T*) ∈ *convert-lits-l M N* ⟹
(∀ *s*∈*set T*. ¬ *is-decided s*) ⟷
(∀ *s*∈*set S*. ¬ *is-decided s*)›
unfolding *convert-lits-l-def*
by (*induction rule*: *list-rel-induct*)
(*auto simp*: *dest*!: *p2relD*)

lemma *twl-st-l-init-no-decision-iff*:
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹
(∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*) ⟷
(∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*)›
by (*subst convert-lits-l-no-decision-iff*[*of* - - ‹*get-clauses-l-init S*›
‹*get-unit-clauses-l-init S*›])
(*auto simp*: *twl-st-l-init-def*)

lemma *twl-st-l-init-defined-lit*[*twl-st-l-init*]:
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹
defined-lit (*get-trail-init T*) = *defined-lit* (*get-trail-l-init S*)›
by (*auto simp*: *twl-st-l-init-def*)

lemma [*twl-st-l-init*]:
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹ *get-learned-clauses-init T* = {#} ⟷ *learned-clss-l* (*get-clauses-l-init S*) = {#}›
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹ *get-unit-learned-clauses-init T* = {#} ⟷ *get-learned-unit-clauses-l-init S* = {#}
›
by (*cases S*; *cases T*; *auto simp*: *twl-st-l-init-def*; *fail*)+

lemma *init-dt-pre-already-propagated-unit-init-l*:
assumes
hd-C: ‹*hd C* ∈ *lits-of-l* (*get-trail-l-init S*)› **and**
pre: ‹*init-dt-pre CS S*› **and**
nempty: ‹*C* ≠ []› **and**
dist-C: ‹*distinct C*› **and**
lev: ‹*count-decided* (*get-trail-l-init S*) = *0*›
shows
‹*init-dt-pre CS* (*already-propagated-unit-init-l* (*mset C*) *S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [*C*] *S* (*already-propagated-unit-init-l* (*mset C*) *S*)› (**is** *?spec*)
proof −

obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
dist: ‹*Ball* (*set CS*) *distinct*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* '# *lit-of* '# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC'-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal*+
by *presburger*
obtain *M N D NE UE Q U OC* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *Q*), *OC*)›
by (*cases S*) *auto*
have [*simp*]: ‹*twl-list-invs* (*fst* (*already-propagated-unit-init-l* (*mset C*) *S*))›
using *add-inv* **by** (*auto simp*: *already-propagated-unit-init-l-def S*
twl-list-invs-def)
have [*simp*]: ‹(*already-propagated-unit-init-l* (*mset C*) *S*, *add-to-unit-init-clauses* (*mset C*) *T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*)
(*auto simp*: *twl-st-l-init-def already-propagated-unit-init-l-def*
convert-lits-l-extend-mono)
have *dec'*: ‹∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*›
using *SOC-T dec* **by** (*subst twl-st-l-init-no-decision-iff*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*))›
using *stgy-inv dec'* **unfolding** *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def* $cdcl_W$*-restart-mset.no-smaller-confl-def*
by (*cases T*)
(*auto simp*: $cdcl_W$*-restart-mset-state clauses-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*add-to-unit-init-clauses* (*mset C*) *T*)›
apply (*rule twl-struct-invs-init-add-to-unit-init-clauses*)
using *inv hd-C nempty dist-C lev SOC-T dec'*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff intro*: *bexI*[*of* - ‹*hd C*›])
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*add-to-unit-init-clauses* (*mset C*) *T*›])
using *dist WS dec in-literals-to-update OC'-empty* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*add-to-unit-init-clauses* (*mset C*) *T*›])
using *dist WS dec in-literals-to-update OC'-empty nempty*
by (*auto simp*: *twl-st-init twl-st-l-init*)
qed

lemma (**in** −) *twl-stgy-invs-backtrack-lvl-0*:
‹*count-decided* (*get-trail T*) = *0* ⟹ *twl-stgy-invs T*›
using *count-decided-ge-get-level*[*of* ‹*get-trail T*›]
by (*cases T*)
(*auto simp*: *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.no-smaller-confl-def* $cdcl_W$*-restart-mset-state*

cdcl$_W$-restart-mset.conflict-non-zero-unless-level-0-def)

lemma [*twl-st-l-init*]:
‹*clauses-to-update-l-init* (*propagate-unit-init-l L S*) = *clauses-to-update-l-init S*›
‹*get-trail-l-init* (*propagate-unit-init-l L S*) = *Propagated L 0* # *get-trail-l-init S*›
‹*literals-to-update-l-init* (*propagate-unit-init-l L S*) =
add-mset (−*L*) (*literals-to-update-l-init S*)›
‹*get-conflict-l-init* (*propagate-unit-init-l L S*) = *get-conflict-l-init S*›
‹*clauses-to-update-l-init* (*propagate-unit-init-l L S*) = *clauses-to-update-l-init S*›
‹*other-clauses-l-init* (*propagate-unit-init-l L S*) = *other-clauses-l-init S*›
‹*get-clauses-l-init* (*propagate-unit-init-l L S*) = *get-clauses-l-init S*›
‹*get-learned-unit-clauses-l-init* (*propagate-unit-init-l L S*) = *get-learned-unit-clauses-l-init S*›
‹*get-unit-clauses-l-init* (*propagate-unit-init-l L S*) = *add-mset* {#*L*#} (*get-unit-clauses-l-init S*)›
by (*cases S*; *auto simp*: *propagate-unit-init-l-def*; *fail*)+

lemma *init-dt-pre-propagate-unit-init*:
assumes
hd-C: ‹*undefined-lit* (*get-trail-l-init S*) *L*› **and**
pre: ‹*init-dt-pre CS S*› **and**
lev: ‹*count-decided* (*get-trail-l-init S*) = *0*› **and**
confl: ‹*get-conflict-l-init S* = *None*›
shows
‹*init-dt-pre CS* (*propagate-unit-init-l L S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [[*L*]] *S* (*propagate-unit-init-l L S*)› (**is** *?spec*)
proof −
obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
dist: ‹*Ball* (*set CS*) *distinct*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC′-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal*+
by *presburger*
obtain *M N D NE UE Q U OC* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *Q*), *OC*)›
by (*cases S*) *auto*
have [*simp*]: ‹(*propagate-unit-init-l L S*, *propagate-unit-init L T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*) (*auto simp*: *twl-st-l-init-def propagate-unit-init-l-def*
convert-lit.simps convert-lits-l-extend-mono)
have *dec′*: ‹∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*›
using *SOC-T dec* **by** (*subst twl-st-l-init-no-decision-iff*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*propagate-unit-init L T*))›
apply (*rule twl-stgy-invs-backtrack-lvl-0*)
using *lev SOC-T*
by (*cases S*) (*auto simp*: *cdcl$_W$-restart-mset-state clauses-def twl-st-l-init-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*propagate-unit-init L T*)›
apply (*rule twl-struct-invs-init-propagate-unit-init*)

subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff uminus-lit-of-image-mset*)
subgoal
using *inv hd-C lev SOC-T dec′ confl in-literals-to-update WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff uminus-lit-of-image-mset*)
done
have [*simp*]: ‹*twl-list-invs* (*fst* (*propagate-unit-init-l L S*))›
using *add-inv*
by (*auto simp*: *S twl-list-invs-def propagate-unit-init-l-def*)
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*propagate-unit-init L T*›])
using *dist WS dec in-literals-to-update OC′-empty confl*
by (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*propagate-unit-init L T*›])
using *dist WS dec in-literals-to-update OC′-empty confl*
by (*auto simp*: *twl-st-init twl-st-l-init*)
qed

lemma [*twl-st-l-init*]:
‹*get-trail-l-init* (*set-conflict-init-l C S*) = *get-trail-l-init S*›
‹*literals-to-update-l-init* (*set-conflict-init-l C S*) = {#}›
‹*clauses-to-update-l-init* (*set-conflict-init-l C S*) = {#}›
‹*get-conflict-l-init* (*set-conflict-init-l C S*) = *Some* (*mset C*)›
‹*get-unit-clauses-l-init* (*set-conflict-init-l C S*) = *add-mset* (*mset C*) (*get-unit-clauses-l-init S*)›
‹*get-learned-unit-clauses-l-init* (*set-conflict-init-l C S*) = *get-learned-unit-clauses-l-init S*›
‹*get-clauses-l-init* (*set-conflict-init-l C S*) = *get-clauses-l-init S*›
‹*other-clauses-l-init* (*set-conflict-init-l C S*) = *other-clauses-l-init S*›
by (*cases S*; *auto simp*: *set-conflict-init-l-def*; *fail*)+

lemma *init-dt-pre-set-conflict-init-l*:
assumes
[*simp*]: ‹*get-conflict-l-init S* = *None*› **and**
pre: ‹*init-dt-pre* (*C* # *CS*) *S*› **and**
false: ‹$\forall$ *L* $\in$ *set C*. $-L$ $\in$ *lits-of-l* (*get-trail-l-init S*)› **and**
nempty: ‹*C* $\neq$ []›
shows
‹*init-dt-pre CS* (*set-conflict-init-l C S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [*C*] *S* (*set-conflict-init-l C S*)› (**is** *?spec*)
proof −
obtain *T* **where**

SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
dist: ‹*Ball* (*set CS*) *distinct*› **and**
dist-C: ‹*distinct C*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* '# *lit-of* '# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC'-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal*+
by *force*
obtain *M N D NE UE Q U OC* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *Q*), *OC*)›
by (*cases S*) *auto*
have [*simp*]: ‹*twl-list-invs* (*fst* (*set-conflict-init-l C S*))›
using *add-inv* **by** (*auto simp*: *set-conflict-init-l-def S*
twl-list-invs-def)
have [*simp*]: ‹(*set-conflict-init-l C S*, *set-conflict-init C T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*) (*auto simp*: *twl-st-l-init-def set-conflict-init-l-def convert-lit.simps*
convert-lits-l-extend-mono)
have *dec'*: ‹*count-decided* (*get-trail-init T*) = *0*›
apply (*subst count-decided-0-iff*)
apply (*subst twl-st-l-init-no-decision-iff*)
using *SOC-T dec SOC-T* **by** (*auto simp*: *twl-st-l-init twl-st-init convert-lits-l-def*)
have [*simp*]: ‹*twl-stgy-invs* (*fst* (*set-conflict-init C T*))›
using *stgy-inv dec' nempty count-decided-ge-get-level*[*of* ‹*get-trail-init T*›]
unfolding *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def* $cdcl_W$*-restart-mset.no-smaller-confl-def*
by (*cases T*; *cases C*)
(*auto 5 5 simp*: $cdcl_W$*-restart-mset-state clauses-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*set-conflict-init C T*)›
apply (*rule twl-struct-invs-init-set-conflict-init*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec' false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
subgoal
using *inv nempty dist-C SOC-T dec false nempty*
by (*auto simp*: *twl-st-init count-decided-0-iff*)
done
show *?pre*
unfolding *init-dt-pre-def*

apply (*rule exI*[*of* - ‹*set-conflict-init C T*›])
using *dist WS dec in-literals-to-update OC′-empty* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*set-conflict-init C T*›])
using *dist WS dec in-literals-to-update OC′-empty* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
qed

lemma [*twl-st-init*]:
‹*get-trail-init* (*add-empty-conflict-init T*) = *get-trail-init T*›
‹*get-conflict-init* (*add-empty-conflict-init T*) = *Some* {#}›
‹ *clauses-to-update-init* (*add-empty-conflict-init T*) = *clauses-to-update-init T*›
‹*literals-to-update-init* (*add-empty-conflict-init T*) = {#}›
by (*cases T*; *auto simp*:; *fail*)+

lemma [*twl-st-l-init*]:
‹*get-trail-l-init* (*add-empty-conflict-init-l T*) = *get-trail-l-init T*›
‹*get-conflict-l-init* (*add-empty-conflict-init-l T*) = *Some* {#}›
‹*clauses-to-update-l-init* (*add-empty-conflict-init-l T*) = *clauses-to-update-l-init T*›
‹*literals-to-update-l-init* (*add-empty-conflict-init-l T*) = {#}›
‹*get-unit-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-unit-clauses-l-init T*›
‹*get-learned-unit-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-learned-unit-clauses-l-init T*›
‹*get-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-clauses-l-init T*›
‹*other-clauses-l-init* (*add-empty-conflict-init-l T*) = *add-mset* {#} (*other-clauses-l-init T*)›
by (*cases T*; *auto simp*: *add-empty-conflict-init-l-def*; *fail*)+

lemma *twl-struct-invs-init-add-empty-conflict-init-l*:
assumes
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-init T* = {#}›
shows ‹*twl-struct-invs-init* (*add-empty-conflict-init T*)›
(**is** *?all-struct*)
proof −
obtain *M N U D NE UE Q OC* **where**
T: ‹*T* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *Q*), *OC*)›
using *WS* **by** (*cases T*) *auto*
have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, *add-mset* {#} (*clauses N* + *NE* + *OC*),
clauses U + *UE*, *Some* {#})›
unfolding *T twl-struct-invs-init-def*
by (*auto 5 5 simp*: $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset-state*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def all-decomposition-implies-def*
clauses-def $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*
true-annots-true-cls-def-iff-negation-in-model)

have ‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *clauses N* + *NE* + *OC*, *clauses U* + *UE*, *D*)›
using *invs* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:
‹$cdcl_W$*-restart-mset.no-smaller-propa* (*M*, *add-mset* {#} (*clauses N* + *NE* + *OC*),
clauses U + *UE*, *Some* {#})›
using *lev*

by (*auto simp*: *cdcl*$_W$*-restart-mset.no-smaller-propa-def cdcl*$_W$*-restart-mset-state clauses-def T count-decided-0-iff*)
let *?T* = ‹(*M*, *N*, *U*, *Some* {#}, *NE*, *UE*, {#}, {#})›

have [*simp*]: ‹*confl-cands-enqueued ?T*›
‹*propa-cands-enqueued ?T*›
‹*twl-st-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *Q*) ⟹ *twl-st-inv ?T*›
‹⋀*x*. *twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *Q*) *x* ⟹ *twl-exception-inv ?T x*›
‹*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *Q*) ⟹ *clauses-to-update-inv ?T*›
‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *Q*) ⟹ *past-invs ?T*›
by (*auto simp*: *twl-st-inv.simps twl-exception-inv.simps past-invs.simps*; *fail*)+
have [*simp*]: ‹*entailed-clss-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, *Q*) ⟹ *entailed-clss-inv ?T*›
using *count-decided-ge-get-level*[*of M*] *lev* **by** (*auto simp*: *T*)
show *?all-struct*
using *invs*
unfolding *twl-struct-invs-init-def T*
unfolding *fst-conv add-to-other-init.simps state*$_W$*-of-init.simps get-conflict.simps*
by (*clarsimp simp del*: *entailed-clss-inv.simps*)
qed

lemma *init-dt-pre-add-empty-conflict-init-l*:
assumes
confl[*simp*]: ‹*get-conflict-l-init S* = *None*› **and**
pre: ‹*init-dt-pre* ([] # *CS*) *S*›
shows
‹*init-dt-pre CS* (*add-empty-conflict-init-l S*)› (**is** *?pre*)
‹*init-dt-spec* [[]] *S* (*add-empty-conflict-init-l S*)› (**is** *?spec*)
proof −
obtain *T* **where**
SOC-T: ‹(*S*, *T*) ∈ *twl-st-l-init*› **and**
dist: ‹*Ball* (*set CS*) *distinct*› **and**
inv: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-l-init S* = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*› **and**
in-literals-to-update: ‹*get-conflict-l-init S* = *None* ⟶
literals-to-update-l-init S = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init S*)› **and**
add-inv: ‹*twl-list-invs* (*fst S*)› **and**
stgy-inv: ‹*twl-stgy-invs* (*fst T*)› **and**
OC′-empty: ‹*other-clauses-l-init S* ≠ {#} ⟶ *get-conflict-l-init S* ≠ *None*›
using *pre* **unfolding** *init-dt-pre-def*
apply −
apply *normalize-goal*+
by *force*
obtain *M N D NE UE Q U OC* **where**
S: ‹*S* = ((*M*, *N*, *U*, *D*, *NE*, *UE*, *Q*), *OC*)›
by (*cases S*) *auto*
have [*simp*]: ‹*twl-list-invs* (*fst* (*add-empty-conflict-init-l S*))›
using *add-inv* **by** (*auto simp*: *add-empty-conflict-init-l-def S twl-list-invs-def*)
have [*simp*]: ‹(*add-empty-conflict-init-l S*, *add-empty-conflict-init T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases S*) (*auto simp*: *twl-st-l-init-def add-empty-conflict-init-l-def*)
have *dec′*: ‹*count-decided* (*get-trail-init T*) = *0*›
apply (*subst count-decided-0-iff*)
apply (*subst twl-st-l-init-no-decision-iff*)
using *SOC-T dec SOC-T* **by** (*auto simp*: *twl-st-l-init twl-st-init convert-lits-l-def*)

have [*simp*]: ‹*twl-stgy-invs* (*fst* (*add-empty-conflict-init T*))›
using *stgy-inv dec′ count-decided-ge-get-level*[*of* ‹*get-trail-init T*›]
unfolding *twl-stgy-invs-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def* $cdcl_W$*-restart-mset.no-smaller-confl-def*
by (*cases T*)
(*auto 5 5 simp*: $cdcl_W$*-restart-mset-state clauses-def*)
note *clauses-to-update-inv.simps*[*simp del*] *valid-enqueued-alt-simps*[*simp del*]
have [*simp*]: ‹*twl-struct-invs-init* (*add-empty-conflict-init T*)›
apply (*rule twl-struct-invs-init-add-empty-conflict-init-l*)
using *inv SOC-T dec′ WS*
by (*auto simp*: *twl-st-init twl-st-l-init count-decided-0-iff*)
show *?pre*
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*add-empty-conflict-init T*›])
using *dist WS dec in-literals-to-update OC′-empty* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
show *?spec*
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*add-empty-conflict-init T*›])
using *dist WS dec in-literals-to-update OC′-empty* **by** (*auto simp*: *twl-st-init twl-st-l-init*)
qed

lemma [*twl-st-l-init*]:
‹*get-trail* (*fst* (*add-to-clauses-init a T*)) = *get-trail-init T*›
by (*cases T*; *auto*; *fail*)

lemma [*twl-st-l-init*]:
‹*other-clauses-l-init* (*T*, *OC*) = *OC*›
‹*clauses-to-update-l-init* (*T*, *OC*) = *clauses-to-update-l T*›
by (*cases T*; *auto*; *fail*)+

lemma *twl-struct-invs-init-add-to-clauses-init*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
confl: ‹*get-conflict-init T* = *None*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
dist-C: ‹*distinct C*› **and**
le-2: ‹*length C* ≥ *2*›
shows
‹*twl-struct-invs-init* (*add-to-clauses-init C T*)›
(**is** *?all-struct*)
proof −
obtain *M N U NE UE OC WS* **where**
T: ‹*T* = ((*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *uminus* ‘# *lit-of* ‘# *mset M*), *OC*)›
using *confl MQ* **by** (*cases T*) *auto*
let *?Q* = ‹*uminus* ‘# *lit-of* ‘# *mset M*›
have [*simp*]: ‹*get-all-ann-decomposition M* = [([], *M*)]›
by (*rule no-decision-get-all-ann-decomposition*) (*use lev* **in** ‹*auto simp*: *T count-decided-0-iff*›)
have ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* (*M*, (*clauses N* + *NE* + *OC*), *clauses U* + *UE*, *None*)›
and
excep: ‹*twl-st-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *?Q*)› **and**
st-inv: ‹*twl-st-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *?Q*)›
using *invs confl* **unfolding** *T twl-struct-invs-init-def* **by** *auto*
then have [*simp*]:

⟨$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, add-mset (mset C) (clauses N + NE + OC), clauses U + UE, None)⟩ **and**
n-d: ⟨no-dup M⟩
using dist-C
by (auto simp: $cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv-def $cdcl_W$-restart-mset.no-strange-atm-def $cdcl_W$-restart-mset-state $cdcl_W$-restart-mset.$cdcl_W$-M-level-inv-def $cdcl_W$-restart-mset.$cdcl_W$-conflicting-def $cdcl_W$-restart-mset.distinct-$cdcl_W$-state-def all-decomposition-implies-def clauses-def $cdcl_W$-restart-mset.$cdcl_W$-learned-clause-alt-def)
have ⟨$cdcl_W$-restart-mset.no-smaller-propa (M, clauses N + NE + OC, clauses U + UE, None)⟩
using invs confl **unfolding** T twl-struct-invs-init-def **by** auto
then have [simp]:
⟨$cdcl_W$-restart-mset.no-smaller-propa (M, add-mset (mset C) (clauses N + NE + OC), clauses U + UE, None)⟩
using lev
by (auto simp: $cdcl_W$-restart-mset.no-smaller-propa-def $cdcl_W$-restart-mset-state clauses-def T count-decided-0-iff)

let ?S = ⟨(M, N, U, None, NE, UE, WS, ?Q)⟩

have struct: ⟨struct-wf-twl-cls C⟩ **if** ⟨C ∈# N + U⟩ **for** C
using st-inv that **by** (simp add: twl-st-inv.simps)
have ⟨entailed-clss-inv (fst T)⟩
using invs **unfolding** T twl-struct-invs-init-def fst-conv **by** fast
then have ent: ⟨entailed-clss-inv (fst (add-to-clauses-init C T))⟩
using lev **by** (auto simp: T get-level-cons-if)
show ⟨twl-struct-invs-init (add-to-clauses-init C T)⟩
apply (rule twl-struct-invs-init-init-state)
subgoal using lev **by** (auto simp: T)
subgoal using struct dist-C le-2 **by** (auto simp: T mset-take-mset-drop-mset')
subgoal using MQ **by** (auto simp: T)
subgoal using WS **by** (auto simp: T)
subgoal by (simp add: T mset-take-mset-drop-mset')
subgoal by (auto simp: T mset-take-mset-drop-mset')
subgoal by (rule ent)
subgoal by (auto simp: T)
done
qed

lemma get-trail-init-add-to-clauses-init[simp]:
⟨get-trail-init (add-to-clauses-init a T) = get-trail-init T⟩
by (cases T) auto

lemma init-dt-pre-add-to-clauses-init-l:
assumes
D: ⟨get-conflict-l-init S = None⟩ **and**
a: ⟨length a ≠ Suc 0⟩ ⟨a ≠ []⟩ **and**
pre: ⟨init-dt-pre (a # CS) S⟩ **and**
⟨∀ s∈set (get-trail-l-init S). ¬ is-decided s⟩
shows
⟨add-to-clauses-init-l a S ≤ SPEC (init-dt-pre CS)⟩ (**is** ?pre) **and**
⟨add-to-clauses-init-l a S ≤ SPEC (init-dt-spec [a] S)⟩ (**is** ?spec)
proof −
obtain T **where**
SOC-T: ⟨(S, T) ∈ twl-st-l-init⟩ **and**
dist: ⟨Ball (set (a # CS)) distinct⟩ **and**

```
    inv: ‹twl-struct-invs-init T› and
    WS: ‹clauses-to-update-l-init S = {#}› and
    dec: ‹∀ s∈set (get-trail-l-init S). ¬ is-decided s› and
    in-literals-to-update: ‹get-conflict-l-init S = None ⟶
     literals-to-update-l-init S = uminus `# lit-of `# mset (get-trail-l-init S)› and
    add-inv: ‹twl-list-invs (fst S)› and
    stgy-inv: ‹twl-stgy-invs (fst T)› and
    OC'-empty: ‹other-clauses-l-init S ≠ {#} ⟶ get-conflict-l-init S ≠ None›
    using pre unfolding init-dt-pre-def
    apply −
    apply normalize-goal+
    by force
  have dec': ‹∀ L ∈ set (get-trail-init T). ¬is-decided L›
    using SOC-T dec apply −
    apply (rule twl-st-l-init-no-decision-iff[THEN iffD2])
    using SOC-T dec SOC-T by (auto simp: twl-st-l-init twl-st-init convert-lits-l-def)
  obtain M N NE UE Q OC where
    S: ‹S = ((M, N, None, NE, UE, {#}, Q), OC)›
    using D WS by (cases S) auto
  have le-2: ‹length a ≥ 2›
    using a by (cases a) auto
  have
    ‹init-dt-pre CS ((M, fmupd i (a, True) N, None, NE, UE, {#}, Q), OC)› (is ?pre1) and
    ‹init-dt-spec [a] S
         ((M, fmupd i (a, True) N, None, NE, UE, {#}, Q), OC)› (is ?spec1)
    if
      i-0: ‹0 < i› and
      i-dom: ‹i ∉# dom-m N›
    for i :: ‹nat›
  proof −
    let ?S = ‹((M, fmupd i (a, True) N, None, NE, UE, {#}, Q), OC)›

    have ‹Propagated L i ∉ set M› for L
      using add-inv i-dom i-0 unfolding S
      by (auto simp: twl-list-invs-def)
    then have ‹(?S, add-to-clauses-init a T) ∈ twl-st-l-init›
      using SOC-T i-dom
      by (auto simp: S twl-st-l-init-def init-clss-l-mapsto-upd-notin
          learned-clss-l-mapsto-upd-notin-irrelev convert-lit.simps
          intro!: convert-lits-l-extend-mono[of - - N ‹NE+UE› ‹fmupd i (a, True) N›])
    moreover have ‹twl-struct-invs-init (add-to-clauses-init a T)›
      apply (rule twl-struct-invs-init-add-to-clauses-init)
      subgoal
        apply (subst count-decided-0-iff)
        apply (subst twl-st-l-init-no-decision-iff)
        using SOC-T dec SOC-T by (auto simp: twl-st-l-init twl-st-init convert-lits-l-def)
      subgoal by (use dec SOC-T in-literals-to-update dist in
          ‹auto simp: S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv›)
      subgoal by (use dec SOC-T in-literals-to-update dist in
          ‹auto simp: S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv›)
      subgoal by (use dec SOC-T in-literals-to-update dist in
          ‹auto simp: S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv›)
      subgoal by (use dec SOC-T in-literals-to-update dist in
          ‹auto simp: S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv›)
      subgoal by (use dec SOC-T in-literals-to-update dist in
          ‹auto simp: S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv›)
```

subgoal by (*use dec SOC-T in-literals-to-update dist* **in**
⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init le-2 inv*⟩)
done
moreover have ⟨*twl-list-invs* (*M*, *fmupd i* (*a*, *True*) *N*, *None*, *NE*, *UE*, {#}, *Q*)⟩
using *add-inv i-dom i-0* **by** (*auto simp*: *S twl-list-invs-def*)
moreover have ⟨*twl-stgy-invs* (*fst* (*add-to-clauses-init a T*))⟩
by (*rule twl-stgy-invs-backtrack-lvl-0*)
(*use dec′ SOC-T* **in** ⟨*auto simp*: *S count-decided-0-iff twl-st-l-init twl-st-init*
twl-st-l-init-def⟩)
ultimately show *?pre1 ?spec1*
unfolding *init-dt-pre-def init-dt-spec-def* **apply** −
subgoal
apply (*rule exI*[*of* - ⟨*add-to-clauses-init a T*⟩])
using *dist dec OC′-empty in-literals-to-update* **by** (*auto simp*: *S*)
subgoal
apply (*rule exI*[*of* - ⟨*add-to-clauses-init a T*⟩])
using *dist dec OC′-empty in-literals-to-update i-dom i-0 a*
by (*auto simp*: *S learned-clss-l-mapsto-upd-notin-irrelev ran-m-mapsto-upd-notin*)
done
qed
then show *?pre ?spec*
by (*auto simp*: *S add-to-clauses-init-l-def get-fresh-index-def RES-RETURN-RES*)
qed

lemma *init-dt-pre-init-dt-step*:
assumes *pre*: ⟨*init-dt-pre* (*a* # *CS*) *SOC*⟩
shows ⟨*init-dt-step a SOC* ≤ *SPEC* (λ*SOC′*. *init-dt-pre CS SOC′* ∧ *init-dt-spec* [*a*] *SOC SOC′*)⟩
proof −
obtain *S OC* **where** *SOC*: ⟨*SOC* = (*S*, *OC*)⟩
by (*cases SOC*) *auto*
obtain *T* **where**
SOC-T: ⟨((*S*, *OC*), *T*) ∈ *twl-st-l-init*⟩ **and**
dist: ⟨*Ball* (*set* (*a* # *CS*)) *distinct*⟩ **and**
inv: ⟨*twl-struct-invs-init T*⟩ **and**
WS: ⟨*clauses-to-update-l-init* (*S*, *OC*) = {#}⟩ **and**
dec: ⟨∀ *s*∈*set* (*get-trail-l-init* (*S*, *OC*)). ¬ *is-decided s*⟩ **and**
in-literals-to-update: ⟨*get-conflict-l-init* (*S*, *OC*) = *None* ⟶
literals-to-update-l-init (*S*, *OC*) = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init* (*S*, *OC*))⟩ **and**
add-inv: ⟨*twl-list-invs* (*fst* (*S*, *OC*))⟩ **and**
stgy-inv: ⟨*twl-stgy-invs* (*fst T*)⟩ **and**
OC′-empty: ⟨*other-clauses-l-init* (*S*, *OC*) ≠ {#} ⟶ *get-conflict-l-init* (*S*, *OC*) ≠ *None*⟩
using *pre* **unfolding** *SOC init-dt-pre-def*
apply −
apply *normalize-goal*+
by *presburger*
have *dec′*: ⟨∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*⟩
using *SOC-T dec* **by** (*rule twl-st-l-init-no-decision-iff*[*THEN iffD2*])

obtain *M N D NE UE Q* **where**
S: ⟨*SOC* = ((*M*, *N*, *D*, *NE*, *UE*, {#}, *Q*), *OC*)⟩
using *WS* **by** (*cases SOC*) (*auto simp*: *SOC*)
then have *S′*: ⟨*S* = (*M*, *N*, *D*, *NE*, *UE*, {#}, *Q*)⟩
using *S* **unfolding** *SOC* **by** *auto*
show *?thesis*
proof (*cases* ⟨*get-conflict-l* (*fst SOC*)⟩)
case *None*

then show *?thesis*
using *pre dec* **by** (*auto simp add*: *Let-def count-decided-0-iff SOC twl-st-l-init twl-st-init*
true-annot-iff-decided-or-true-lit length-list-Suc-0
init-dt-step-def get-fresh-index-def RES-RETURN-RES
intro!: *init-dt-pre-already-propagated-unit-init-l init-dt-pre-set-conflict-init-l*
init-dt-pre-propagate-unit-init init-dt-pre-add-empty-conflict-init-l
init-dt-pre-add-to-clauses-init-l SPEC-rule-conjI
dest: *init-dt-pre-ConsD in-lits-of-l-defined-litD*)
next
case (*Some D′*)
then have [*simp*]: ‹*D* = *Some D′*›
by (*auto simp*: *S*)
have [*simp*]:
‹(((*M*, *N*, *Some D′*, *NE*, *UE*, {#}, *Q*), *add-mset* (*mset a*) *OC*), *add-to-other-init a T*)
∈ *twl-st-l-init*›
using *SOC-T* **by** (*cases T*; *auto simp*: *S S′ twl-st-l-init-def*; *fail*)+
have ‹*init-dt-pre CS* ((*M*, *N*, *Some D′*, *NE*, *UE*, {#}, *Q*), *add-mset* (*mset a*) *OC*)›
unfolding *init-dt-pre-def*
apply (*rule exI*[*of* - ‹*add-to-other-init a T*›])
using *dist inv WS dec′ dec in-literals-to-update add-inv stgy-inv SOC-T*
by (*auto simp*: *S′ count-decided-0-iff twl-st-init*
intro!: *twl-struct-invs-init-add-to-other-init*)
moreover have ‹*init-dt-spec* [*a*] ((*M*, *N*, *Some D′*, *NE*, *UE*, {#}, *Q*), *OC*)
((*M*, *N*, *Some D′*, *NE*, *UE*, {#}, *Q*), *add-mset* (*mset a*) *OC*)›
unfolding *init-dt-spec-def*
apply (*rule exI*[*of* - ‹*add-to-other-init a T*›])
using *dist inv WS dec dec′ in-literals-to-update add-inv stgy-inv SOC-T*
by (*auto simp*: *S′ count-decided-0-iff twl-st-init*
intro!: *twl-struct-invs-init-add-to-other-init*)
ultimately show *?thesis*
by (*auto simp*: *S init-dt-step-def*)
qed
qed

lemma [*twl-st-l-init*]:
‹*get-trail-l-init* (*S*, *OC*) = *get-trail-l S*›
‹*literals-to-update-l-init* (*S*, *OC*) = *literals-to-update-l S*›
by (*cases S*; *auto*; *fail*)+

lemma *init-dt-spec-append*:
assumes
spec1: ‹*init-dt-spec CS S T*› **and**
spec: ‹*init-dt-spec CS′ T U*›
shows ‹*init-dt-spec* (*CS* @ *CS′*) *S U*›
proof −
obtain *T′* **where**
TT′: ‹(*T*, *T′*) ∈ *twl-st-l-init*› **and**
‹*twl-struct-invs-init T′*› **and**
‹*clauses-to-update-l-init T* = {#}› **and**
‹∀ *s*∈*set* (*get-trail-l-init T*). ¬ *is-decided s*› **and**
‹*get-conflict-l-init T* = *None* ⟶
literals-to-update-l-init T = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-l-init T*)› **and**
clss: ‹*mset* ‘# *mset CS* + *mset* ‘# *ran-mf* (*get-clauses-l-init S*) + *other-clauses-l-init S* +
get-unit-clauses-l-init S =
mset ‘# *ran-mf* (*get-clauses-l-init T*) + *other-clauses-l-init T* + *get-unit-clauses-l-init T*› **and**
learned: ‹*learned-clss-lf* (*get-clauses-l-init S*) = *learned-clss-lf* (*get-clauses-l-init T*)› **and**

unit-le: ‹*get-learned-unit-clauses-l-init* T = *get-learned-unit-clauses-l-init* S› **and**
‹*twl-list-invs* (*fst* T)› **and**
‹*twl-stgy-invs* (*fst* T')› **and**
‹*other-clauses-l-init* $T \neq$ {#} ⟶ *get-conflict-l-init* $T \neq$ *None*› **and**
empty: ‹{#} ∈# *mset* '# *mset* *CS* ⟶ *get-conflict-l-init* $T \neq$ *None*› **and**
confl-kept: ‹*get-conflict-l-init* $S \neq$ *None* ⟶ *get-conflict-l-init* S = *get-conflict-l-init* T›
using *spec1*
unfolding *init-dt-spec-def* **apply** −
apply *normalize-goal+*
by *metis*

obtain U' **where**
UU': ‹(U, U') ∈ *twl-st-l-init*› **and**
struct-invs: ‹*twl-struct-invs-init* U'› **and**
WS: ‹*clauses-to-update-l-init* U = {#}› **and**
dec: ‹∀ *s*∈*set* (*get-trail-l-init* U). ¬ *is-decided s*› **and**
confl: ‹*get-conflict-l-init* U = *None* ⟶
literals-to-update-l-init U = *uminus* '# *lit-of* '# *mset* (*get-trail-l-init* U)› **and**
clss': ‹*mset* '# *mset* CS' + *mset* '# *ran-mf* (*get-clauses-l-init* T) + *other-clauses-l-init* T +
get-unit-clauses-l-init T =
mset '# *ran-mf* (*get-clauses-l-init* U) + *other-clauses-l-init* U + *get-unit-clauses-l-init* U› **and**
learned': ‹*learned-clss-lf* (*get-clauses-l-init* T) = *learned-clss-lf* (*get-clauses-l-init* U)› **and**
unit-le': ‹*get-learned-unit-clauses-l-init* U = *get-learned-unit-clauses-l-init* T› **and**
list-invs: ‹*twl-list-invs* (*fst* U)› **and**
stgy-invs: ‹*twl-stgy-invs* (*fst* U')› **and**
oth: ‹*other-clauses-l-init* $U \neq$ {#} ⟶ *get-conflict-l-init* $U \neq$ *None*› **and**
empty': ‹{#} ∈# *mset* '# *mset* CS' ⟶ *get-conflict-l-init* $U \neq$ *None*› **and**
confl-kept': ‹*get-conflict-l-init* $T \neq$ *None* ⟶ *get-conflict-l-init* T = *get-conflict-l-init* U›
using *spec*
unfolding *init-dt-spec-def* **apply** −
apply *normalize-goal+*
by *metis*

show *?thesis*
unfolding *init-dt-spec-def* **apply** −
apply (*rule exI*[*of - U'*])
apply (*intro conjI*)
subgoal using UU' **.**
subgoal using *struct-invs* **.**
subgoal using *WS* **.**
subgoal using *dec* **.**
subgoal using *confl* **.**
subgoal using *clss clss'*
by (*smt ab-semigroup-add-class.add.commute ab-semigroup-add-class.add.left-commute image-mset-union mset-append*)
subgoal using *learned' learned* **by** *simp*
subgoal using *unit-le unit-le'* **by** *simp*
subgoal using *list-invs* **.**
subgoal using *stgy-invs* **.**
subgoal using *oth* **.**
subgoal using *empty empty' oth confl-kept'* **by** *auto*
subgoal using *confl-kept confl-kept'* **by** *auto*
done
qed

lemma *init-dt-full*:

fixes CS :: ⟨$'v$ *literal list list*⟩ **and** SOC :: ⟨$'v$ *twl-st-l-init*⟩ **and** S'
defines
⟨$S \equiv$ *fst* SOC⟩ **and**
⟨$OC \equiv$ *snd* SOC⟩
assumes
⟨*init-dt-pre* CS SOC⟩
shows
⟨*init-dt* CS $SOC \leq$ *SPEC* (*init-dt-spec* CS SOC)⟩
using *assms* **unfolding** *S-def OC-def*
proof (*induction CS arbitrary*: *SOC*)
case *Nil*
then obtain S OC **where** *SOC*: ⟨$SOC = (S, OC)$⟩
by (*cases SOC*) *auto*
from *Nil*
obtain T **where**
T: ⟨$(SOC, T) \in$ *twl-st-l-init*⟩
⟨*Ball* (*set* []) *distinct*⟩
⟨*twl-struct-invs-init* T⟩
⟨*clauses-to-update-l-init* $SOC = \{\#\}$⟩
⟨$\forall s \in$ *set* (*get-trail-l-init* SOC). $\neg$ *is-decided* s⟩
⟨*get-conflict-l-init* $SOC =$ *None* $\longrightarrow$
literals-to-update-l-init $SOC =$
uminus ‘# *lit-of* ‘# *mset* (*get-trail-l-init* SOC)⟩
⟨*twl-list-invs* (*fst* SOC)⟩
⟨*twl-stgy-invs* (*fst* T)⟩
⟨*other-clauses-l-init* $SOC \neq \{\#\} \longrightarrow$ *get-conflict-l-init* $SOC \neq$ *None*⟩
unfolding *init-dt-pre-def* **apply** −
apply *normalize-goal+*
by *auto*

then show *?case*
unfolding *init-dt-def SOC init-dt-spec-def nfoldli-simps*
apply (*intro RETURN-rule*)
unfolding *prod.simps*
apply (*rule exI*[*of - T*])
using T **by** (*auto simp*: *SOC twl-st-init twl-st-l-init*)
next
case (*Cons a CS*) **note** $IH = this(1)$ **and** $pre = this(2)$
note *init-dt-step-def*[*simp*]
have *1*: ⟨*init-dt-step* a $SOC \leq$ *SPEC* ($\lambda SOC'$. *init-dt-pre* CS $SOC' \wedge$ *init-dt-spec* [a] SOC SOC')⟩
by (*rule init-dt-pre-init-dt-step*[*OF pre*])
have *2*: ⟨*init-dt-spec* (a # CS) SOC UOC⟩
if *spec*: ⟨*init-dt-spec* CS T UOC⟩ **and**
spec': ⟨*init-dt-spec* [a] SOC T⟩ **for** T UOC
using *init-dt-spec-append*[*OF spec' spec*] **by** *simp*
show *?case*
unfolding *init-dt-def nfoldli-simps if-True*
apply (*rule specify-left*)
apply (*rule 1*)
apply (*rule order.trans*)
unfolding *init-dt-def*[*symmetric*]
apply (*rule IH*)
apply (*solves* ⟨*simp*⟩)
apply (*rule SPEC-rule*)
by (*rule 2*) *fast+*
qed

lemma *init-dt-pre-empty-state*:
⟨*init-dt-pre* [] (([], *fmempty*, *None*, {#}, {#}, {#}, {#}), {#})⟩
unfolding *init-dt-pre-def*
by (*auto simp*: *twl-st-l-init-def twl-struct-invs-init-def twl-st-inv.simps*
twl-struct-invs-def twl-st-inv.simps $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def* $cdcl_W$*-restart-mset.no-smaller-propa-def*
past-invs.simps clauses-def
$cdcl_W$*-restart-mset-state twl-list-invs-def*
twl-stgy-invs-def $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.no-smaller-confl-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)

lemma *twl-init-invs*:
⟨*twl-struct-invs-init* (([], {#}, {#}, *None*, {#}, {#}, {#}, {#}), {#})⟩
⟨*twl-list-invs* ([], *fmempty*, *None*, {#}, {#}, {#}, {#})⟩
⟨*twl-stgy-invs* ([], {#}, {#}, *None*, {#}, {#}, {#}, {#})⟩
by (*auto simp*: *twl-struct-invs-init-def twl-st-inv.simps twl-list-invs-def twl-stgy-invs-def*
past-invs.simps
twl-struct-invs-def twl-st-inv.simps $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.no-strange-atm-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
$cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def* $cdcl_W$*-restart-mset.*$cdcl_W$*-conflicting-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def* $cdcl_W$*-restart-mset.no-smaller-propa-def*
past-invs.simps clauses-def
$cdcl_W$*-restart-mset-state twl-list-invs-def*
twl-stgy-invs-def $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant-def*
$cdcl_W$*-restart-mset.no-smaller-confl-def*
$cdcl_W$*-restart-mset.conflict-non-zero-unless-level-0-def*)
end
theory *Watched-Literals-Watch-List-Initialisation*
imports *Watched-Literals-Watch-List Watched-Literals-Initialisation*
begin

1.4.7 Initialisation

type-synonym *'v twl-st-wl-init'* = ⟨((*'v*, *nat*) *ann-lits* × *'v clauses-l* ×
'v cconflict × *'v clauses* × *'v clauses* × *'v lit-queue-wl*)⟩

type-synonym *'v twl-st-wl-init* = ⟨*'v twl-st-wl-init'* × *'v clauses*⟩
type-synonym *'v twl-st-wl-init-full* = ⟨*'v twl-st-wl* × *'v clauses*⟩

fun *get-trail-init-wl* :: ⟨*'v twl-st-wl-init* ⇒ (*'v*, *nat*) *ann-lit list*⟩ **where**
⟨*get-trail-init-wl* ((*M*, -, -, -, -, -), -) = *M*⟩

fun *get-clauses-init-wl* :: ⟨*'v twl-st-wl-init* ⇒ *'v clauses-l*⟩ **where**
⟨*get-clauses-init-wl* ((-, *N*, -, -, -, -), *OC*) = *N*⟩

fun *get-conflict-init-wl* :: ⟨*'v twl-st-wl-init* ⇒ *'v cconflict*⟩ **where**
⟨*get-conflict-init-wl* ((-, -, *D*, -, -, -), -) = *D*⟩

fun *literals-to-update-init-wl* :: ⟨*'v twl-st-wl-init* ⇒ *'v clause*⟩ **where**
⟨*literals-to-update-init-wl* ((-, -, -, -, -, *Q*), -) = *Q*⟩

fun *other-clauses-init-wl* :: ⟨*'v twl-st-wl-init* ⇒ *'v clauses*⟩ **where**

⟨*other-clauses-init-wl* ((-, -, -, -, -, -), *OC*) = *OC*⟩

fun *add-empty-conflict-init-wl* :: ⟨′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init*⟩ **where**
add-empty-conflict-init-wl-def[*simp del*]:
⟨*add-empty-conflict-init-wl* ((*M*, *N*, *D*, *NE*, *UE*, *Q*), *OC*) =
((*M*, *N*, *Some* {#}, *NE*, *UE*, {#}), *add-mset* {#} *OC*)⟩

fun *propagate-unit-init-wl* :: ⟨′*v literal* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init*⟩ **where**
propagate-unit-init-wl-def[*simp del*]:
⟨*propagate-unit-init-wl L* ((*M*, *N*, *D*, *NE*, *UE*, *Q*), *OC*) =
((*Propagated L 0* # *M*, *N*, *D*, *add-mset* {#*L*#} *NE*, *UE*, *add-mset* (−*L*) *Q*), *OC*)⟩

fun *already-propagated-unit-init-wl* :: ⟨′*v clause* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init*⟩ **where**
already-propagated-unit-init-wl-def[*simp del*]:
⟨*already-propagated-unit-init-wl C* ((*M*, *N*, *D*, *NE*, *UE*, *Q*), *OC*) =
((*M*, *N*, *D*, *add-mset C NE*, *UE*, *Q*), *OC*)⟩

fun *set-conflict-init-wl* :: ⟨′*v literal* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init*⟩ **where**
set-conflict-init-wl-def[*simp del*]:
⟨*set-conflict-init-wl L* ((*M*, *N*, -, *NE*, *UE*, *Q*), *OC*) =
((*M*, *N*, *Some* {#*L*#}, *add-mset* {#*L*#} *NE*, *UE*, {#}), *OC*)⟩

fun *add-to-clauses-init-wl* :: ⟨′*v clause-l* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init nres*⟩ **where**
add-to-clauses-init-wl-def[*simp del*]:
⟨*add-to-clauses-init-wl C* ((*M*, *N*, *D*, *NE*, *UE*, *Q*), *OC*) = *do* {
i ← *get-fresh-index N*;
let b = (*length C* = *2*);
RETURN ((*M*, *fmupd i* (*C*, *True*) *N*, *D*, *NE*, *UE*, *Q*), *OC*)
}⟩

definition *init-dt-step-wl* :: ⟨′*v clause-l* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init nres*⟩ **where**
⟨*init-dt-step-wl C S* =
(*case get-conflict-init-wl S of*
None ⇒
if length C = *0*
then RETURN (*add-empty-conflict-init-wl S*)
else if length C = *1*
then
let L = *hd C in*
if undefined-lit (*get-trail-init-wl S*) *L*
then RETURN (*propagate-unit-init-wl L S*)
else if L ∈ *lits-of-l* (*get-trail-init-wl S*)
then RETURN (*already-propagated-unit-init-wl* (*mset C*) *S*)
else RETURN (*set-conflict-init-wl L S*)
else
add-to-clauses-init-wl C S
| *Some D* ⇒
RETURN (*add-to-other-init C S*))⟩

fun *st-l-of-wl-init* :: ⟨′*v twl-st-wl-init*′ ⇒ ′*v twl-st-l*⟩ **where**
⟨*st-l-of-wl-init* (*M*, *N*, *D*, *NE*, *UE*, *Q*) = (*M*, *N*, *D*, *NE*, *UE*, {#}, *Q*)⟩

definition *state-wl-l-init′* **where**
‹*state-wl-l-init′* = {(*S* ,*S′*). *S′* = *st-l-of-wl-init S*}›

definition *init-dt-wl* :: ‹′*v clause-l list* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init nres*› **where**
‹*init-dt-wl CS* = *nfoldli CS* (λ-. *True*) *init-dt-step-wl*›

definition *state-wl-l-init* :: ‹(′*v twl-st-wl-init* × ′*v twl-st-l-init*) *set*› **where**
‹*state-wl-l-init* = {(*S*, *S′*). (*fst S*, *fst S′*) ∈ *state-wl-l-init′* ∧
other-clauses-init-wl S = *other-clauses-l-init S′*}›

fun *all-blits-are-in-problem-init* **where**
[*simp del*]: ‹*all-blits-are-in-problem-init* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
(∀ *L*. (∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *K* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))))›

We assume that no clause has been deleted during initialisation. The definition is slightly redundant since *i* ∈# *dom-m N* is already entailed by *fst* ‘# *mset* (*W L*) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}).

named-theorems *twl-st-wl-init*

lemma [*twl-st-wl-init*]:
assumes ‹(*S*, *S′*) ∈ *state-wl-l-init*›
shows
‹*get-conflict-l-init S′* = *get-conflict-init-wl S*›
‹*get-trail-l-init S′* = *get-trail-init-wl S*›
‹*other-clauses-l-init S′* = *other-clauses-init-wl S*›
‹*count-decided* (*get-trail-l-init S′*) = *count-decided* (*get-trail-init-wl S*)›
using *assms*
by (*solves* ‹*cases S*; *cases S′*; *auto simp*: *state-wl-l-init-def state-wl-l-def*
state-wl-l-init′-def›)+

lemma *in-clause-to-update-in-dom-mD*:
‹*bb* ∈# *clause-to-update L* (*a*, *aa*, *ab*, *ac*, *ad*, {#}, {#}) ⟹ *bb* ∈# *dom-m aa*›
unfolding *clause-to-update-def*
by *force*

lemma *init-dt-step-wl-init-dt-step*:
assumes *S-S′*: ‹(*S*, *S′*) ∈ *state-wl-l-init*› **and**
dist: ‹*distinct C*›
shows ‹*init-dt-step-wl C S* ≤ ⇓ *state-wl-l-init*
(*init-dt-step C S′*)›
(**is** ‹- ≤ ⇓ *?A* -›)
proof −
have *confl*: ‹(*get-conflict-init-wl S*, *get-conflict-l-init S′*) ∈ ⟨*Id*⟩*option-rel*›
using *S-S′* **by** (*auto simp*: *twl-st-wl-init*)
have *false*: ‹(*add-empty-conflict-init-wl S*, *add-empty-conflict-init-l S′*) ∈ *?A*›
using *S-S′*
apply (*cases S*; *cases S′*)
apply (*auto simp*: *add-empty-conflict-init-wl-def add-empty-conflict-init-l-def*
all-blits-are-in-problem-init.simps state-wl-l-init′-def
state-wl-l-init-def state-wl-l-def correct-watching.simps clause-to-update-def)
done
have *propa-unit*:
‹(*propagate-unit-init-wl* (*hd C*) *S*, *propagate-unit-init-l* (*hd C*) *S′*) ∈ *?A*›
using *S-S′* **apply** (*cases S*; *cases S′*)

apply (*auto simp*: *propagate-unit-init-l-def propagate-unit-init-wl-def state-wl-l-init'-def state-wl-l-init-def state-wl-l-def clause-to-update-def all-lits-of-mm-add-mset all-lits-of-m-add-mset all-lits-of-mm-union*)
done
have *already-propa*:
‹(*already-propagated-unit-init-wl* (*mset C*) *S*, *already-propagated-unit-init-l* (*mset C*) *S'*) ∈ *?A*›
using *S-S'*
by (*cases S*; *cases S'*)
(*auto simp*: *already-propagated-unit-init-wl-def already-propagated-unit-init-l-def state-wl-l-init-def state-wl-l-def clause-to-update-def all-lits-of-mm-add-mset all-lits-of-m-add-mset state-wl-l-init'-def*)
have *set-conflict*: ‹(*set-conflict-init-wl* (*hd C*) *S*, *set-conflict-init-l C S'*) ∈ *?A*›
if ‹*C* = [*hd C*]›
using *S-S' that*
by (*cases S*; *cases S'*)
(*auto simp*: *set-conflict-init-wl-def set-conflict-init-l-def state-wl-l-init-def state-wl-l-def clause-to-update-def state-wl-l-init'-def all-lits-of-mm-add-mset all-lits-of-m-add-mset*)
have *add-to-clauses-init-wl*: ‹*add-to-clauses-init-wl C S* ≤ ⇓ *state-wl-l-init* (*add-to-clauses-init-l C S'*)›
if *C*: ‹*length C* ≥ *2*› **and** *conf*: ‹*get-conflict-l-init S'* = *None*›
proof −
have [*iff*]: ‹*C* ! *Suc 0* ∉ *set* (*watched-l C*) ⟷ *False*›
‹*C* ! *0* ∉ *set* (*watched-l C*) ⟷ *False*› **and**
[*dest*!]: ‹⋀*L*. *L* ≠ *C* ! *0* ⟹ *L* ≠ *C* ! *Suc 0* ⟹ *L* ∈ *set* (*watched-l C*) ⟹ *False*›
using *C* **by** (*cases C*; *cases* ‹*tl C*›; *auto*)+
have [*dest*!]: ‹*C* ! *0* = *C* ! *Suc 0* ⟹ *False*›
using *C dist* **by** (*cases C*; *cases* ‹*tl C*›; *auto*)+
show *?thesis*
using *S-S' conf C*
by (*cases S*; *cases S'*)
(*auto 5 5 simp*: *add-to-clauses-init-wl-def add-to-clauses-init-l-def get-fresh-index-def state-wl-l-init-def state-wl-l-def clause-to-update-def all-lits-of-mm-add-mset all-lits-of-m-add-mset state-wl-l-init'-def RES-RETURN-RES Let-def intro*!: *RES-refine filter-mset-cong2*)
qed
have *add-to-other-init*:
‹(*add-to-other-init C S*, *add-to-other-init C S'*) ∈ *?A*›
using *S-S'*
by (*cases S*; *cases S'*)
(*auto simp*: *state-wl-l-init-def state-wl-l-def clause-to-update-def all-lits-of-mm-add-mset all-lits-of-m-add-mset state-wl-l-init'-def*)
show *?thesis*
unfolding *init-dt-step-wl-def init-dt-step-def*
apply (*refine-vcg confl false propa-unit already-propa set-conflict add-to-clauses-init-wl add-to-other-init*)
subgoal by *simp*
subgoal by *simp*
subgoal using *S-S'* **by** (*simp add*: *twl-st-wl-init*)
subgoal using *S-S'* **by** (*simp add*: *twl-st-wl-init*)
subgoal using *S-S'* **by** (*cases C*) *simp-all*
subgoal by *linarith*
done
qed

lemma *init-dt-wl-init-dt*:
  assumes *S-S′*: ‹$(S, S') \in$ *state-wl-l-init*› **and**
    dist: ‹$\forall$ *C*$\in$*set C*. *distinct C*›
  shows ‹*init-dt-wl C S* $\leq \Downarrow$ *state-wl-l-init*
        (*init-dt C S′*)›
proof −
  have *C*: ‹$(C, C) \in$ ⟨{(C, C'). $(C, C') \in$ *Id* $\wedge$ *distinct C*}⟩*list-rel*›
    using *dist*
    by (*auto simp*: *list-rel-def list.rel-refl-strong*)
  show *?thesis*
    unfolding *init-dt-wl-def init-dt-def*
    apply (*refine-vcg C S-S′*)
    subgoal using *S-S′* **by** *fast*
    subgoal by (*auto intro*!: *init-dt-step-wl-init-dt-step*)
    done
qed

definition *init-dt-wl-pre* **where**
  ‹*init-dt-wl-pre C S* $\longleftrightarrow$
    ($\exists$ *S′*. $(S, S') \in$ *state-wl-l-init* $\wedge$
      init-dt-pre C S′)›

definition *init-dt-wl-spec* **where**
  ‹*init-dt-wl-spec C S T* $\longleftrightarrow$
    ($\exists$ *S′ T′*. $(S, S') \in$ *state-wl-l-init* $\wedge$ $(T, T') \in$ *state-wl-l-init* $\wedge$
      init-dt-spec C S′ T′)›

lemma *init-dt-wl-init-dt-wl-spec*:
  assumes ‹*init-dt-wl-pre CS S*›
  shows ‹*init-dt-wl CS S* $\leq$ *SPEC* (*init-dt-wl-spec CS S*)›
proof −
  obtain *S′* **where**
     SS′: ‹$(S, S') \in$ *state-wl-l-init*› **and**
     pre: ‹*init-dt-pre CS S′*›
    using *assms* **unfolding** *init-dt-wl-pre-def* **by** *blast*
  have *dist*: ‹$\forall$ *C*$\in$*set CS*. *distinct C*›
    using *pre* **unfolding** *init-dt-pre-def* **by** *blast*
  show *?thesis*
    apply (*rule order.trans*)
     apply (*rule init-dt-wl-init-dt*[*OF SS′ dist*])
    apply (*rule order.trans*)
     apply (*rule ref-two-step′*)
     apply (*rule init-dt-full*[*OF pre*])
    apply (*unfold conc-fun-SPEC*)
    apply (*rule SPEC-rule*)
    apply *normalize-goal*+
    using *SS′ pre* **unfolding** *init-dt-wl-spec-def*
    by *blast*
qed

fun *correct-watching-init* :: ‹*′v twl-st-wl* $\Rightarrow$ *bool*› **where**
  [*simp del*]: ‹*correct-watching-init* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) $\longleftrightarrow$
    all-blits-are-in-problem-init (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) $\wedge$

```
  (∀ L.
     distinct-watched (W L) ∧
     (∀ (i, K, b)∈#mset (W L). i ∈# dom-m N ∧ K ∈ set (N ∝ i) ∧ K ≠ L ∧
        correctly-marked-as-binary N (i, K, b)) ∧
     fst `# mset (W L) = clause-to-update L (M, N, D, NE, UE, {#}, {#}))›

lemma correct-watching-init-correct-watching:
  ‹correct-watching-init T ⟹ correct-watching T›
  by (cases T)
    (fastforce simp: correct-watching.simps correct-watching-init.simps filter-mset-eq-conv
      all-blits-are-in-problem-init.simps
      in-clause-to-update-in-dom-mD)

lemma image-mset-Suc: ‹Suc `# {#C ∈# M. P C#} = {#C ∈# Suc `# M. P (C−1)#}›
  by (induction M) auto

lemma correct-watching-init-add-unit:
  assumes ‹correct-watching-init (M, N, D, NE, UE, Q, W)›
  shows ‹correct-watching-init (M, N, D, add-mset C NE, UE, Q, W)›
proof −
  have [intro!]: ‹(a, x) ∈ set (W L) ⟹ a ∈# dom-m N ⟹ b ∈ set (N ∝a)  ⟹
       b ∉# all-lits-of-mm {#mset (fst x). x ∈# ran-m N#} ⟹ b ∈# all-lits-of-mm NE›
    for x b F a L
    unfolding ran-m-def
    by (auto dest!: multi-member-split simp: all-lits-of-mm-add-mset in-clause-in-all-lits-of-m)

  show ?thesis
    using assms
    unfolding correct-watching-init.simps clause-to-update-def Ball-def
    by (fastforce simp: correct-watching.simps all-lits-of-mm-add-mset
        all-lits-of-m-add-mset Ball-def all-conj-distrib clause-to-update-def
        all-blits-are-in-problem-init.simps all-lits-of-mm-union
        dest!: )
qed

lemma correct-watching-init-propagate:
  ‹correct-watching-init ((L # M, N, D, NE, UE, Q, W)) ⟷
        correct-watching-init ((M, N, D, NE, UE, Q, W))›
  ‹correct-watching-init ((M, N, D, NE, UE, add-mset C Q, W)) ⟷
        correct-watching-init ((M, N, D, NE, UE, Q, W))›
  unfolding correct-watching-init.simps clause-to-update-def Ball-def
  by (auto simp: correct-watching.simps all-lits-of-mm-add-mset
      all-lits-of-m-add-mset Ball-def all-conj-distrib clause-to-update-def
      all-blits-are-in-problem-init.simps)

lemma all-blits-are-in-problem-cons[simp]:
  ‹all-blits-are-in-problem-init (Propagated L i # a, aa, ab, ac, ad, ae, b) ⟷
     all-blits-are-in-problem-init (a, aa, ab, ac, ad, ae, b)›
  ‹all-blits-are-in-problem-init (Decided L # a, aa, ab, ac, ad, ae, b) ⟷
     all-blits-are-in-problem-init (a, aa, ab, ac, ad, ae, b)›
  ‹all-blits-are-in-problem-init (a, aa, ab, ac, ad, add-mset L ae, b) ⟷
     all-blits-are-in-problem-init (a, aa, ab, ac, ad, ae, b)›
  ‹NO-MATCH None y ⟹ all-blits-are-in-problem-init (a, aa, y, ac, ad, ae, b) ⟷
     all-blits-are-in-problem-init (a, aa, None, ac, ad, ae, b)›
  ‹NO-MATCH {#} ae ⟹ all-blits-are-in-problem-init (a, aa, y, ac, ad, ae, b) ⟷
     all-blits-are-in-problem-init (a, aa, y, ac, ad, {#}, b)›
```

by (*auto simp*: *all-blits-are-in-problem-init.simps*)

lemma *correct-watching-init-cons*[*simp*]:
‹*NO-MATCH None y* $\Longrightarrow$ *correct-watching-init* ((*a*, *aa*, *y*, *ac*, *ad*, *ae*, *b*)) $\longleftrightarrow$
correct-watching-init ((*a*, *aa*, *None*, *ac*, *ad*, *ae*, *b*))›
‹*NO-MATCH* {#} *ae* $\Longrightarrow$ *correct-watching-init* ((*a*, *aa*, *y*, *ac*, *ad*, *ae*, *b*)) $\longleftrightarrow$
correct-watching-init ((*a*, *aa*, *y*, *ac*, *ad*, {#}, *b*))›
apply (*auto simp*: *correct-watching-init.simps clause-to-update-def*)
apply (*subst* (*asm*) *all-blits-are-in-problem-cons*(*4*))
apply *auto*
apply (*subst all-blits-are-in-problem-cons*(*4*))
apply *auto*
apply (*subst* (*asm*) *all-blits-are-in-problem-cons*(*5*))
apply *auto*
apply (*subst all-blits-are-in-problem-cons*(*5*))
apply *auto*
done

lemma *clause-to-update-mapsto-upd-notin*:
assumes
i: ‹$i \notin\#$ *dom-m N*›
shows
‹*clause-to-update L* (*M*, *N*($i \hookrightarrow C'$), *C*, *NE*, *UE*, *WS*, *Q*) =
(*if L* $\in$ *set* (*watched-l C'*)
then add-mset i (*clause-to-update L* (*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*))
else (*clause-to-update L* (*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*)))›
‹*clause-to-update L* (*M*, *fmupd i* (*C'*, *b*) *N*, *C*, *NE*, *UE*, *WS*, *Q*) =
(*if L* $\in$ *set* (*watched-l C'*)
then add-mset i (*clause-to-update L* (*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*))
else (*clause-to-update L* (*M*, *N*, *C*, *NE*, *UE*, *WS*, *Q*)))›
using *assms*
by (*auto simp*: *clause-to-update-def intro*!: *filter-mset-cong*)

lemma *correct-watching-init-add-clause*:
assumes
corr: ‹*correct-watching-init* ((*a*, *aa*, *None*, *ac*, *ad*, *Q*, *b*))› **and**
leC: ‹$2 \leq$ *length C*› **and**
i-notin[*simp*]: ‹$i \notin\#$ *dom-m aa*› **and**
dist[*iff*]: ‹*C* ! *0* $\neq$ *C* ! *Suc 0*›
shows ‹*correct-watching-init*
((*a*, *fmupd i* (*C*, *red*) *aa*, *None*, *ac*, *ad*, *Q*, *b*
(*C* ! *0* := *b* (*C* ! *0*) @ [(*i*, *C* ! *Suc 0*, *length C* = *2*)],
C ! *Suc 0* := *b* (*C* ! *Suc 0*) @ [(*i*, *C* ! *0*, *length C* = *2*)]))))›
proof −
have [*iff*]: ‹*C* ! *Suc 0* $\neq$ *C* ! *0*›
using ‹*C* ! *0* $\neq$ *C* ! *Suc 0*› **by** *argo*
have [*iff*]: ‹*C* ! *Suc 0* $\in\#$ *all-lits-of-m* (*mset C*)› ‹*C* ! *0* $\in\#$ *all-lits-of-m* (*mset C*)›
‹*C* ! *Suc 0* $\in$ *set C*› ‹ *C* ! *0* $\in$ *set C*› ‹*C* ! *0* $\in$ *set* (*watched-l C*)› ‹*C* ! *Suc 0* $\in$ *set* (*watched-l C*)›
using *leC* **by** (*force intro*!: *in-clause-in-all-lits-of-m nth-mem simp*: *in-set-conv-iff*
intro: *exI*[*of - 0*] *exI*[*of -* ‹*Suc 0*›])+
have [*dest*!]: ‹$\bigwedge$*L*. *L* $\neq$ *C* ! *0* $\Longrightarrow$ *L* $\neq$ *C* ! *Suc 0* $\Longrightarrow$ *L* $\in$ *set* (*watched-l C*) $\Longrightarrow$ *False*›
by (*cases C*; *cases* ‹*tl C*›; *auto*)+
have *i*: ‹$i \notin$ *fst* ‘ *set* (*b L*)› **for** *L*
using *corr i-notin* **unfolding** *correct-watching-init.simps*
by *force*

have [*iff*]: ‹$(i,c,\ d) \notin set\ (b\ L)$› **for** *L c d*
  using *i*[*of L*] **by** (*auto simp*: *image-iff*)
then show *?thesis*
  using *corr*
 by (*force simp*: *correct-watching-init.simps all-blits-are-in-problem-init.simps ran-m-mapsto-upd-notin all-lits-of-mm-add-mset all-lits-of-mm-union clause-to-update-mapsto-upd-notin correctly-marked-as-binary.simps*
   split: *if-splits*)
qed

definition *rewatch*
 :: ‹$'v$ *clauses-l* ⇒ ($'v$ *literal* ⇒ $'v$ *watched*) ⇒ ($'v$ *literal* ⇒ $'v$ *watched*) *nres*›
where
‹*rewatch N W* = *do* {
 xs ← *SPEC*(λ*xs*. *set-mset* (*dom-m N*) ⊆ *set xs* ∧ *distinct xs*);
 nfoldli
  xs
  (λ-. *True*)
  (λ*i W*. *do* {
   if i ∈# *dom-m N*
   then do {
    ASSERT(*i* ∈# *dom-m N*);
    ASSERT(*length* (*N* ∝ *i*) ≥ *2*);
    let L1 = *N* ∝ *i* ! *0*;
    let L2 = *N* ∝ *i* ! *1*;
    let b = (*length* (*N* ∝ *i*) = *2*);
    ASSERT(*L1* ≠ *L2*);
    ASSERT(*length* (*W L1*) < *size* (*dom-m N*));
    let W = *W*(*L1* := *W L1* @ [(*i*, *L2*, *b*)]);
    ASSERT(*length* (*W L2*) < *size* (*dom-m N*));
    let W = *W*(*L2* := *W L2* @ [(*i*, *L1*, *b*)]);
    RETURN W
   }
   else RETURN W
  })
  W
 }›

lemma *rewatch-correctness*:
 assumes [*simp*]: ‹*W* = (λ-. [])› **and**
  H[*dest*]: ‹⋀*x*. *x* ∈# *dom-m N* ⟹ *distinct* (*N* ∝ *x*) ∧ *length* (*N* ∝ *x*) ≥ *2*›
 shows
  ‹*rewatch N W* ≤ *SPEC*(λ*W*. *correct-watching-init* (*M*, *N*, *C*, *NE*, *UE*, *Q*, *W*))›
proof −
 define *I* **where**
  ‹*I* ≡ λ(*a* :: *nat list*) (*b* :: *nat list*) *W*.
    correct-watching-init ((*M*, *fmrestrict-set* (*set a*) *N*, *C*, *NE*, *UE*, *Q*, *W*))›
 have *I0*: ‹*set-mset* (*dom-m N*) ⊆ *set x* ∧ *distinct x* ⟹ *I* [] *x W*› **for** *x*
  unfolding *I-def* **by** (*auto simp*: *correct-watching-init.simps*
   all-blits-are-in-problem-init.simps clause-to-update-def)
 have *le*: ‹*length* (σ *L*) < *size* (*dom-m N*)›
   if ‹*correct-watching-init* (*M*, *fmrestrict-set* (*set l1*) *N*, *C*, *NE*, *UE*, *Q*, σ)› **and**
   ‹*set-mset* (*dom-m N*) ⊆ *set x* ∧ *distinct x*› **and**
   ‹*x* = *l1* @ *xa* # *l2*› ‹*xa* ∈# *dom-m N*›
   for *L l1* σ *xa l2 x*
 proof −
  have *1*: ‹*card* (*set l1*) ≤ *length l1*›

by (*auto simp*: *card-length*)
have ⟨*distinct-watched* (σ *L*)⟩ **and** ⟨*fst* ‘ *set* (σ *L*) ⊆ *set l1* ∩ *set-mset* (*dom-m N*)⟩
using *that* **by** (*fastforce simp*: *correct-watching-init.simps dom-m-fmrestrict-set*′)+
then have ⟨*length* (*map fst* (σ *L*)) ≤ *card* (*set l1* ∩ *set-mset* (*dom-m N*))⟩
using *1* **by** (*subst distinct-card*[*symmetric*])
(*auto simp*: *distinct-watched-alt-def intro*!: *card-mono intro*: *order-trans*)
also have ⟨... < *card* (*set-mset* (*dom-m N*))⟩
using *that* **by** (*auto intro*!: *psubset-card-mono*)
also have ⟨... = *size* (*dom-m N*)⟩
by (*simp add*: *distinct-mset-dom distinct-mset-size-eq-card*)
finally show *?thesis* **by** *simp*
qed
show *?thesis*
unfolding *rewatch-def*
apply (*refine-vcg*
nfoldli-rule[**where** *I* = ⟨*I*⟩])
subgoal by (*rule I0*)
subgoal using *assms* **unfolding** *I-def* **by** *auto*
subgoal for *x xa l1 l2* σ **using** *H*[*of xa*] **unfolding** *I-def* **apply** −
by (*rule*, *subst* (*asm*)*nth-eq-iff-index-eq*)
linarith+
subgoal for *x xa l1 l2* σ **unfolding** *I-def* **by** (*rule le*)
subgoal for *x xa l1 l2* σ **unfolding** *I-def* **by** (*drule le*[**where** *L* = ⟨*N* ∝ *xa* ! *1*⟩]) (*auto simp*: *I-def dest*!: *le*)
subgoal for *x xa l1 l2* σ
unfolding *I-def*
by (*cases* ⟨*the* (*fmlookup N xa*)⟩)
(*auto simp*: *dom-m-fmrestrict-set*′ *intro*!: *correct-watching-init-add-clause*)
subgoal
unfolding *I-def* **by** *auto*
subgoal by *auto*
subgoal unfolding *I-def*
by (*auto simp*: *fmlookup-restrict-set-id*′)
done
qed

definition *state-wl-l-init-full* :: ⟨(′*v twl-st-wl-init-full* × ′*v twl-st-l-init*) *set*⟩ **where**
⟨*state-wl-l-init-full* = {(*S*, *S*′). (*fst S*, *fst S*′) ∈ *state-wl-l None* ∧
snd S = *snd S*′}⟩

definition *added-only-watched* :: ⟨(′*v twl-st-wl-init-full* × ′*v twl-st-wl-init*) *set*⟩ **where**
⟨*added-only-watched* = {(((*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*), *OC*), ((*M*′, *N*′, *D*′, *NE*′, *UE*′, *Q*′), *OC*′)).
(*M*, *N*, *D*, *NE*, *UE*, *Q*) = (*M*′, *N*′, *D*′, *NE*′, *UE*′, *Q*′) ∧ *OC* = *OC*′}⟩

definition *init-dt-wl-spec-full*
:: ⟨′*v clause-l list* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init-full* ⇒ *bool*⟩
where
⟨*init-dt-wl-spec-full C S T*′′ ⟷
(∃ *S*′ *T T*′. (*S*, *S*′) ∈ *state-wl-l-init* ∧ (*T* :: ′*v twl-st-wl-init*, *T*′) ∈ *state-wl-l-init* ∧
init-dt-spec C S′ *T*′ ∧ *correct-watching-init* (*fst T*′′) ∧ (*T*′′, *T*) ∈ *added-only-watched*)⟩

definition *init-dt-wl-full* :: ⟨′*v clause-l list* ⇒ ′*v twl-st-wl-init* ⇒ ′*v twl-st-wl-init-full nres*⟩ **where**
⟨*init-dt-wl-full CS S* = *do*{
((*M*, *N*, *D*, *NE*, *UE*, *Q*), *OC*) ← *init-dt-wl CS S*;
W ← *rewatch N* (λ-. []);
RETURN ((*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*), *OC*)

}⟩

lemma *init-dt-wl-spec-rewatch-pre*:
assumes ⟨*init-dt-wl-spec CS S T*⟩ **and** ⟨*N = get-clauses-init-wl T*⟩ **and** ⟨*C ∈# dom-m N*⟩
shows ⟨*distinct* (*N* ∝ *C*) ∧ *length* (*N* ∝ *C*) ≥ *2*⟩
proof −
obtain *x xa xb* **where**
⟨*N = get-clauses-init-wl T*⟩ **and**
Sx: ⟨(*S*, *x*) ∈ *state-wl-l-init*⟩ **and**
Txa: ⟨(*T*, *xa*) ∈ *state-wl-l-init*⟩ **and**
xa-xb: ⟨(*xa*, *xb*) ∈ *twl-st-l-init*⟩ **and**
struct-invs: ⟨*twl-struct-invs-init xb*⟩ **and**
⟨*clauses-to-update-l-init xa* = {#}⟩ **and**
⟨∀ *s*∈*set* (*get-trail-l-init xa*). ¬ *is-decided s*⟩ **and**
⟨*get-conflict-l-init xa = None* ⟶
literals-to-update-l-init xa = uminus '# *lit-of* '# *mset* (*get-trail-l-init xa*)⟩ **and**
⟨*mset* '# *mset CS* + *mset* '# *ran-mf* (*get-clauses-l-init x*) + *other-clauses-l-init x* +
get-unit-clauses-l-init x =
mset '# *ran-mf* (*get-clauses-l-init xa*) + *other-clauses-l-init xa* +
get-unit-clauses-l-init xa⟩ **and**
⟨*learned-clss-lf* (*get-clauses-l-init x*) =
learned-clss-lf (*get-clauses-l-init xa*)⟩ **and**
⟨*get-learned-unit-clauses-l-init xa = get-learned-unit-clauses-l-init x*⟩ **and**
⟨*twl-list-invs* (*fst xa*)⟩ **and**
⟨*twl-stgy-invs* (*fst xb*)⟩ **and**
⟨*other-clauses-l-init xa* ≠ {#} ⟶ *get-conflict-l-init xa* ≠ *None*⟩ **and**
⟨{#} ∈# *mset* '# *mset CS* ⟶ *get-conflict-l-init xa* ≠ *None*⟩ **and**
⟨*get-conflict-l-init x* ≠ *None* ⟶ *get-conflict-l-init x = get-conflict-l-init xa*⟩
using *assms*
unfolding *init-dt-wl-spec-def init-dt-spec-def* **apply** −
by *normalize-goal*+ *presburger*

have ⟨*twl-st-inv* (*fst xb*)⟩
using *struct-invs* **unfolding** *twl-struct-invs-init-def* **by** *fast*
then have ⟨*Multiset.Ball* (*get-clauses* (*fst xb*)) *struct-wf-twl-cls*⟩
by (*cases xb*) (*auto simp*: *twl-st-inv.simps*)
with ⟨*C* ∈# *dom-m N*⟩ **show** *?thesis*
using *Txa xa-xb assms* **by** (*cases T*; *cases* ⟨*fmlookup N C*⟩; *cases* ⟨*snd* (*the*(*fmlookup N C*))⟩)
(*auto simp*: *state-wl-l-init-def twl-st-l-init-def conj-disj-distribR Collect-disj-eq*
Collect-conv-if mset-take-mset-drop-mset'
state-wl-l-init'-def ran-m-def dest!: *multi-member-split*)
qed

lemma *init-dt-wl-full-init-dt-wl-spec-full*:
assumes ⟨*init-dt-wl-pre CS S*⟩
shows ⟨*init-dt-wl-full CS S* ≤ *SPEC* (*init-dt-wl-spec-full CS S*)⟩
proof −
show *?thesis*
unfolding *init-dt-wl-full-def*
apply (*rule specify-left*)
apply (*rule init-dt-wl-init-dt-wl-spec*)
subgoal by (*rule assms*)
apply *clarify*
apply (*rule specify-left*)
apply (*rule-tac M =a* **and** *N=aa* **and** *C=ab* **and** *NE=ac* **and** *UE=ad* **and** *Q=b* **in**
rewatch-correctness[*OF - init-dt-wl-spec-rewatch-pre*])

subgoal by *rule*
  apply *assumption*
subgoal by *simp*
subgoal by *simp*
subgoal for *a aa ab ac ad b ba W*
  using *assms*
  unfolding *init-dt-wl-spec-full-def init-dt-wl-pre-def init-dt-wl-spec-def*
  by (*auto simp*: *added-only-watched-def state-wl-l-init-def state-wl-l-init′-def*)
done
qed

end
theory *CDCL-Conflict-Minimisation*
  imports
    Watched-Literals-Watch-List-Domain
    WB-More-Refinement
    WB-More-Refinement-List List−Index.List-Index HOL−Imperative-HOL.Imperative-HOL
begin

We implement the conflict minimisation as presented by Sörensson and Biere ("Minimizing Learned Clauses"').

We refer to the paper for further details, but the general idea is to produce a series of resolution steps such that eventually (i.e., after enough resolution steps) no new literals has been introduced in the conflict clause.

The resolution steps are only done with the reasons of the of literals appearing in the trail. Hence these steps are terminating: we are "shortening" the trail we have to consider with each resolution step. Remark that the shortening refers to the length of the trail we have to consider, not the levels.

The concrete proof was harder than we initially expected. Our first proof try was to certify the resolution steps. While this worked out, adding caching on top of that turned to be rather hard, since it is not obvious how to add resolution steps in the middle of the current proof if the literal has already been removed (basically we would have to prove termination and confluence of the rewriting system). Therefore, we worked instead directly on the entailment of the literals of the conflict clause (up to the point in the trail we currently considering, which is also the termination measure). The previous try is still present in our formalisation (see *minimize-conflict-support*, which we however only use for the termination proof).

The algorithm presented above does not distinguish between literals propagated at the same level: we cannot reuse information about failures to cut branches. There is a variant of the algorithm presented above that is able to do so (Van Gelder, "Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces"). The algorithm is however more complicated and has only be implemented in very few solvers (at least lingeling and cadical) and is especially not part of glucose nor cryptominisat. Therefore, we have decided to not implement it: It is probably not worth it and requires some additional data structures.

declare $cdcl_W$*-restart-mset-state*[*simp*]

type-synonym *out-learned* = ‹*nat clause-l*›

The data structure contains the (unique) literal of highest at position one. This is useful since this is what we want to have at the end (propagation clause) and we can skip the first literal when minimising the clause.

definition *out-learned* :: ‹(*nat*, *nat*) *ann-lits* ⇒ *nat clause option* ⇒ *out-learned* ⇒ *bool*› **where**

‹out-learned M D out ⟷
  out ≠ [] ∧
  (D = None ⟶ length out = 1) ∧
  (D ≠ None ⟶ mset (tl out) = filter-mset (λL. get-level M L < count-decided M) (the D))›

definition *out-learned-confl* :: ‹(nat, nat) ann-lits ⇒ nat clause option ⇒ out-learned ⇒ bool› **where**
‹out-learned-confl M D out ⟷
  out ≠ [] ∧ (D ≠ None ∧ mset out = the D)›

lemma *out-learned-Cons-None*[*simp*]:
‹out-learned (L # aa) None ao ⟷ out-learned aa None ao›
by (*auto simp*: *out-learned-def*)

lemma *out-learned-tl-None*[*simp*]:
‹out-learned (tl aa) None ao ⟷ out-learned aa None ao›
by (*auto simp*: *out-learned-def*)

definition *index-in-trail* :: ‹('v, 'a) ann-lits ⇒ 'v literal ⇒ nat› **where**
‹index-in-trail M L = index (map (atm-of o lit-of) (rev M)) (atm-of L)›

lemma *Propagated-in-trail-entailed*:
assumes
  invs: ‹$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, N, U, D)› **and**
  in-trail: ‹Propagated L C ∈ set M›
shows
  ‹M ⊨as CNot (remove1-mset L C)› **and** ‹L ∈# C› **and** ‹N + U ⊨pm C› **and**
  ‹K ∈# remove1-mset L C ⟹ index-in-trail M K < index-in-trail M L› **and**
  ‹¬tautology C› **and** ‹distinct-mset C›
proof −
  obtain *M2 M1* **where**
    M: ‹M = M2 @ Propagated L C # M1›
    using *split-list*[*OF in-trail*] **by** *metis*
  have ‹a @ Propagated L mark # b = trail (M, N, U, D) ⟶
      b ⊨as CNot (remove1-mset L mark) ∧ L ∈# mark› **and**
    dist: ‹$cdcl_W$-restart-mset.distinct-$cdcl_W$-state (M, N, U, D)›
    for *L mark a b*
    using *invs*
    unfolding *$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv-def*
      $cdcl_W$-restart-mset.$cdcl_W$-conflicting-def
    by *fast+*
  then have *L-E*: ‹L ∈# C› **and** *M1-E*: ‹M1 ⊨as CNot (remove1-mset L C)›
    unfolding *M* **by** *force+*
  then have *M-E*: ‹M ⊨as CNot (remove1-mset L C)›
    unfolding *M* **by** (*simp add*: *true-annots-append-l*)
  show ‹M ⊨as CNot (remove1-mset L C)› **and** ‹L ∈# C›
    using *L-E M-E* **by** *fast+*
  have ‹set (get-all-mark-of-propagated (trail (M, N, U, D)))
    ⊆ set-mset ($cdcl_W$-restart-mset.clauses (M, N, U, D))›
    using *invs*
    unfolding *$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv-def*
      $cdcl_W$-restart-mset.$cdcl_W$-learned-clause-alt-def
    by *fast*
  then have ‹C ∈# N + U›
    using *in-trail $cdcl_W$-restart-mset.in-get-all-mark-of-propagated-in-trail*[*of C M*]
    by (*auto simp*: *clauses-def*)
  then show ‹N + U ⊨pm C› **by** *auto*

have *n-d*: ⟨*no-dup M*⟩
  using *invs*
  unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
    $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
  by *auto*
show ⟨*index-in-trail M K* < *index-in-trail M L*⟩ **if** *K-C*: ⟨*K* ∈# *remove1-mset L C*⟩
proof −
  have
    KL: ⟨*atm-of K* ≠ *atm-of L*⟩ **and**
    uK-M1: ⟨−*K* ∈ *lits-of-l M1*⟩ **and**
    L: ⟨*L* ∉ *lit-of* ‘ (*set M2* ∪ *set M1*)⟩ ⟨−*L* ∉ *lit-of* ‘ (*set M2* ∪ *set M1*)⟩
    using *M1-E K-C n-d* **unfolding** *M true-annots-true-cls-def-iff-negation-in-model*
    by (*auto dest*!: *multi-member-split simp*: *atm-of-eq-atm-of lits-of-def uminus-lit-swap*
      Decided-Propagated-in-iff-in-lits-of-l)
  have *L-M1*: ⟨*atm-of L* ∉ (*atm-of* ∘ *lit-of*) ‘ *set M1*⟩
    using *L* **by** (*auto simp*: *image-Un atm-of-eq-atm-of*)
  have *K-M1*: ⟨*atm-of K* ∈ (*atm-of* ∘ *lit-of*) ‘ *set M1*⟩
    using *uK-M1* **by** (*auto simp*: *lits-of-def image-image comp-def uminus-lit-swap*)
  show *?thesis*
    using *KL L-M1 K-M1* **unfolding** *index-in-trail-def M* **by** (*auto simp*: *index-append*)
qed
have ⟨¬*tautology*(*remove1-mset L C*)⟩
  by (*rule consistent-CNot-not-tautology*[*of* ⟨*lits-of-l M1*⟩])
   (*use n-d M1-E* **in** ⟨*auto dest*: *distinct-consistent-interp no-dup-appendD*
     simp: *true-annots-true-cls M*⟩)
then show ⟨¬*tautology C*⟩
  using *multi-member-split*[*OF L-E*] *M1-E n-d*
  by (*auto simp*: *tautology-add-mset true-annots-true-cls-def-iff-negation-in-model M*
     dest!: *multi-member-split in-lits-of-l-defined-litD*)
show ⟨*distinct-mset* (*C*)⟩
  using *dist* ⟨*C* ∈# *N* + *U*⟩ **unfolding** $cdcl_W$*-restart-mset.distinct-*$cdcl_W$*-state-def*
  by (*auto dest*: *multi-member-split*)
qed

This predicate corresponds to one resolution step.

inductive *minimize-conflict-support* :: ⟨(*'v*, *'v clause*) *ann-lits* ⇒ *'v clause* ⇒ *'v clause* ⇒ *bool*⟩
  for *M* **where**
resolve-propa:
  ⟨*minimize-conflict-support M* (*add-mset* (−*L*) *C*) (*C* + *remove1-mset L E*)⟩
  if ⟨*Propagated L E* ∈ *set M*⟩ |
remdups: ⟨*minimize-conflict-support M* (*add-mset L C*) *C*⟩

lemma *index-in-trail-uminus*[*simp*]: ⟨*index-in-trail M* (−*L*) = *index-in-trail M L*⟩
  by (*auto simp*: *index-in-trail-def*)

This is the termination argument of the conflict minimisation: the multiset of the levels decreases (for the multiset ordering).

definition *minimize-conflict-support-mes* :: ⟨(*'v*, *'v clause*) *ann-lits* ⇒ *'v clause* ⇒ *nat multiset*⟩
where
  ⟨*minimize-conflict-support-mes M C* = *index-in-trail M* ‘# *C*⟩

context
  fixes *M* :: ⟨(*'v*, *'v clause*) *ann-lits*⟩ **and** *N U* :: ⟨*'v clauses*⟩ **and**

D :: ⟨′v clause option⟩
assumes *invs*: ⟨$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, N, U, D)⟩
begin

private lemma
no-dup: ⟨no-dup M⟩ **and**
consistent: ⟨consistent-interp (lits-of-l M)⟩
using *invs* **unfolding** *$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv-def*
$cdcl_W$-restart-mset.$cdcl_W$-M-level-inv-def
by *simp-all*

lemma *minimize-conflict-support-entailed-trail*:
assumes ⟨minimize-conflict-support M C E⟩ **and** ⟨M ⊨as CNot C⟩
shows ⟨M ⊨as CNot E⟩
using *assms*
proof (*induction rule*: *minimize-conflict-support.induct*)
case (*resolve-propa L E C*) **note** *in-trail* = *this*(1) **and** *M-C* = *this*(2)
then show *?case*
using *Propagated-in-trail-entailed*[*OF invs in-trail*] **by** (*auto dest*!: *multi-member-split*)
next
case (*remdups L C*)
then show *?case*
by *auto*
qed

lemma *rtranclp-minimize-conflict-support-entailed-trail*:
assumes ⟨(minimize-conflict-support M)** C E⟩ **and** ⟨M ⊨as CNot C⟩
shows ⟨M ⊨as CNot E⟩
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
subgoal by *fast*
subgoal using *minimize-conflict-support-entailed-trail* **by** *fast*
done

lemma *minimize-conflict-support-mes*:
assumes ⟨minimize-conflict-support M C E⟩
shows ⟨minimize-conflict-support-mes M E < minimize-conflict-support-mes M C⟩
using *assms* **unfolding** *minimize-conflict-support-mes-def*
proof (*induction rule*: *minimize-conflict-support.induct*)
case (*resolve-propa L E C*) **note** *in-trail* = *this*
let *?f* = ⟨λxa. index (map (λa. atm-of (lit-of a)) (rev M)) xa⟩
have ⟨?f (atm-of x) < ?f (atm-of L)⟩ **if** *x*: ⟨x ∈# remove1-mset L E⟩ **for** *x*
proof −
obtain *M2 M1* **where**
M: ⟨M = M2 @ Propagated L E # M1⟩
using *split-list*[*OF in-trail*] **by** *metis*
have ⟨a @ Propagated L mark # b = trail (M, N, U, D) ⟶
b ⊨as CNot (remove1-mset L mark) ∧ L ∈# mark⟩ **for** *L mark a b*
using *invs*
unfolding *$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv-def*
$cdcl_W$-restart-mset.$cdcl_W$-conflicting-def
by *fast*
then have *L-E*: ⟨L ∈# E⟩ **and** *M-E*: ⟨M1 ⊨as CNot (remove1-mset L E)⟩
unfolding *M* **by** *force+*
then have ⟨−x ∈ lits-of-l M1⟩
using *x* **unfolding** *true-annots-true-cls-def-iff-negation-in-model* **by** *auto*
then have ⟨?f (atm-of x) < length M1⟩

using *no-dup*
by (*auto simp*: *M lits-of-def index-append Decided-Propagated-in-iff-in-lits-of-l uminus-lit-swap*)
moreover have ‹*?f* (*atm-of L*) = *length M1*›
using *no-dup* **unfolding** *M* **by** (*auto simp*: *index-append Decided-Propagated-in-iff-in-lits-of-l atm-of-eq-atm-of lits-of-def*)
ultimately show *?thesis* **by** *auto*
qed

then show *?case* **by** (*auto simp*: *comp-def index-in-trail-def*)
next
case (*remdups L C*)
then show *?case* **by** *auto*
qed

lemma *wf-minimize-conflict-support*:
shows ‹*wf* {(*C'*, *C*). *minimize-conflict-support M C C'*}›
apply (*rule wf-if-measure-in-wf*[*of* ‹{(*C'*, *C*). *C'* < *C*}› - ‹*minimize-conflict-support-mes M*›])
subgoal using *wf* **.**
subgoal using *minimize-conflict-support-mes* **by** *auto*
done
end

lemma *conflict-minimize-step*:
assumes
‹*NU* ⊨*p add-mset L C*› **and**
‹*NU* ⊨*p add-mset* (−*L*) *D*› **and**
‹⋀*K'*. *K'* ∈# *C* ⟹ *NU* ⊨*p add-mset* (−*K'*) *D*›
shows ‹*NU* ⊨*p D*›
proof −
have ‹*NU* ⊨*p D* + *C*›
using *assms*(*1*,*2*) *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or* **by** *blast*
then show *?thesis*
using *assms*(*3*)
proof (*induction C*)
case *empty*
then show *?case*
using *true-clss-cls-in true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or* **by** *fastforce*
next
case (*add x C*) **note** *IH* =*this*(*1*) **and** *NU-DC* = *this*(*2*) **and** *entailed* = *this*(*3*)
have ‹*NU* ⊨*p D* + *C* + *D*›
using *entailed*[*of x*] *NU-DC*
true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or[*of NU* ‹−*x*› ‹*D* + *C*› *D*]
by *auto*
then have ‹*NU* ⊨*p D* + *C*›
by (*metis add.comm-neutral diff-add-zero sup-subset-mset-def true-clss-cls-sup-iff-add*)
from *IH*[*OF this*] *entailed* **show** *?case* **by** *auto*
qed
qed

This function filters the clause by the levels up the level of the given literal. This is the part the conflict clause that is considered when testing if the given literal is redundant.

definition *filter-to-poslev* **where**
‹*filter-to-poslev M L D* = *filter-mset* (λ*K*. *index-in-trail M K* < *index-in-trail M L*) *D*›

lemma *filter-to-poslev-uminus*[*simp*]:

⟨*filter-to-poslev M* (−*L*) *D* = *filter-to-poslev M L D*⟩
by (*auto simp*: *filter-to-poslev-def*)

lemma *filter-to-poslev-empty*[*simp*]:
⟨*filter-to-poslev M L* {#} = {#}⟩
by (*auto simp*: *filter-to-poslev-def*)

lemma *filter-to-poslev-mono*:
⟨*index-in-trail M K′* ≤ *index-in-trail M L* ⟹
filter-to-poslev M K′ D ⊆# *filter-to-poslev M L D*⟩
unfolding *filter-to-poslev-def*
by (*auto simp*: *multiset-filter-mono2*)

lemma *filter-to-poslev-mono-entailement*:
⟨*index-in-trail M K′* ≤ *index-in-trail M L* ⟹
NU ⊨*p filter-to-poslev M K′ D* ⟹ *NU* ⊨*p filter-to-poslev M L D*⟩
by (*metis* (*full-types*) *filter-to-poslev-mono subset-mset.le-iff-add true-clss-cls-mono-r*)

lemma *filter-to-poslev-mono-entailement-add-mset*:
⟨*index-in-trail M K′* ≤ *index-in-trail M L* ⟹
NU ⊨*p add-mset J* (*filter-to-poslev M K′ D*) ⟹ *NU* ⊨*p add-mset J* (*filter-to-poslev M L D*)⟩
by (*metis filter-to-poslev-mono mset-subset-eq-add-mset-cancel subset-mset.le-iff-add*
true-clss-cls-mono-r)

lemma *conflict-minimize-intermediate-step*:
assumes
⟨*NU* ⊨*p add-mset L C*⟩ **and**
K′-C: ⟨⋀*K′*. *K′* ∈# *C* ⟹ *NU* ⊨*p add-mset* (−*K′*) *D* ∨ *K′* ∈# *D*⟩
shows ⟨*NU* ⊨*p add-mset L D*⟩
proof −
have ⟨*NU* ⊨*p add-mset L C* + *D*⟩
using *assms*(*1*) *true-clss-cls-mono-r* **by** *blast*
then show *?thesis*
using *assms*(*2*)
proof (*induction C*)
case *empty*
then show *?case*
using *true-clss-cls-in true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or* **by** *fastforce*
next
case (*add x C*) **note** *IH* =*this*(*1*) **and** *NU-DC* = *this*(*2*) **and** *entailed* = *this*(*3*)

have *1*: ⟨*NU* ⊨*p add-mset x* (*add-mset L* (*D* + *C*))⟩
using *NU-DC* **by** (*auto simp*: *add-mset-commute ac-simps*)
moreover have *2*: ⟨*remdups-mset* (*add-mset L* (*D* + *C* + *D*)) = *remdups-mset* (*add-mset L* (*C* +
D))⟩
by (*auto simp*: *remdups-mset-def*)
moreover have *3*: ⟨*remdups-mset* (*D* + *C* + *D*) = *remdups-mset* (*D* + *C*)⟩
by (*auto simp*: *remdups-mset-def*)
moreover have ⟨*x* ∈# *D* ⟹ *NU* ⊨*p add-mset L* (*D* + *C* + *D*)⟩
using *1*
apply (*subst* (*asm*) *true-clss-cls-remdups-mset*[*symmetric*])
apply (*subst true-clss-cls-remdups-mset*[*symmetric*])
by (*auto simp*: *2 3*)
ultimately have ⟨*NU* ⊨*p add-mset L* (*D* + *C* + *D*)⟩
using *entailed*[*of x*] *NU-DC*
true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or[*of NU* ⟨−*x*⟩ ⟨*add-mset L D* + *C*⟩ *D*]

by *auto*
moreover have ‹*remdups-mset* (*D* + (*C* + *D*)) = *remdups-mset* (*D* + *C*)›
by (*auto simp*: *remdups-mset-def*)
ultimately have ‹*NU* ⊨*p* *add-mset L C* + *D*›
apply (*subst true-clss-cls-remdups-mset*[*symmetric*])
apply (*subst* (*asm*) *true-clss-cls-remdups-mset*[*symmetric*])
by (*auto simp add*: *3 2 add.commute simp del*: *true-clss-cls-remdups-mset*)
from *IH*[*OF this*] *entailed* **show** *?case* **by** *auto*
qed
qed

lemma *conflict-minimize-intermediate-step-filter-to-poslev*:
assumes
lev-K-L: ‹$\bigwedge K'$. $K' \in\#$ *C* ⟹ *index-in-trail M K'* < *index-in-trail M L*› **and**
NU-LC: ‹*NU* ⊨*p* *add-mset L C*› **and**
K'-C: ‹$\bigwedge K'$. $K' \in\#$ *C* ⟹ *NU* ⊨*p* *add-mset* (−*K'*) (*filter-to-poslev M L D*) ∨
K' ∈# *filter-to-poslev M L D*›
shows ‹*NU* ⊨*p* *add-mset L* (*filter-to-poslev M L D*)›
proof −
have *C-entailed*: ‹*K'* ∈# *C* ⟹ *NU* ⊨*p* *add-mset* (−*K'*) (*filter-to-poslev M L D*) ∨
K' ∈# *filter-to-poslev M L D*› **for** *K'*
using *filter-to-poslev-mono*[*of M K' L D*] *lev-K-L*[*of K'*] *K'-C*[*of K'*]
true-clss-cls-mono-r[*of* - ‹ *add-mset* (− *K'*) (*filter-to-poslev M K' D*)›]
by (*auto simp*: *mset-subset-eq-exists-conv*)
show *?thesis*
using *conflict-minimize-intermediate-step*[*OF NU-LC C-entailed*] **by** *fast*
qed

datatype *minimize-status* = *SEEN-FAILED* | *SEEN-REMOVABLE* | *SEEN-UNKNOWN*

instance *minimize-status* :: *heap*
proof *standard*
let *?f* = ‹λ*s*. *case s of SEEN-FAILED* ⇒ (*0* :: *nat*) | *SEEN-REMOVABLE* ⇒ *1* | *SEEN-UNKNOWN*
⇒ *2*›
have ‹*inj ?f*›
by (*auto simp*: *inj-def split*: *minimize-status.splits*)
then show ‹∃ *to-nat*. *inj* (*to-nat* :: *minimize-status* ⇒ *nat*)›
by *blast*
qed

instantiation *minimize-status* :: *default*
begin
definition *default-minimize-status* **where**
‹*default-minimize-status* = *SEEN-UNKNOWN*›

instance by *standard*
end

type-synonym *'v conflict-min-analyse* = ‹(*'v literal* × *'v clause*) *list*›
type-synonym *'v conflict-min-cach* = ‹*'v* ⇒ *minimize-status*›

definition *get-literal-and-remove-of-analyse*
:: ‹*'v conflict-min-analyse* ⇒ (*'v literal* × *'v conflict-min-analyse*) *nres*› **where**
‹*get-literal-and-remove-of-analyse analyse* =
SPEC(λ(*L*, *ana*). *L* ∈# *snd* (*hd analyse*) ∧ *tl ana* = *tl analyse* ∧ *ana* ≠ [] ∧
hd ana = (*fst* (*hd analyse*), *snd* (*hd* (*analyse*)) − {#*L*#}))›

definition *mark-failed-lits*
:: ‹- ⇒ 'v conflict-min-analyse ⇒ 'v conflict-min-cach ⇒ 'v conflict-min-cach nres›
where
‹*mark-failed-lits NU analyse cach = SPEC*(λ*cach'*.
($\forall$ *L*. *cach' L = SEEN-REMOVABLE* $\longrightarrow$ *cach L = SEEN-REMOVABLE*))›

definition *conflict-min-analysis-inv*
:: ‹('v, 'a) ann-lits ⇒ 'v conflict-min-cach ⇒ 'v clauses ⇒ 'v clause ⇒ bool›
where
‹*conflict-min-analysis-inv M cach NU D* $\longleftrightarrow$
($\forall$ *L*. $-L \in$ *lits-of-l M* $\longrightarrow$ *cach* (*atm-of L*) = *SEEN-REMOVABLE* $\longrightarrow$
set-mset NU $\models$*p add-mset* ($-L$) (*filter-to-poslev M L D*)))›

lemma *conflict-min-analysis-inv-update-removable*:
‹*no-dup M* $\Longrightarrow$ $-L \in$ *lits-of-l M* $\Longrightarrow$
conflict-min-analysis-inv M (*cach*(*atm-of L* := *SEEN-REMOVABLE*)) *NU D* $\longleftrightarrow$
conflict-min-analysis-inv M cach NU D $\wedge$ *set-mset NU* $\models$*p add-mset* ($-L$) (*filter-to-poslev M L D*)›
by (*auto simp*: *conflict-min-analysis-inv-def atm-of-eq-atm-of dest*: *no-dup-consistentD*)

lemma *conflict-min-analysis-inv-update-failed*:
‹*conflict-min-analysis-inv M cach NU D* $\Longrightarrow$
conflict-min-analysis-inv M (*cach*(*L* := *SEEN-FAILED*)) *NU D*›
by (*auto simp*: *conflict-min-analysis-inv-def*)

fun *conflict-min-analysis-stack*
:: ‹('v, 'a) ann-lits ⇒ 'v clauses ⇒ 'v clause ⇒ 'v conflict-min-analyse ⇒ bool›
where
‹*conflict-min-analysis-stack M NU D* [] $\longleftrightarrow$ *True*› |
‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # []) $\longleftrightarrow$ $-L \in$ *lits-of-l M*› |
‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # (*L'*, *E'*) # *analyse*) $\longleftrightarrow$
($\exists$ *C*. *set-mset NU* $\models$*p add-mset* ($-L'$) *C* $\wedge$
($\forall$ *K*$\in$#*C*$-$*add-mset L E'*. *set-mset NU* $\models$*p* (*filter-to-poslev M L' D*) + {#$-K$#} $\vee$
K $\in$# *filter-to-poslev M L' D*) $\wedge$
($\forall$ *K*$\in$#*C*. *index-in-trail M K* < *index-in-trail M L'*) $\wedge$
E' $\subseteq$# *C*) $\wedge$
$-L' \in$ *lits-of-l M* $\wedge$
$-L \in$ *lits-of-l M* $\wedge$
index-in-trail M L < *index-in-trail M L'* $\wedge$
conflict-min-analysis-stack M NU D ((*L'*, *E'*) # *analyse*)›

lemma *conflict-min-analysis-stack-change-hd*:
‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # *ana*) $\Longrightarrow$
conflict-min-analysis-stack M NU D ((*L*, *E'*) # *ana*)›
by (*cases ana*, *auto*)

lemma *conflict-min-analysis-stack-sorted*:
‹*conflict-min-analysis-stack M NU D analyse* $\Longrightarrow$
sorted (*map* (*index-in-trail M o fst*) *analyse*)›
by (*induction rule*: *conflict-min-analysis-stack.induct*)
auto
lemma *conflict-min-analysis-stack-sorted-and-distinct*:
‹*conflict-min-analysis-stack M NU D analyse* $\Longrightarrow$
sorted (*map* (*index-in-trail M o fst*) *analyse*) $\wedge$

distinct (*map* (*index-in-trail M o fst*) *analyse*)›
by (*induction rule*: *conflict-min-analysis-stack.induct*)
auto

lemma *conflict-min-analysis-stack-distinct-fst*:
assumes ‹*conflict-min-analysis-stack M NU D analyse*›
shows ‹*distinct* (*map fst analyse*)› **and** ‹*distinct* (*map* (*atm-of o fst*) *analyse*)›
proof −
have *dist*: ‹*distinct* (*map* (*index-in-trail M* ∘ *fst*) *analyse*)›
using *conflict-min-analysis-stack-sorted-and-distinct*[*of M NU D analyse*, *OF assms*]
by *auto*
then show ‹*distinct* (*map fst analyse*)›
by (*auto simp*: *intro*!: *distinct-mapI*[*of* ‹(*index-in-trail M*)›])
show ‹*distinct* (*map* (*atm-of o fst*) *analyse*)›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
from *not-distinct-decomp*[*OF this*]
obtain *xs L ys zs* **where** ‹*map* (*atm-of o fst*) *analyse* = *xs* @ *L* # *ys* @ *L* # *zs*›
by *auto*
then show *False*
using *dist*
by (*auto simp*: *map-eq-append-conv atm-of-eq-atm-of Int-Un-distrib image-Un*)
qed
qed

lemma *conflict-min-analysis-stack-neg*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹
M ⊨*as CNot* (*fst* ‘# *mset analyse*)›
by (*induction M NU D analyse rule*: *conflict-min-analysis-stack.induct*)
auto

fun *conflict-min-analysis-stack-hd*
:: ‹(′*v*, ′*a*) *ann-lits* ⇒ ′*v clauses* ⇒ ′*v clause* ⇒ ′*v conflict-min-analyse* ⇒ *bool*›
where
‹*conflict-min-analysis-stack-hd M NU D* [] ⟷ *True*› |
‹*conflict-min-analysis-stack-hd M NU D* ((*L*, *E*) # -) ⟷
(∃ *C*. *set-mset NU* ⊨*p add-mset* (−*L*) *C* ∧
(∀ *K*∈#*C*. *index-in-trail M K* < *index-in-trail M L*) ∧ *E* ⊆# *C* ∧ −*L* ∈ *lits-of-l M* ∧
(∀ *K*∈#*C*−*E*. *set-mset NU* ⊨*p* (*filter-to-poslev M L D*) + {#−*K*#} ∨ *K* ∈# *filter-to-poslev M L D*))›
lemma *conflict-min-analysis-stack-tl*:
‹*conflict-min-analysis-stack M NU D analyse* ⟹ *conflict-min-analysis-stack M NU D* (*tl analyse*)›
by (*cases* ‹(*M*, *NU*, *D*, *analyse*)› *rule*: *conflict-min-analysis-stack.cases*) *auto*

definition *lit-redundant-inv*
:: ‹(′*v*, ′*v clause*) *ann-lits* ⇒ ′*v clauses* ⇒ ′*v clause* ⇒ ′*v conflict-min-analyse* ⇒
′*v conflict-min-cach* × ′*v conflict-min-analyse* × *bool* ⇒ *bool*› **where**
‹*lit-redundant-inv M NU D init-analyse* = (λ(*cach*, *analyse*, *b*).
conflict-min-analysis-inv M cach NU D ∧
(*analyse* ≠ [] ⟶ *fst* (*hd init-analyse*) = *fst* (*last analyse*)) ∧
(*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analyse*))) = *SEEN-REMOVABLE*) ∧
conflict-min-analysis-stack M NU D analyse ∧
conflict-min-analysis-stack-hd M NU D analyse)›

definition *lit-redundant-rec-loop-inv* :: ‹(′*v*, ′*v clause*) *ann-lits* ⇒

$'v$ *conflict-min-cach* × $'v$ *conflict-min-analyse* × *bool* ⇒ *bool*⟩ **where**
⟨*lit-redundant-rec-loop-inv M* = (λ(*cach*, *analyse*, *b*).
  (*uminus o fst*) ‘# *mset analyse* ⊆# *lit-of* ‘# *mset M* ∧
  (∀ *L* ∈ *set analyse*. *cach* (*atm-of* (*fst L*)) = *SEEN-UNKNOWN*))⟩

definition *lit-redundant-rec* :: ⟨($'v$, $'v$ *clause*) *ann-lits* ⇒ $'v$ *clauses* ⇒ $'v$ *clause* ⇒
  $'v$ *conflict-min-cach* ⇒ $'v$ *conflict-min-analyse* ⇒
  ($'v$ *conflict-min-cach* × $'v$ *conflict-min-analyse* × *bool*) *nres*⟩
where
⟨*lit-redundant-rec M NU D cach analysis* =
  $WHILE_T{}^{lit\text{-}redundant\text{-}rec\text{-}loop\text{-}inv\ M}$
  (λ(*cach*, *analyse*, *b*). *analyse* ≠ [])
  (λ(*cach*, *analyse*, *b*). *do* {
    ASSERT(*analyse* ≠ []);
    ASSERT(*length analyse* ≤ *length M*);
    ASSERT(−*fst* (*hd analyse*) ∈ *lits-of-l M*);
    if snd (*hd analyse*) = {#}
    then
      RETURN(*cach* (*atm-of* (*fst* (*hd analyse*)) := *SEEN-REMOVABLE*), *tl analyse*, *True*)
    else do {
      (*L*, *analyse*) ← *get-literal-and-remove-of-analyse analyse*;
      ASSERT(−*L* ∈ *lits-of-l M*);
      b ← *RES UNIV*;
      if (*get-level M L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ∨ *L* ∈# *D*)
      then RETURN (*cach*, *analyse*, *False*)
      else if b ∨ *cach* (*atm-of L*) = *SEEN-FAILED*
      then do {
        cach ← *mark-failed-lits NU analyse cach*;
        RETURN (*cach*, [], *False*)
      }
      else do {
        ASSERT(*cach* (*atm-of L*) = *SEEN-UNKNOWN*);
        C ← *get-propagation-reason M* (−*L*);
        case C of
          Some C ⇒ *do* {
  ASSERT (*distinct-mset C* ∧ ¬*tautology C*);
  RETURN (*cach*, (*L*, *C* − {#−*L*#}) # *analyse*, *False*)}
        | *None* ⇒ *do* {
            cach ← *mark-failed-lits NU analyse cach*;
            RETURN (*cach*, [], *False*)
        }
      }
    }
  })
  (*cach*, *analysis*, *False*)⟩

definition *lit-redundant-rec-spec* **where**
⟨*lit-redundant-rec-spec M NU D L* =
  SPEC(λ(*cach*, *analysis*, *b*). (*b* ⟶ *NU* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M L D*)) ∧
  conflict-min-analysis-inv M cach NU D)⟩

lemma *WHILEIT-rule-stronger-inv-keepI′*:
assumes
⟨*wf R*⟩ **and**
⟨*I s*⟩ **and**
⟨*I′ s*⟩ **and**

⟨⋀s. I s ⟹ I′ s ⟹ b s ⟹ f s ≤ SPEC (λs′. I′ s′)⟩ **and**
⟨⋀s. I s ⟹ I′ s ⟹ b s ⟹ f s ≤ SPEC (λs′. I′ s′ ⟶ (I s′ ∧ (s′, s) ∈ R))⟩ **and**
⟨⋀s. I s ⟹ I′ s ⟹ ¬ b s ⟹ Φ s⟩
shows ⟨$WHILE_T{}^I$ b f s ≤ SPEC Φ⟩
proof −
have *A*[*iff*]: ⟨f s ≤ SPEC (λv. I′ v ∧ I v ∧ (v, s) ∈ R) ⟷ f s ≤ SPEC (λs′. I s′ ∧ I′ s′ ∧ (s′, s) ∈ R)⟩ **for** *s*
by (*rule cong*[*of* ⟨ λn. f s ≤ n⟩]) *auto*
then have *H*: ⟨I s ⟹ I′ s ⟹ b s ⟹ f s ≤ SPEC (λs′. I s′ ∧ I′ s′ ∧ (s′, s) ∈ R)⟩ **for** *s*
using *SPEC-rule-conjI* [*OF assms*(*4*,*5*)[*of s*]] **by** *auto*
have ⟨$WHILE_T{}^I$ b f s ≤ $WHILE_T{}^{\lambda s.\ I\ s\ \wedge\ I'\ s}$ b f s⟩
by (*metis* (*mono-tags*, *lifting*) *WHILEIT-weaken*)

also have ⟨$WHILE_T{}^{\lambda s.\ I\ s\ \wedge\ I'\ s}$ b f s ≤ SPEC Φ⟩
by (*rule WHILEIT-rule*) (*use assms H* **in** ⟨*auto simp*: ⟩)
finally show *?thesis* **.**
qed

lemma *lit-redundant-rec-spec*:
fixes *L* :: ⟨′v literal⟩
assumes *invs*: ⟨$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, N + NE, U + UE, D′)⟩
assumes
init-analysis: ⟨init-analysis = [(L, C)]⟩ **and**
in-trail: ⟨Propagated (−L) (add-mset (−L) C) ∈ set M⟩ **and**
⟨conflict-min-analysis-inv M cach (N + NE + U + UE) D⟩ **and**
L-D: ⟨L ∈# D⟩ **and**
M-D: ⟨M ⊨as CNot D⟩ **and**
unknown: ⟨cach (atm-of L) = SEEN-UNKNOWN⟩
shows
⟨lit-redundant-rec M (N + U) D cach init-analysis ≤
lit-redundant-rec-spec M (N + U + NE + UE) D L⟩
proof −
let *?N* = ⟨N + NE + U + UE⟩
obtain *M2 M1* **where**
M: ⟨M = M2 @ Propagated (− L) (add-mset (− L) C) # M1⟩
using *split-list*[*OF in-trail*] **by** (*auto 5 5*)
have ⟨a @ Propagated L mark # b = trail (M, N + NE, U + UE, D′) ⟶
b ⊨as CNot (remove1-mset L mark) ∧ L ∈# mark⟩ **for** *L mark a b*
using *invs*
unfolding $cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv-def*
$cdcl_W$-*restart-mset*.$cdcl_W$-*conflicting-def*
by *fast*
then have ⟨M1 ⊨as CNot C⟩
by (*force simp*: *M*)
then have *M-C*: ⟨M ⊨as CNot C⟩
unfolding *M* **by** (*simp add*: *true-annots-append-l*)
have ⟨set (get-all-mark-of-propagated (trail (M, N + NE, U + UE, D′)))
⊆ set-mset ($cdcl_W$-restart-mset.clauses (M, N + NE, U + UE, D′))⟩
using *invs*
unfolding $cdcl_W$-*restart-mset*.$cdcl_W$-*all-struct-inv-def*
$cdcl_W$-*restart-mset*.$cdcl_W$-*learned-clause-alt-def*
by *fast*
then have ⟨add-mset (−L) C ∈# ?N⟩
using *in-trail* $cdcl_W$-*restart-mset*.*in-get-all-mark-of-propagated-in-trail*[*of* ⟨add-mset (−L) C⟩ *M*]
by (*auto simp*: *clauses-def*)

```
then have NU-C: ‹?N ⊨pm add-mset (- L) C›
  by auto
have n-d: ‹no-dup M›
  using invs
  unfolding cdcl_W-restart-mset.cdcl_W-all-struct-inv-def
     cdcl_W-restart-mset.cdcl_W-M-level-inv-def
  by auto

let ?f = ‹λanalysis. fold-mset (+) D (snd `# mset analysis)›
define I' where
  ‹I' = (λ(cach :: 'v conflict-min-cach, analysis :: 'v conflict-min-analyse, b::bool).
      lit-redundant-inv M ?N D init-analysis (cach, analysis, b) ∧ M ⊨as CNot (?f analysis) ∧
      distinct (map (atm-of o fst) analysis))›
define R where
  ‹R = {((cach :: 'v conflict-min-cach, analysis :: 'v conflict-min-analyse, b::bool),
         (cach' :: 'v conflict-min-cach, analysis' :: 'v conflict-min-analyse, b' :: bool)).
         (analysis' ≠ [] ∧ (minimize-conflict-support M) (?f analysis') (?f analysis)) ∨
         (analysis' ≠ [] ∧ analysis = tl analysis' ∧ snd (hd analysis') = {#}) ∨
         (analysis' ≠ [] ∧ analysis = [])}›
have wf-R: ‹wf R›
proof -
  have R: ‹R =
            {((cach, analysis, b), (cach', analysis', b')).
               analysis' ≠ [] ∧analysis = []} ∪
            ({((cach, analysis, b), (cach', analysis', b')).
                analysis' ≠ [] ∧ (minimize-conflict-support M) (?f analysis') (?f analysis)} ∪
            {((cach, analysis, b), (cach', analysis', b')).
                analysis' ≠ [] ∧ analysis = tl analysis' ∧ snd (hd analysis') = {#}})›
    (is ‹- = ?end ∪ (?Min ∪ ?ana)›)
    unfolding R-def by auto
  have 1: ‹wf {((cach:: 'v conflict-min-cach, analysis:: 'v conflict-min-analyse, b::bool),
      (cach':: 'v conflict-min-cach, analysis':: 'v conflict-min-analyse, b'::bool)).
     length analysis < length analysis'}›
    using wf-if-measure-f[of ‹measure length›, of ‹λ(-, xs, -). xs›] apply auto
    apply (rule subst[of - - wf])
     prefer 2 apply assumption
    apply auto
    done

  have 2: ‹wf {(C', C).minimize-conflict-support M C C'}›
    by (rule wf-minimize-conflict-support[OF invs])
  from wf-if-measure-f[OF this, of ?f]
  have 2: ‹wf {(C', C). minimize-conflict-support M (?f C) (?f C')}›
    by auto
  from wf-fst-wf-pair[OF this, where 'b = bool]
  have ‹wf {((analysis':: 'v conflict-min-analyse, - :: bool),
             (analysis:: 'v conflict-min-analyse,  -:: bool)).
         (minimize-conflict-support M) (?f analysis) (?f analysis')}›
    by blast
  from wf-snd-wf-pair[OF this, where 'b = ‹'v conflict-min-cach›]
  have ‹wf {((M' :: 'v conflict-min-cach, N'), Ma, N).
    (case N' of
     (analysis' :: 'v conflict-min-analyse, - :: bool) ⇒
       λ(analysis, -).
          minimize-conflict-support M (fold-mset (+) D (snd `# mset analysis))
            (fold-mset (+) D (snd `# mset analysis'))) N}›
```

by *blast*
then have *wf-Min*: ‹*wf ?Min*›
apply (*rule wf-subset*)
by *auto*
have *wf-ana*: ‹*wf ?ana*›
by (*rule wf-subset*[*OF 1*]) *auto*
have *wf*: ‹*wf* (*?Min* ∪ *?ana*)›
apply (*rule wf-union-compatible*)
subgoal by (*rule wf-Min*)
subgoal by (*rule wf-ana*)
subgoal by (*auto elim*!: *neq-NilE*)
done
have *wf-end*: ‹*wf ?end*›
proof (*rule ccontr*)
assume ‹¬ *?thesis*›
then obtain *f* **where** *f*: ‹(*f* (*Suc i*), *f i*) ∈ *?end*› **for** *i*
unfolding *wf-iff-no-infinite-down-chain* **by** *auto*
have ‹*fst* (*snd* (*f* (*Suc 0*))) = []›
using *f*[*of 0*] **by** *auto*
moreover have ‹*fst* (*snd* (*f* (*Suc 0*))) ≠ []›
using *f*[*of 1*] **by** *auto*
ultimately show *False* **by** *blast*
qed
show *?thesis*
unfolding *R*
apply (*rule wf-Un*)
subgoal by (*rule wf-end*)
subgoal by (*rule wf*)
subgoal by *auto*
done
qed
have *uL-M*: ‹− *L* ∈ *lits-of-l M*›
using *in-trail* **by** (*force simp*: *lits-of-def*)
then have *init-I*: ‹*lit-redundant-inv M ?N D init-analysis* (*cach*, *init-analysis*, *False*)›
using *assms NU-C Propagated-in-trail-entailed*[*OF invs in-trail*]
unfolding *lit-redundant-inv-def*
by (*auto simp*: *ac-simps*)

have ‹(*minimize-conflict-support M*) *D* (*remove1-mset L* (*C* + *D*))›
using *minimize-conflict-support.resolve-propa*[*OF in-trail*, *of* ‹*remove1-mset L D*›] *L-D*
by (*auto simp*: *ac-simps*)

then have *init-I′*: ‹*I′* (*cach*, *init-analysis*, *False*)›
using *M-D L-D M-C init-I* **unfolding** *I′-def* **by** (*auto simp*: *init-analysis*)

have *hd-M*: ‹− *fst* (*hd analyse*) ∈ *lits-of-l M*›
if
inv-I′: ‹*I′ s*› **and**
s: ‹*s* = (*cach*, *s′*)› ‹*s′* = (*analyse*, *ba*)› **and**
nempty: ‹*analyse* ≠ []›
for *analyse s s′ ba cach*
proof −
have
cach: ‹*conflict-min-analysis-inv M cach ?N D*› **and**
ana: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**

stack-hd: ‹*conflict-min-analysis-stack-hd M ?N D analyse*› **and**
last-analysis: ‹*analyse* $\neq$ [] $\longrightarrow$ *fst* (*last analyse*) = *fst* (*hd init-analysis*)› **and**
b: ‹*analyse* = [] $\longrightarrow$ *ba* $\longrightarrow$ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*›
using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def* **by** *auto*
show *?thesis*
using *stack-hd nempty* **by** (*cases analyse*) *auto*
qed

have *all-removed*: ‹*lit-redundant-inv M ?N D init-analysis*
(*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*), *tl analysis*, *True*)› (**is** *?I*) **and**
all-removed-I': ‹*I'* (*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*), *tl analysis*, *True*)›
(**is** *?I'*) **and**
all-removed-J: ‹*lit-redundant-rec-loop-inv M* (*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*),
tl analysis, *True*)› (**is** *?J*)
if
inv-I': ‹*I' s*› **and** *inv-J*: ‹*lit-redundant-rec-loop-inv M s*›
‹*case s of* (*cach*, *analyse*, *b*) $\Rightarrow$ *analyse* $\neq$ []› **and**
s: ‹*s* = (*cach*, *s'*)›
‹*s'* = (*analysis*, *b*)› **and**
nemtpy-stack: ‹*analysis* $\neq$ []› **and**
finished: ‹*snd* (*hd analysis*) = {#}›
for *s cach s' analysis b*
proof −
obtain *L ana'* **where** *analysis*: ‹*analysis* = (*L*, {#}) # *ana'*›
using *nemtpy-stack finished* **by** (*cases analysis*) *auto*
have
cach: ‹*conflict-min-analysis-inv M cach ?N D*› **and**
ana: ‹*conflict-min-analysis-stack M ?N D analysis*› **and**
stack: ‹*conflict-min-analysis-stack M ?N D analysis*› **and**
stack-hd: ‹*conflict-min-analysis-stack-hd M ?N D analysis*› **and**
last-analysis: ‹*analysis* $\neq$ [] $\longrightarrow$ *fst* (*last analysis*) = *fst* (*hd init-analysis*)› **and**
b: ‹*analysis* = [] $\longrightarrow$ *b* $\longrightarrow$ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*› **and**
dist: ‹*distinct* (*map* (*atm-of o fst*) *analysis*)›
using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def* **by** *auto*
obtain *C* **where**
NU-C: ‹*?N* $\models$*pm add-mset* (−*L*) *C*› **and**
IH: ‹$\bigwedge$*K*. *K* $\in$# *C* $\Longrightarrow$ *?N* $\models$*pm add-mset* (−*K*) (*filter-to-poslev M L D*) $\vee$
K $\in$# *filter-to-poslev M L D*› **and**
index-K: ‹*K*$\in$#*C* $\Longrightarrow$ *index-in-trail M K* < *index-in-trail M L*› **and**
L-M: ‹−*L* $\in$ *lits-of-l M*› **for** *K*
using *stack-hd* **unfolding** *analysis* **by** *auto*

have *NU-D*: ‹*?N* $\models$*pm add-mset* (− *fst* (*hd analysis*)) (*filter-to-poslev M* (*fst* (*hd analysis*)) *D*)›
using *conflict-minimize-intermediate-step-filter-to-poslev*[*OF* - *NU-C*, *simplified*, *OF index-K*]
IH
unfolding *analysis* **by** *auto*
have *ana'*: ‹*conflict-min-analysis-stack M ?N D* (*tl analysis*)›
using *ana* **by** (*auto simp*: *conflict-min-analysis-stack-tl*)
have ‹−*fst* (*hd analysis*) $\in$ *lits-of-l M*›
using *L-M* **by** (*auto simp*: *analysis I'-def s ana*)
then have *cach'*:
‹*conflict-min-analysis-inv M* (*cach*(*atm-of* (*fst* (*hd analysis*)) := *SEEN-REMOVABLE*)) *?N D*›
using *NU-D n-d* **by** (*auto simp*: *conflict-min-analysis-inv-update-removable cach*)
have *stack-hd'*: ‹*conflict-min-analysis-stack-hd M ?N D ana'*›
proof (*cases* ‹*ana'* = []›)
case *True*

then show *?thesis* **by** *auto*
next
case *False*
then obtain *L′ C′ ana′′* **where** *ana′′*: ‹*ana′* = (*L′*, *C′*) # *ana′′*›
by (*cases ana′*; *cases* ‹*hd ana′*›) *auto*
then obtain *E′* **where**
NU-E′: ‹*?N* ⊨*pm add-mset* (− *L′*) *E′*› **and**
‹∀ *K*∈#*E′* − *add-mset L C′*. *?N* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M L′ D*) ∨
K ∈# *filter-to-poslev M L′ D*› **and**
index-C′: ‹∀ *K*∈#*E′*. *index-in-trail M K* < *index-in-trail M L′*› **and**
index-L′-L: ‹*index-in-trail M L* < *index-in-trail M L′*› **and**
C′-E′: ‹*C′* ⊆# *E′*› **and**
uL′-M: ‹− *L′* ∈ *lits-of-l M*›
using *stack* **by** (*auto simp*: *analysis ana′′*)
moreover have ‹*?N* ⊨*pm add-mset* (− *L*) (*filter-to-poslev M L D*)›
using *NU-D analysis* **by** *auto*
moreover have ‹*K* ∈# *E′* − *C′* ⟹ *K* ∈# *E′* − *add-mset L C′* ∨ *K* = *L*› **for** *K*
by (*cases* ‹*L* ∈# *E′*›)
(*fastforce simp*: *minus-notin-trivial dest*!: *multi-member-split*[*of L*]
dest: *in-remove1-msetI*)+
moreover have ‹*K* ∈# *E′* − *C′* ⟹ *index-in-trail M K* ≤ *index-in-trail M L′*› **for** *K*
by (*meson in-diffD index-C′ less-or-eq-imp-le*)
ultimately have ‹*K* ∈# *E′* − *C′* ⟹ *?N* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M L′ D*) ∨
K ∈# *filter-to-poslev M L′ D*› **for** *K*
using *filter-to-poslev-mono-entailement-add-mset*[*of M K L′*]
filter-to-poslev-mono[*of M L L′*]
by *fastforce*
then show *?thesis*
using *NU-E′ uL′-M index-C′ C′-E′* **unfolding** *ana′′* **by** (*auto intro*!: *exI*[*of - E′*])
qed

have ‹*fst* (*hd init-analysis*) = *fst* (*last* (*tl analysis*))› **if** ‹*tl analysis* ≠ []›
using *last-analysis tl-last*[*symmetric*, *OF that*] *that* **unfolding** *ana′* **by** *auto*
then show *?I*
using *ana′ cach′ last-analysis stack-hd′ dist* **unfolding** *lit-redundant-inv-def*
by (*cases ana′*; *auto simp*: *analysis atm-of-eq-atm-of split*: *if-splits*)
then show *I′*: *?I′*
using *inv-I′* **unfolding** *I′-def s* **by** (*auto simp*: *analysis*)
have ‹*distinct* (*map* (λ*x*. − *fst x*) (*tl analysis*))›
using *dist distinct-mapI*[*of* ‹*atm-of o uminus*› ‹*map* (*uminus o fst*) (*tl analysis*)›]
conflict-min-analysis-stack-neg[*OF ana′*] **by** (*auto simp*: *comp-def map-tl*
simp flip: *distinct-mset-image-mset*)
then show *?J*
using *inv-J* **unfolding** *lit-redundant-rec-loop-inv-def prod.case s*
apply (*subst distinct-subseteq-iff*[*symmetric*])
using *conflict-min-analysis-stack-neg*[*OF ana′*] *no-dup-distinct*[*OF n-d*] *dist*
by (*force simp*: *comp-def entails-CNot-negate-ann-lits negate-ann-lits-def*
analysis ana′
simp flip: *distinct-mset-image-mset*)+
qed
have *all-removed-R*:
‹((*cach*(*atm-of* (*fst* (*hd analyse*)) := *SEEN-REMOVABLE*), *tl analyse*, *True*), *s*) ∈ *R*›
if
s: ‹*s* = (*cach*, *s′*)› ‹*s′* = (*analyse*, *b*)› **and**
nempty: ‹*analyse* ≠ []› **and**
finished: ‹*snd* (*hd analyse*) = {#}›

for *s cach s′ analyse b*
using *nempty finished* **unfolding** *R-def s* **by** *auto*
have
seen-removable-inv: ‹*lit-redundant-inv M ?N D init-analysis* (*cach*, *ana*, *False*)› (**is** *?I*) **and**
seen-removable-I′: ‹*I′* (*cach*, *ana*, *False*)› (**is** *?I′*) **and**
seen-removable-R: ‹((*cach*, *ana*, *False*), *s*) ∈ *R*› (**is** *?R*) **and**
seen-removable-J: ‹*lit-redundant-rec-loop-inv M* (*cach*, *ana*, *False*)› (**is** *?J*)
if
inv-I′: ‹*I′ s*› **and** *inv-J*: ‹*lit-redundant-rec-loop-inv M s*› **and**
cond: ‹*case s of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
s: ‹*s* = (*cach*, *s′*)› ‹*s′* = (*analyse*, *b*)› ‹*x* = (*L*, *ana*)› **and**
nemtpy-stack: ‹*analyse* ≠ []› **and**
‹*snd* (*hd analyse*) ≠ {#}› **and**
next-lit: ‹*case x of*
(*L*, *ana*) ⇒ *L* ∈# *snd* (*hd analyse*) ∧ *tl ana* = *tl analyse* ∧ *ana* ≠ [] ∧
hd ana = (*fst* (*hd analyse*), *remove1-mset L* (*snd* (*hd analyse*)))› **and**
lev0-removable: ‹*get-level M L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ∨ *L* ∈# *D*›
for *s cach s′ analyse b x L ana*
proof −
obtain *K C ana′* **where** *analysis*: ‹*analyse* = (*K*, *C*) # *ana′*›
using *nemtpy-stack* **by** (*cases analyse*) *auto*
have *ana′*: ‹*ana* = (*K*, *remove1-mset L C*) # *ana′*› **and** *L-C*: ‹*L* ∈# *C*›
using *next-lit* **unfolding** *s* **by** (*cases ana*; *auto simp*: *analysis*)+
have
cach: ‹*conflict-min-analysis-inv M cach* (*?N*) *D*› **and**
ana: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack: ‹*conflict-min-analysis-stack M ?N D analyse*› **and**
stack-hd: ‹*conflict-min-analysis-stack-hd M ?N D analyse*› **and**
last-analysis: ‹*analyse* ≠ [] ⟶ *fst* (*last analyse*) = *fst* (*hd init-analysis*)› **and**
b: ‹*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*› **and**
dist: ‹*distinct* (*map* (*atm-of* ∘ *fst*) *analyse*)›
using *inv-I′* **unfolding** *lit-redundant-inv-def s I′-def prod.case* **by** *auto*

have *last-analysis′*: ‹*ana* ≠ [] ⟹ *fst* (*hd init-analysis*) = *fst* (*last ana*)›
using *last-analysis next-lit* **unfolding** *analysis s*
by (*cases ana*) (*auto split*: *if-splits*)
have *uL-M*: ‹−*L* ∈ *lits-of-l M*›
using *inv-I′ L-C* **unfolding** *analysis ana s I′-def*
by (*auto dest*!: *multi-member-split*)
have *uK-M*: ‹− *K* ∈ *lits-of-l M*›
using *stack-hd* **unfolding** *analysis* **by** *auto*
consider
(*lev0*) ‹*get-level M L* = *0*› |
(*Removable*) ‹*cach* (*atm-of L*) = *SEEN-REMOVABLE*› |
(*in-D*) ‹*L* ∈# *D*›
using *lev0-removable* **by** *fast*
then have *H*: ‹∃ *CK*. *?N* ⊨*pm add-mset* (− *K*) *CK* ∧
(∀ *Ka*∈#*CK* − *remove1-mset L C*. *?N* ⊨*pm* (*filter-to-poslev M K D*) + {#− *Ka*#} ∨
Ka ∈# *filter-to-poslev M K D*) ∧
(∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*) ∧
remove1-mset L C ⊆# *CK*›
(**is** ‹∃ *C*. *?P C*›)
proof *cases*
case *Removable*
then have *L*: ‹*?N* ⊨*pm add-mset* (− *L*) (*filter-to-poslev M L D*)›
using *cach uL-M* **unfolding** *conflict-min-analysis-inv-def* **by** *auto*

obtain *CK* **where**
⟨*?N* $\models$*pm add-mset* (− *K*) *CK*⟩ **and**
⟨∀ *K'*∈#*CK* − *C*. *?N* $\models$*pm* (*filter-to-poslev M K D*) + {#− *K'*#} ∨ *K'* ∈# *filter-to-poslev M K D*⟩ **and**
index-CK: ⟨∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*⟩ **and**
C-CK: ⟨*C* ⊆# *CK*⟩
using *stack-hd* **unfolding** *analysis* **by** *auto*
moreover have ⟨*remove1-mset L C* ⊆# *CK*⟩
using *C-CK* **by** (*meson diff-subset-eq-self subset-mset.dual-order.trans*)
moreover have ⟨*index-in-trail M L* < *index-in-trail M K*⟩
using *index-CK C-CK L-C* **unfolding** *analysis ana'* **by** *auto*
moreover have *index-CK'*: ⟨∀ *Ka*∈#*CK*. *index-in-trail M Ka* ≤ *index-in-trail M K*⟩
using *index-CK* **by** *auto*
ultimately have ⟨*?P CK*⟩
using *filter-to-poslev-mono-entailement-add-mset*[*of M - -*]
filter-to-poslev-mono[*of M K L*]
using *L L-C C-CK* **by** (*auto simp*: *minus-remove1-mset-if*)
then show *?thesis* **by** *blast*
next
assume *lev0*: ⟨*get-level M L* = *0*⟩
have ⟨*M* $\models$*as CNot* (*?f analyse*)⟩
using *inv-I'* **unfolding** *I'-def s* **by** *auto*
then have ⟨−*L* ∈ *lits-of-l M*⟩
using *next-lit* **unfolding** *analysis s* **by** (*auto dest*: *multi-member-split*)
then have ⟨*?N* $\models$*pm* {#−*L*#}⟩
using *lev0* $cdcl_W$*-restart-mset.literals-of-level0-entailed*[*OF invs*, *of* ⟨−*L*⟩]
by (*auto simp*: *clauses-def ac-simps*)
moreover obtain *CK* **where**
⟨*?N* $\models$*pm add-mset* (− *K*) *CK*⟩ **and**
⟨∀ *K'*∈#*CK* − *C*. *?N* $\models$*pm* (*filter-to-poslev M K D*) + {#− *K'*#} ∨ *K'* ∈# *filter-to-poslev M K D*⟩ **and**
⟨∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*⟩ **and**
C-CK: ⟨*C* ⊆# *CK*⟩
using *stack-hd* **unfolding** *analysis* **by** *auto*
moreover have ⟨*remove1-mset L C* ⊆# *CK*⟩
using *C-CK* **by** (*meson diff-subset-eq-self subset-mset.order-trans*)
ultimately have ⟨*?P CK*⟩
by (*auto simp*: *minus-remove1-mset-if intro*: *conflict-minimize-intermediate-step*)
then show *?thesis* **by** *blast*
next
case *in-D*
obtain *CK* **where**
⟨*?N* $\models$*pm add-mset* (− *K*) *CK*⟩ **and**
⟨∀ *Ka*∈#*CK* − *C*. *?N* $\models$*pm* (*filter-to-poslev M K D*) + {#− *Ka*#} ∨ *Ka* ∈# *filter-to-poslev M K D*⟩ **and**
index-CK: ⟨∀ *Ka*∈#*CK*. *index-in-trail M Ka* < *index-in-trail M K*⟩ **and**
C-CK: ⟨*C* ⊆# *CK*⟩
using *stack-hd* **unfolding** *analysis* **by** *auto*
moreover have ⟨*remove1-mset L C* ⊆# *CK*⟩
using *C-CK* **by** (*meson diff-subset-eq-self subset-mset.order-trans*)
moreover have ⟨*L* ∈# *filter-to-poslev M K D*⟩
using *in-D L-C index-CK C-CK* **by** (*fastforce simp*: *filter-to-poslev-def*)

ultimately have ⟨*?P CK*⟩
using *in-D*
using *filter-to-poslev-mono-entailement-add-mset*[*of M L K*]

by (*auto simp*: *minus-remove1-mset-if dest*!:
intro: *conflict-minimize-intermediate-step*)
then show *?thesis* **by** *blast*
qed note $H = this$
have *stack'*: ⟨*conflict-min-analysis-stack M ?N D ana*⟩
using *stack* **unfolding** *ana' analysis* **by** (*cases ana'*) *auto*
have *stack-hd'*: ⟨*conflict-min-analysis-stack-hd M ?N D ana*⟩
using *H uL-M uK-M* **unfolding** *ana'* **by** *auto*

show *?I*
using *last-analysis' cach stack' stack-hd'* **unfolding** *lit-redundant-inv-def s*
by *auto*
have ⟨$M \models as$ *CNot* (*?f ana*)⟩
using *inv-I'* **unfolding** *I'-def s ana analysis ana'*
by (*cases* ⟨$L \in\#$ C⟩) (*auto dest*!: *multi-member-split*)
then show *?I'*
using *inv-I'* ⟨*?I*⟩ **unfolding** *I'-def s* **by** (*auto simp*: *analysis ana'*)

show *?R*
using *next-lit*
unfolding *R-def s* **by** (*auto simp*: *ana' analysis dest*!: *multi-member-split*
intro: *minimize-conflict-support.intros*)
have ⟨*distinct* (*map* ($\lambda x.\ -$ *fst x*) *ana*)⟩
using *dist distinct-mapI*[*of* ⟨*atm-of o uminus*⟩ ⟨*map* (*uminus o fst*) (*tl analyse*)⟩]
conflict-min-analysis-stack-neg[*OF stack'*] **by** (*auto simp*: *comp-def map-tl*
analysis ana'
simp flip: *distinct-mset-image-mset*)
then show *?J*
using *inv-J* **unfolding** *lit-redundant-rec-loop-inv-def prod.case s*
apply (*subst distinct-subseteq-iff*[*symmetric*])
using *conflict-min-analysis-stack-neg*[*OF stack'*] *no-dup-distinct*[*OF n-d*]
apply (*auto simp*: *comp-def entails-CNot-negate-ann-lits negate-ann-lits-def*
simp flip: *distinct-mset-image-mset*)
apply (*force simp add*: *analysis ana ana'*)
done

qed
have
failed-I: ⟨*lit-redundant-inv M ?N D init-analysis*
(*cach'*, [], *False*)⟩ (**is** *?I*) **and**
failed-I': ⟨*I'* (*cach'*, [], *False*)⟩ (**is** *?I'*) **and**
failed-R: ⟨((*cach'*, [], *False*), *s*) $\in$ *R*⟩ (**is** *?R*) **and**
failed-J: ⟨*lit-redundant-rec-loop-inv M* (*cach'*, [], *False*)⟩ (**is** *?J*)
if
inv-I': ⟨*I' s*⟩ **and** *inv-J*: ⟨*lit-redundant-rec-loop-inv M s*⟩ **and**
cond: ⟨*case s of* (*cach*, *analyse*, *b*) $\Rightarrow$ *analyse* $\neq$ []⟩ **and**
s: ⟨$s = (cach,\ s')$⟩ ⟨$s' = (analyse,\ b)$⟩ **and**
nempty: ⟨*analyse* $\neq$ []⟩ **and**
⟨*snd* (*hd analyse*) $\neq$ {#}⟩ **and**
⟨*case x of* (*L*, *ana*) $\Rightarrow$ $L \in\#$ *snd* (*hd analyse*) $\wedge$ *tl ana* = *tl analyse* $\wedge$
ana $\neq$ [] $\wedge$ *hd ana* = (*fst* (*hd analyse*), *remove1-mset L* (*snd* (*hd analyse*)))⟩ **and**
⟨$x = (L,\ ana)$⟩ **and**
⟨$\neg$ (*get-level M L* = *0* $\vee$ *cach* (*atm-of L*) = *SEEN-REMOVABLE* $\vee$ $L \in\#$ *D*)⟩ **and**
cach-update: ⟨$\forall$ *L*. *cach' L* = *SEEN-REMOVABLE* $\longrightarrow$ *cach L* = *SEEN-REMOVABLE*⟩
for *s cach s' analyse b x L ana E cach'*
proof −

have
cach: ⟨*conflict-min-analysis-inv M cach ?N D*⟩ **and**
ana: ⟨*conflict-min-analysis-stack M ?N D analyse*⟩ **and**
stack: ⟨*conflict-min-analysis-stack M ?N D analyse*⟩ **and**
last-analysis: ⟨*analyse* $\neq$ [] $\longrightarrow$ *fst* (*last analyse*) = *fst* (*hd init-analysis*)⟩ **and**
b: ⟨*analyse* = [] $\longrightarrow$ *b* $\longrightarrow$ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*⟩
using *inv-I'* **unfolding** *lit-redundant-inv-def s I'-def* **by** *auto*
have ⟨*conflict-min-analysis-inv M cach' ?N D*⟩
using *cach cach-update* **by** (*auto simp*: *conflict-min-analysis-inv-def*)
moreover have ⟨*conflict-min-analysis-stack M ?N D* []⟩
by *simp*
ultimately show *?I*
unfolding *lit-redundant-inv-def* **by** *simp*
then show *?I'*
using *M-D* **unfolding** *I'-def* **by** *auto*
show *?R*
using *nempty* **unfolding** *R-def s* **by** *auto*
show *?J*
by (*auto simp*: *lit-redundant-rec-loop-inv-def*)
qed
have *is-propagation-inv*: ⟨*lit-redundant-inv M ?N D init-analysis*
(*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*)⟩ (**is** *?I*) **and**
is-propagation-I': ⟨*I'* (*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*)⟩ (**is** *?I'*) **and**
is-propagation-R: ⟨((*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*), *s*) $\in$ *R*⟩ (**is** *?R*) **and**
is-propagation-dist: ⟨*distinct-mset E'*⟩ (**is** *?dist*) **and**
is-propagation-tauto: ⟨$\neg$*tautology E'*⟩ (**is** *?tauto*) **and**
is-propagation-J': ⟨*lit-redundant-rec-loop-inv M* (*cach*, (*L*, *remove1-mset* (−*L*) *E'*) # *ana*, *False*)⟩ (**is** *?J*)
if
inv-I': ⟨*I' s*⟩ **and** *inv-J*: ⟨*lit-redundant-rec-loop-inv M s*⟩ **and**
⟨*case s of* (*cach*, *analyse*, *b*) $\Rightarrow$ *analyse* $\neq$ []⟩ **and**
s: ⟨*s* = (*cach*, *s'*)⟩ ⟨*s'* = (*analyse*, *b*)⟩ ⟨*x* = (*L*, *ana*)⟩ **and**
nemtpy-stack: ⟨*analyse* $\neq$ []⟩ **and**
⟨*snd* (*hd analyse*) $\neq$ {#}⟩ **and**
next-lit: ⟨*case x of* (*L*, *ana*) $\Rightarrow$
L $\in$# *snd* (*hd analyse*) $\wedge$
tl ana = *tl analyse* $\wedge$
ana $\neq$ [] $\wedge$
hd ana =
(*fst* (*hd analyse*),
remove1-mset L (*snd* (*hd analyse*)))⟩ **and**
⟨$\neg$ (*get-level M L* = *0* $\vee$ *cach* (*atm-of L*) = *SEEN-REMOVABLE* $\vee$ *L* $\in$# *D*)⟩ **and**
E: ⟨*E* $\neq$ *None* $\longrightarrow$ *Propagated* (− *L*) (*the E*) $\in$ *set M*⟩ ⟨*E* = *Some E'*⟩ **and**
st: ⟨*cach* (*atm-of L*) = *SEEN-UNKNOWN*⟩
for *s cach s' analyse b x L ana E E'*
proof −
obtain *K C ana'* **where** *analysis*: ⟨*analyse* = (*K*, *C*) # *ana'*⟩
using *nemtpy-stack* **by** (*cases analyse*) *auto*
have *ana'*: ⟨*ana* = (*K*, *remove1-mset L C*) # *ana'*⟩
using *next-lit* **unfolding** *s* **by** (*cases ana*) (*auto simp*: *analysis*)
have
cach: ⟨*conflict-min-analysis-inv M cach ?N D*⟩ **and**
ana: ⟨*conflict-min-analysis-stack M ?N D analyse*⟩ **and**
stack: ⟨*conflict-min-analysis-stack M ?N D analyse*⟩ **and**
stack-hd: ⟨*conflict-min-analysis-stack-hd M ?N D analyse*⟩ **and**
last-analysis: ⟨*analyse* $\neq$ [] $\longrightarrow$ *fst* (*last analyse*) = *fst* (*hd init-analysis*)⟩ **and**

b: ⟨*analyse* = [] ⟶ *b* ⟶ *cach* (*atm-of* (*fst* (*hd init-analysis*))) = *SEEN-REMOVABLE*⟩ **and**
dist-ana: ⟨*distinct* (*map* (*atm-of* ∘ *fst*) *analyse*)⟩
using *inv-I′* **unfolding** *lit-redundant-inv-def s I′-def* **by** *auto*
have
NU-E: ⟨*?N* ⊨*pm add-mset* (− *L*) (*remove1-mset* (−*L*) *E′*)⟩ **and**
uL-E: ⟨−*L* ∈# *E′*⟩ **and**
M-E′: ⟨*M* ⊨*as CNot* (*remove1-mset* (− *L*) *E′*)⟩ **and**
tauto: ⟨¬ *tautology E′*⟩ **and**
dist: ⟨*distinct-mset E′*⟩ **and**
lev-E′: ⟨*K* ∈# *remove1-mset* (− *L*) *E′* ⟹ *index-in-trail M K* < *index-in-trail M* (− *L*)⟩ **for** *K*
using *Propagated-in-trail-entailed*[*OF invs*, *of* ⟨−*L*⟩ *E′*] *E* **by** (*auto simp*: *ac-simps*)
have *uL-M*: ⟨− *L* ∈ *lits-of-l M*⟩
using *next-lit inv-I′* **unfolding** *s analysis I′-def* **by** (*auto dest*!: *multi-member-split*)
obtain *C′* **where**
⟨*?N* ⊨*pm add-mset* (− *K*) *C′*⟩ **and**
⟨∀ *Ka*∈#*C′*. *index-in-trail M Ka* < *index-in-trail M K*⟩ **and**
⟨*C* ⊆# *C′*⟩ **and**
⟨∀ *Ka*∈#*C′* − *C*.
?N ⊨*pm add-mset* (− *Ka*) (*filter-to-poslev M K D*) ∨
Ka ∈# *filter-to-poslev M K D*⟩ **and**
uK-M: ⟨− *K* ∈ *lits-of-l M*⟩
using *stack-hd*
unfolding *s ana′*[*symmetric*]
by (*auto simp*: *analysis ana′ conflict-min-analysis-stack-change-hd*)

then have *cmas*: ⟨*conflict-min-analysis-stack M ?N D* ((*L*, *remove1-mset* (−*L*) *E′*) # *ana*)⟩
using *stack E next-lit NU-E uL-E uL-M*
filter-to-poslev-mono-entailement-add-mset[*of M* - - ⟨*set-mset ?N*⟩ - *D*]
filter-to-poslev-mono[*of M*] *uK-M*
unfolding *s ana′*[*symmetric*] *prod.case*
by (*auto simp*: *analysis ana′ conflict-min-analysis-stack-change-hd*)
moreover have ⟨*conflict-min-analysis-stack-hd M ?N D* ((*L*, *remove1-mset* (− *L*) *E′*) # *ana*)⟩
using *NU-E lev-E′ uL-M* **by** (*auto intro*!:*exI*[*of* - ⟨*remove1-mset* (− *L*) *E′*⟩])
moreover have ⟨*fst* (*hd init-analysis*) = *fst* (*last* ((*L*, *remove1-mset* (− *L*) *E′*) # *ana*))⟩
using *last-analysis* **unfolding** *analysis ana′* **by** *auto*
ultimately show *?I*
using *cach b* **unfolding** *lit-redundant-inv-def analysis* **by** *auto*
moreover have ⟨*L* ≠ *K*⟩
using *cmas*
unfolding *ana′ conflict-min-analysis-stack.simps*(*3*) **by** *blast*
moreover have ⟨*L* ≠ −*K*⟩
using *cmas*
unfolding *ana′ conflict-min-analysis-stack.simps*(*3*) **by** *auto*
ultimately show *?I′*
using *M-E′ inv-I′ conflict-min-analysis-stack-distinct-fst*[*OF cmas*]
unfolding *I′-def s ana analysis ana′*
by (*auto simp*: *true-annot-CNot-diff atm-of-eq-atm-of uminus-lit-swap*)

have ⟨*L* ∈# *C*⟩ **and** *in-trail*: ⟨*Propagated* (− *L*) (*the E*) ∈ *set M*⟩ **and** *EE′*: ⟨*the E* = *E′*⟩
using *next-lit E* **by** (*auto simp*: *analysis ana′ s*)
then obtain *E″ C′* **where**
E′: ⟨*E′* = *add-mset* (−*L*) *E″*⟩ **and**
C: ⟨*C* = *add-mset L C′*⟩
using *uL-E* **by** (*blast dest*: *multi-member-split*)

have ⟨*minimize-conflict-support M* (*C* + *fold-mset* (+) *D* (*snd* ‘# *mset ana′*))

(*remove1-mset* (− *L*) *E′* + (*remove1-mset L C* + *fold-mset* (+) *D* (*snd* ‘# *mset ana′*)))›
using *minimize-conflict-support.resolve-propa*[*OF in-trail*,
of ‹*C′* + *fold-mset* (+) *D* (*snd* ‘# *mset ana′*)›]
unfolding *C E′ EE′*
by (*auto simp*: *ac-simps*)

then show *?R*
using *nemtpy-stack* **unfolding** *s analysis ana′* **by** (*auto simp*: *R-def*
intro: *resolve-propa*)
have ‹*distinct* (*map* (λ*x*. − *fst x*) *analyse*)›
using *dist-ana distinct-mapI*[*of* ‹*atm-of o uminus*› ‹*map* (*uminus o fst*) *analyse*›]
conflict-min-analysis-stack-neg[*OF cmas*] **unfolding** *analysis ana′*
by (*auto simp*: *comp-def map-tl*
simp flip: *distinct-mset-image-mset*)
then show *?J*
using *inv-J st* **unfolding** *lit-redundant-rec-loop-inv-def prod.case s*
apply (*intro conjI*)
apply (*subst distinct-subseteq-iff*[*symmetric*])
using *conflict-min-analysis-stack-neg*[*OF cmas*] *no-dup-distinct*[*OF n-d*] *uL-M*
‹*L* ≠ −*K*› ‹*L* ≠ *K*› *conflict-min-analysis-stack-distinct-fst*[*OF cmas*]
apply (*auto simp*: *comp-def entails-CNot-negate-ann-lits*
negate-ann-lits-def lits-of-def uminus-lit-swap
simp flip: *distinct-mset-image-mset*)[*3*]
apply (*clarsimp-all simp add*: *analysis ana′*)[]
using *that* **by** (*clarsimp-all simp add*: *analysis ana′*)[]

show *?tauto*
using *tauto* **.**
show *?dist*
using *dist* **.**
qed
have *length-aa-le*: ‹*length aa* ≤ *length M*›
if
‹*I′ s*› **and**
‹*case s of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*s* = (*a*, *b*)› **and**
‹*b* = (*aa*, *ba*)› **and**
‹*aa* ≠ []› **for** *s a b aa ba*
proof −
have ‹*M* ⊨*as CNot* (*fst* ‘# *mset aa*)› **and** ‹*distinct* (*map* (*atm-of o fst*) *aa*)› **and**
‹*distinct* (*map fst aa*)› **and**
‹*conflict-min-analysis-stack M* (*N* + *NE* + *U* + *UE*) *D aa*›
using *distinct-mapI*[*of* ‹*atm-of*› ‹*map fst aa*›]
using *that* **by** (*auto simp*: *I′-def lit-redundant-inv-def*
dest: *conflict-min-analysis-stack-neg*)

then have ‹*set* (*map fst aa*) ⊆ *uminus* ‘ *lits-of-l M*›
by (*auto simp*: *true-annots-true-cls-def-iff-negation-in-model lits-of-def image-image*
uminus-lit-swap
dest!: *multi-member-split*)
from *card-mono*[*OF - this*] **have** ‹*length* (*map fst aa*) ≤ *length M*›
using ‹*distinct* (*map* (*fst*) *aa*)› *distinct-card*[*of* ‹*map fst aa*›] *n-d*
by (*auto simp*: *card-image*[*OF lit-of-inj-on-no-dup*[*OF n-d*]] *lits-of-def image-image*
distinct-card[*OF no-dup-imp-distinct*])
then show ‹*?thesis*› **by** *auto*
qed

show *?thesis*
unfolding *lit-redundant-rec-def lit-redundant-rec-spec-def mark-failed-lits-def get-literal-and-remove-of-analyse-def get-propagation-reason-def*
apply (*refine-vcg WHILEIT-rule-stronger-inv*[**where** $R = R$ **and** $I' = I'$])
— Well-foundness
subgoal by (*rule wf-R*)
subgoal using *assms* **by** (*auto simp*: *lit-redundant-rec-loop-inv-def lits-of-def dest*!: *multi-member-split*)
subgoal by (*rule init-I'*)
subgoal by *auto*
subgoal by (*rule length-aa-le*)
— Assertion:
subgoal by (*rule hd-M*)
— We finished one stage:
subgoal by (*rule all-removed-J*)
subgoal by (*rule all-removed-I'*)
subgoal by (*rule all-removed-R*)
— Assertion:
subgoal for *s cach s' analyse ba*
by (*cases* ‹*analyse*›) (*auto simp*: *I'-def dest*!: *multi-member-split*)

— Cached or level 0:
subgoal by (*rule seen-removable-J*)
subgoal by (*rule seen-removable-I'*)
subgoal by (*rule seen-removable-R*)
— Failed:
subgoal by (*rule failed-J*)
subgoal by (*rule failed-I'*)
subgoal by (*rule failed-R*)
subgoal for *s a b aa ba x ab bb xa* **by** (*cases* ‹*a* (*atm-of ab*)›) *auto*
subgoal by (*rule failed-J*)
subgoal by (*rule failed-I'*)
subgoal by (*rule failed-R*)
— The literal was propagated:
subgoal by (*rule is-propagation-dist*)
subgoal by (*rule is-propagation-tauto*)
subgoal by (*rule is-propagation-J'*)
subgoal by (*rule is-propagation-I'*)
subgoal by (*rule is-propagation-R*)
— End of Loop invariant:
subgoal
using *uL-M* **by** (*auto simp*: *lit-redundant-inv-def conflict-min-analysis-inv-def init-analysis I'-def ac-simps*)
subgoal by (*auto simp*: *lit-redundant-inv-def conflict-min-analysis-inv-def init-analysis I'-def ac-simps*)
done
qed

definition *literal-redundant-spec* **where**
‹*literal-redundant-spec M NU D L* =
SPEC(λ(*cach*, *analysis*, *b*). (*b* $\longrightarrow$ *NU* $\models$*pm add-mset* ($-L$) (*filter-to-poslev M L D*)) $\wedge$
conflict-min-analysis-inv M cach NU D)›

definition *literal-redundant* **where**
‹*literal-redundant M NU D cach L* = *do* {

ASSERT(−*L* ∈ *lits-of-l M*);
if get-level M L = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE*
then RETURN (*cach*, [], *True*)
else if cach (*atm-of L*) = *SEEN-FAILED*
then RETURN (*cach*, [], *False*)
else do {
C ← *get-propagation-reason M* (−*L*);
case C of
Some C ⇒ *do*{
ASSERT(*distinct-mset C* ∧ ¬*tautology C*);
lit-redundant-rec M NU D cach [(*L*, *C* − {#−*L*#})]}
| *None* ⇒ *do* {
RETURN (*cach*, [], *False*)
}
}
}⟩

lemma *true-clss-cls-add-self*: ⟨*NU* ⊨*p D′* + *D′* ⟷ *NU* ⊨*p D′*⟩
by (*metis subset-mset.sup-idem true-clss-cls-sup-iff-add*)

lemma *true-clss-cls-add-add-mset-self*: ⟨*NU* ⊨*p add-mset L* (*D′* + *D′*) ⟷ *NU* ⊨*p add-mset L D′*⟩
using *true-clss-cls-add-self true-clss-cls-mono-r* **by** *fastforce*

lemma *filter-to-poslev-remove1*:
⟨*filter-to-poslev M L* (*remove1-mset K D*) =
(*if index-in-trail M K* ≤ *index-in-trail M L then remove1-mset K* (*filter-to-poslev M L D*)
else filter-to-poslev M L D)⟩
unfolding *filter-to-poslev-def*
by (*auto simp*: *multiset-filter-mono2*)

lemma *filter-to-poslev-add-mset*:
⟨*filter-to-poslev M L* (*add-mset K D*) =
(*if index-in-trail M K* < *index-in-trail M L then add-mset K* (*filter-to-poslev M L D*)
else filter-to-poslev M L D)⟩
unfolding *filter-to-poslev-def*
by (*auto simp*: *multiset-filter-mono2*)

lemma *filter-to-poslev-conflict-min-analysis-inv*:
assumes
L-D: ⟨*L* ∈# *D*⟩ **and**
NU-uLD: ⟨*N*+*U* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M L D*)⟩ **and**
inv: ⟨*conflict-min-analysis-inv M cach* (*N* + *U*) *D*⟩
shows ⟨*conflict-min-analysis-inv M cach* (*N* + *U*) (*remove1-mset L D*)⟩
unfolding *conflict-min-analysis-inv-def*
proof (*intro allI impI*)
fix *K*
assume ⟨−*K* ∈ *lits-of-l M*⟩ **and** ⟨*cach* (*atm-of K*) = *SEEN-REMOVABLE*⟩
then have *K*: ⟨*N* + *U* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M K D*)⟩
using *inv* **unfolding** *conflict-min-analysis-inv-def* **by** *blast*
obtain *D′* **where** *D*: ⟨*D* = *add-mset L D′*⟩
using *multi-member-split*[*OF L-D*] **by** *blast*
have ⟨*N* + *U* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M K D′*)⟩
proof (*cases* ⟨*index-in-trail M L* < *index-in-trail M K*⟩)
case *True*

then have ⟨*N* + *U* ⊨*pm add-mset* (− *K*) (*add-mset L* (*filter-to-poslev M K D′*))⟩
 using *K* **by** (*auto simp*: *filter-to-poslev-add-mset D*)
 then have *1*: ⟨*N* + *U* ⊨*pm add-mset L* (*add-mset* (−*K*) (*filter-to-poslev M K D′*))⟩
 by (*simp add*: *add-mset-commute*)
 have *H*: ⟨*index-in-trail M L* ≤ *index-in-trail M K*⟩
 using *True* **by** *simp*
 have *2*: ⟨*N*+*U* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M K D′*)⟩
 using *filter-to-poslev-mono-entailement-add-mset*[*OF H*] *NU-uLD*
 by (*metis* (*no-types*, *hide-lams*) *D NU-uLD filter-to-poslev-add-mset*
 order-less-irrefl)
 show *?thesis*
 using *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*[*OF 2 1*]
 by (*auto simp*: *true-clss-cls-add-add-mset-self*)
 next
 case *False*
 then show *?thesis* **using** *K* **by** (*auto simp*: *filter-to-poslev-add-mset D split*: *if-splits*)
 qed
 then show ⟨*N* + *U* ⊨*pm add-mset* (− *K*) (*filter-to-poslev M K* (*remove1-mset L D*))⟩
 by (*simp add*: *D*)
qed

lemma *can-filter-to-poslev-can-remove*:
 assumes
 L-D: ⟨*L* ∈# *D*⟩ **and**
 ⟨*M* ⊨*as CNot D*⟩ **and**
 NU-D: ⟨*NU* ⊨*pm D*⟩ **and**
 NU-uLD: ⟨*NU* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M L D*)⟩
 shows ⟨*NU* ⊨*pm remove1-mset L D*⟩
proof −
 obtain *D′* **where**
 D: ⟨*D* = *add-mset L D′*⟩
 using *multi-member-split*[*OF L-D*] **by** *blast*
 then have ⟨*filter-to-poslev M L D* ⊆# *D′*⟩
 by (*auto simp*: *filter-to-poslev-def*)
 then have ⟨*NU* ⊨*pm add-mset* (−*L*) *D′*⟩
 using *NU-uLD true-clss-cls-mono-r*[*of* - ⟨ *add-mset* (− *L*) (*filter-to-poslev M* (−*L*) *D*)⟩]
 by (*auto simp*: *mset-subset-eq-exists-conv*)
 from *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*[*OF this*, *of D′*]
 show ⟨*NU* ⊨*pm remove1-mset L D*⟩
 using *NU-D* **by** (*auto simp*: *D true-clss-cls-add-self*)
qed

lemma *literal-redundant-spec*:
 fixes *L* :: ⟨*′v literal*⟩
 assumes *invs*: ⟨*cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv* (*M*, *N* + *NE*, *U* + *UE*, *D′*)⟩
 assumes
 inv: ⟨*conflict-min-analysis-inv M cach* (*N* + *NE* + *U* + *UE*) *D*⟩ **and**
 L-D: ⟨*L* ∈# *D*⟩ **and**
 M-D: ⟨*M* ⊨*as CNot D*⟩
 shows
 ⟨*literal-redundant M* (*N* + *U*) *D cach L* ≤ *literal-redundant-spec M* (*N* + *U* + *NE* + *UE*) *D L*⟩
proof −
 have *lit-redundant-rec*: ⟨*lit-redundant-rec M* (*N* + *U*) *D cach* [(*L*, *remove1-mset* (− *L*) *E′*)]
 ≤ *literal-redundant-spec M* (*N* + *U* + *NE* + *UE*) *D L*⟩
 if
 E: ⟨*E* ≠ *None* ⟶ *Propagated* (− *L*) (*the E*) ∈ *set M*⟩ **and**

E′: ‹*E* = *Some E′*› **and**
failed: ‹¬ (*get-level M L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE*)›
‹*cach* (*atm-of L*) ≠ *SEEN-FAILED*›
for *E E′*
proof −
have
[*simp*]: ‹−*L* ∈# *E′*› **and**
in-trail: ‹*Propagated* (− *L*) (*add-mset* (−*L*) (*remove1-mset* (−*L*) *E′*)) ∈ *set M*›
using *Propagated-in-trail-entailed*[*OF invs*, *of* ‹−*L*› *E′*] *E E′*
by *auto*
have *H*: ‹*lit-redundant-rec-spec M* (*N* + *U* + *NE* + *UE*) *D L* ≤
literal-redundant-spec M (*N* + *U* + *NE* + *UE*) *D L*›
by (*auto simp*: *lit-redundant-rec-spec-def literal-redundant-spec-def ac-simps*)
show *?thesis*
apply (*rule order.trans*)
apply (*rule lit-redundant-rec-spec*[*OF invs - in-trail*])
subgoal ..
subgoal by (*rule inv*)
subgoal using *assms* **by** *fast*
subgoal by (*rule M-D*)
subgoal using *failed* **by** (*cases* ‹*cach* (*atm-of L*)›) *auto*
subgoal unfolding *literal-redundant-spec-def*[*symmetric*] **by** (*rule H*)
done
qed

have
L-dist: ‹*distinct-mset* (*C*)› **and**
L-tauto: ‹¬*tautology C*›
if
in-trail: ‹*Propagated* (− *L*) *C* ∈ *set M*›
for *C*
using *that*
Propagated-in-trail-entailed[*of M* ‹*N*+*NE*› ‹*U*+*UE*› ‹*D′*› ‹−*L*› ‹*C*›] *invs*
by (*auto simp*:)
have *uL-M*: ‹−*L* ∈ *lits-of-l M*›
using *L-D M-D* **by** (*auto dest*!: *multi-member-split*)
show *?thesis*
unfolding *literal-redundant-def get-propagation-reason-def literal-redundant-spec-def*
apply (*refine-vcg*)
subgoal using *uL-M* **.**
subgoal
using *inv uL-M* $cdcl_W$*-restart-mset.literals-of-level0-entailed*[*OF invs*, *of* ‹−*L*›]
true-clss-cls-mono-r′
by (*fastforce simp*: *mark-failed-lits-def conflict-min-analysis-inv-def*
clauses-def ac-simps)
subgoal using *inv* **by** (*auto simp*: *ac-simps*)
subgoal by *auto*
subgoal using *inv* **by** (*auto simp*: *ac-simps*)
subgoal using *inv* **by** (*auto simp*: *mark-failed-lits-def conflict-min-analysis-inv-def*)
subgoal using *inv* **by** (*auto simp*: *mark-failed-lits-def conflict-min-analysis-inv-def ac-simps*)
subgoal using *L-dist* **by** *simp*
subgoal using *L-tauto* **by** *simp*
subgoal for *E E′*
unfolding *literal-redundant-spec-def*[*symmetric*]
by (*rule lit-redundant-rec*)
done

qed

definition *set-all-to-list* **where**
⟨*set-all-to-list e ys* = *do* {
$S \leftarrow WHILE^{\lambda(i,\ xs).\ i \leq length\ xs \wedge (\forall x \in set\ (take\ i\ xs).\ x = e) \wedge length\ xs = length\ ys}$
$(\lambda(i, xs).\ i < length\ xs)$
$(\lambda(i, xs).\ do$ {
ASSERT($i < length\ xs$);
RETURN($i+1$, $xs[i := e]$)
})
(0, *ys*);
RETURN (*snd S*)
}⟩

lemma
⟨*set-all-to-list e ys* $\leq$ *SPEC*($\lambda xs.\ length\ xs = length\ ys \wedge (\forall x \in set\ xs.\ x = e)$)⟩
unfolding *set-all-to-list-def*
apply (*refine-vcg*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *take-Suc-conv-app-nth list-update-append*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done

definition *get-literal-and-remove-of-analyse-wl*
:: ⟨$'v$ *clause-l* $\Rightarrow$ (*nat* $\times$ *nat* $\times$ *nat* $\times$ *nat*) *list* $\Rightarrow$ $'v$ *literal* $\times$ (*nat* $\times$ *nat* $\times$ *nat* $\times$ *nat*) *list*⟩ **where**
⟨*get-literal-and-remove-of-analyse-wl C analyse* =
(*let* (*i*, *k*, *j*, *ln*) = *last analyse in*
(*C* ! *j*, *analyse*[*length analyse* $-$ *1* := (*i*, *k*, *j* + *1*, *ln*)]))⟩

definition *mark-failed-lits-wl*
where
⟨*mark-failed-lits-wl NU analyse cach* = *SPEC*($\lambda cach'$.
($\forall L.\ cach'\ L = SEEN\text{-}REMOVABLE \longrightarrow cach\ L = SEEN\text{-}REMOVABLE$))⟩

definition *lit-redundant-rec-wl-ref* **where**
⟨*lit-redundant-rec-wl-ref NU analyse* $\longleftrightarrow$
($\forall (i, k, j, ln) \in set\ analyse.\ j \leq ln \wedge i \in\# dom\text{-}m\ NU \wedge i > 0 \wedge$
$ln \leq length\ (NU \propto i) \wedge k < length\ (NU \propto i) \wedge$
distinct $(NU \propto i) \wedge$
$\neg tautology\ (mset\ (NU \propto i))) \wedge$
($\forall (i, k, j, ln) \in set\ (butlast\ analyse).\ j > 0$)⟩

definition *lit-redundant-rec-wl-inv* **where**
⟨*lit-redundant-rec-wl-inv M NU D* = (λ(*cach*, *analyse*, *b*). *lit-redundant-rec-wl-ref NU analyse*)⟩

definition *lit-redundant-reason-stack*
:: ⟨$'v$ *literal* $\Rightarrow$ $'v$ *clauses-l* $\Rightarrow$ *nat* $\Rightarrow$ (*nat* $\times$ *nat* $\times$ *nat* $\times$ *nat*)⟩ **where**
⟨*lit-redundant-reason-stack L NU C'* =

(*if length* $(NU \propto C') > 2$ *then* $(C', 0, 1,$ *length* $(NU \propto C'))$
else if $NU \propto C' ! 0 = L$ *then* $(C', 0, 1,$ *length* $(NU \propto C'))$
else $(C', 1, 0, 1))$›

definition *lit-redundant-rec-wl* :: ‹$('v, nat)$ *ann-lits* ⇒ $'v$ *clauses-l* ⇒ $'v$ *clause* ⇒
  - ⇒ - ⇒ - ⇒
  (- × - × *bool*) *nres*›
where
‹*lit-redundant-rec-wl M NU D cach analysis* - =
    $WHILE_T{}^{lit\text{-}redundant\text{-}rec\text{-}wl\text{-}inv\ M\ NU\ D}$
      $(\lambda(cach, analyse, b).\ analyse \neq [])$
      $(\lambda(cach, analyse, b).$ *do* {
          $ASSERT(analyse \neq [])$;
          $ASSERT(length\ analyse \leq length\ M)$;
    let $(C, k, i, ln) = last\ analyse$;
          $ASSERT(C \in\# dom\text{-}m\ NU)$;
          $ASSERT(length\ (NU \propto C) > k)$;
          $ASSERT(NU \propto C!k \in lits\text{-}of\text{-}l\ M)$;
          let $C = NU \propto C$;
          if $i \geq ln$
          then
            $RETURN(cach\ (atm\text{-}of\ (C\ !\ k) := SEEN\text{-}REMOVABLE), butlast\ analyse, True)$
          else do {
      let $(L, analyse) = get\text{-}literal\text{-}and\text{-}remove\text{-}of\text{-}analyse\text{-}wl\ C\ analyse$;
            $ASSERT(fst(snd(snd\ (last\ analyse))) \neq 0)$;
      $ASSERT(-L \in lits\text{-}of\text{-}l\ M)$;
      $b \leftarrow RES\ (UNIV)$;
      if $(get\text{-}level\ M\ L = 0 \vee cach\ (atm\text{-}of\ L) = SEEN\text{-}REMOVABLE \vee L \in\# D)$
          then $RETURN\ (cach, analyse, False)$
      else if $b \vee cach\ (atm\text{-}of\ L) = SEEN\text{-}FAILED$
      then do {
  $cach \leftarrow mark\text{-}failed\text{-}lits\text{-}wl\ NU\ analyse\ cach$;
  $RETURN\ (cach, [], False)$
      }
      else do {
              $ASSERT(cach\ (atm\text{-}of\ L) = SEEN\text{-}UNKNOWN)$;
  $C' \leftarrow get\text{-}propagation\text{-}reason\ M\ (-L)$;
  case C' *of*
    Some $C' \Rightarrow$ *do* {
      $ASSERT(C' \in\# dom\text{-}m\ NU)$;
      $ASSERT(length\ (NU \propto C') \geq 2)$;
      $ASSERT\ (distinct\ (NU \propto C') \wedge \neg tautology\ (mset\ (NU \propto C')))$;
      $ASSERT(C' > 0)$;
      $RETURN\ (cach, analyse\ @\ [lit\text{-}redundant\text{-}reason\text{-}stack\ (-L)\ NU\ C'], False)$
    }
  | *None* ⇒ *do* {
      $cach \leftarrow mark\text{-}failed\text{-}lits\text{-}wl\ NU\ analyse\ cach$;
      $RETURN\ (cach, [], False)$
  }
      }
          }
      })
      $(cach, analysis, False)$›

fun *convert-analysis-l* **where**
‹*convert-analysis-l NU* $(i, k, j, le) = (-NU \propto i\ !\ k,\ mset\ (Misc.slice\ j\ le\ (NU \propto i)))$›

definition *convert-analysis-list* **where**
‹*convert-analysis-list NU analyse* = *map* (*convert-analysis-l NU*) (*rev analyse*)›

lemma *convert-analysis-list-empty*[*simp*]:
‹*convert-analysis-list NU* [] = []›
‹*convert-analysis-list NU a* = [] ⟷ *a* = []›
by (*auto simp*: *convert-analysis-list-def*)

lemma *trail-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
LaC: ‹*Propagated L C* ∈ *set* (*get-trail-l S*)› **and**
C0: ‹*C* > *0*›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
proof −
have *conv*:
‹(*get-trail-l S*, *get-trail T*) ∈ *convert-lits-l* (*get-clauses-l S*) (*get-unit-clauses-l S*)›
using *ST* **unfolding** *twl-st-l-def* **by** *auto*

have ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-conflicting* ($state_W$*-of T*)› **and**
lev-inv: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv* ($state_W$*-of T*)›
using *struct-invs* **unfolding** *twl-struct-invs-def* $cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv-def*
by *fast*+

have *n-d*: ‹*no-dup* (*get-trail-l S*)›
using *ST lev-inv* **unfolding** $cdcl_W$*-restart-mset*.$cdcl_W$*-M-level-inv-def*
by (*auto simp*: *twl-st-l twl-st*)
have
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
using *list-invs C0 LaC* **by** (*auto simp*: *twl-list-invs-def all-conj-distrib*)
have ‹*twl-st-inv T*›
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
then show *le2*: ‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
using *C ST multi-member-split*[*OF C*] **unfolding** *twl-struct-invs-def*
by (*cases S*; *cases T*)
(*auto simp*: *twl-st-inv.simps twl-st-l-def*
image-Un[*symmetric*])
qed

lemma *clauses-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
proof −
have ‹*twl-st-inv T*›
using *struct-invs* **unfolding** *twl-struct-invs-def* **by** *fast*
then show *le2*: ‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›

using *C ST multi-member-split*[*OF C*] **unfolding** *twl-struct-invs-def*
by (*cases S*; *cases T*)
(*auto simp*: *twl-st-inv.simps twl-st-l-def*
image-Un[*symmetric*])
qed

lemma *lit-redundant-rec-wl*:
fixes *S* :: ‹*nat twl-st-wl*› **and** *S'* :: ‹*nat twl-st-l*› **and** *S''* :: ‹*nat twl-st*› **and** *NU M analyse*
defines
[*simp*]: ‹*S'''* ≡ $state_W$*-of S''*›
defines
‹*M* ≡ *get-trail-wl S*› **and**
M': ‹*M'* ≡ *trail S'''*› **and**
NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
NU': ‹*NU'* ≡ *mset ‘# ran-mf NU*› **and**
‹*analyse'* ≡ *convert-analysis-list NU analyse*›
assumes
S-S': ‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
S'-S'': ‹(*S'*, *S''*) ∈ *twl-st-l None*› **and**
bounds-init: ‹*lit-redundant-rec-wl-ref NU analyse*› **and**
struct-invs: ‹*twl-struct-invs S''*› **and**
add-inv: ‹*twl-list-invs S'*›
shows
‹*lit-redundant-rec-wl M NU D cach analyse lbd* ≤ ⇓
(*Id* $\times_r$ {(*analyse*, *analyse'*). *analyse'* = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse} $\times_r$ *bool-rel*)
(*lit-redundant-rec M' NU' D cach analyse'*)›
(**is** ‹- ≤ ⇓ (- $\times_r$ *?A* $\times_r$ -) -› **is** ‹- ≤ ⇓ *?R* -›)
proof −
obtain *D' NE UE Q W* **where**
S: ‹*S* = (*M*, *NU*, *D'*, *NE*, *UE*, *Q*, *W*)›
using *M-def NU* **by** (*cases S*) *auto*
have *M'-def*: ‹(*M*, *M'*) ∈ *convert-lits-l NU* (*NE* + *UE*)›
using *NU S-S' S'-S''* **unfolding** *M'* **by** (*auto simp*: *S state-wl-l-def twl-st-l-def*)
then have [*simp*]: ‹*lits-of-l M'* = *lits-of-l M*›
by *auto*
have [*simp*]: ‹*fst* (*convert-analysis-l NU x*) = −*NU* ∝ (*fst x*) ! (*fst* (*snd x*))› **for** *x*
by (*cases x*) *auto*
have [*simp*]: ‹*snd* (*convert-analysis-l NU x*) =
mset (*Misc.slice* (*fst* (*snd* (*snd x*))) (*snd* (*snd* (*snd x*))) (*NU* ∝ *fst x*))› **for** *x*
by (*cases x*) *auto*

have
no-smaller-propa: ‹$cdcl_W$*-restart-mset.no-smaller-propa S'''*› **and**
struct-invs: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv S'''*›
using *struct-invs* **unfolding** *twl-struct-invs-def S'''-def*[*symmetric*]
by *fast*+
have *annots*: ‹*set* (*get-all-mark-of-propagated* (*trail S'''*)) ⊆
set-mset ($cdcl_W$*-restart-mset.clauses S'''*)›
using *struct-invs*
unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*
by *fast*
have ‹*no-dup* (*get-trail-wl S*)›
using *struct-invs S-S' S'-S''* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
$cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*

by (*auto simp*: *twl-st-wl twl-st-l twl-st*)
then have *n-d*: ‹*no-dup M*›
by (*auto simp*: *S*)
then have *n-d'*: ‹*no-dup M'*›
using *M'-def* **by** (*auto simp*: *S*)
let *?B* = ‹{(*analyse*, *analyse'*). *analyse'* = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse ∧ *fst* (*snd* (*snd* (*last analyse*))) > *0*}›
have *get-literal-and-remove-of-analyse-wl*:
‹*RETURN* (*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c*)
≤ ⇓ (*Id* $\times_r$ *?B*)
(*get-literal-and-remove-of-analyse x1a*)›
if
xx': ‹(*x*, *x'*) ∈ *?R*› **and**
‹*case x of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*case x' of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*lit-redundant-rec-wl-inv M NU D x*› **and**
s: ‹*x2* = (*x1a*, *x2a*)›
‹*x'* = (*x1*, *x2*)›
‹*x2d* = (*x1f*, *x2e*)›
‹*x2c* = (*x1e*, *x2d*)›
‹(*fst* (*last x1c*), *fst* (*snd* (*last x1c*)), *fst* (*snd* (*snd* (*last x1c*))),
snd (*snd* (*snd* (*last x1c*)))) =
(*x1d*, *x2c*)›
‹*x2b* = (*x1c*, *x2f*)›
‹*x* = (*x1b*, *x2b*)› **and**
‹*x1a* ≠ []› **and**
‹− *fst* (*hd x1a*) ∈ *lits-of-l M'*› **and**
x1c: ‹*x1c* ≠ []› **and**
‹*x1d* ∈# *dom-m NU*› **and**
‹*x1e* < *length* (*NU* ∝ *x1d*)› **and**
‹*NU* ∝ *x1d* ! *x1e* ∈ *lits-of-l M*› **and**
length: ‹¬ *x2e* ≤ *x1f*› **and**
‹*snd* (*hd x1a*) ≠ {#}›
for *x x' x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f*
proof −
have *x1d*: ‹*x1d* = *fst* (*last x1c*)›
using *s* **by** *auto*
have ‹*last x1c* = (*a*, *b*, *c*, *d*) ⟹ *d* ≤ *length* (*NU* ∝ *a*)›
‹*last x1c* = (*a*, *b*, *c*, *d*) ⟹ *c* ≤ *d*› **for** *aa ba list a b c d*
using *xx' x1c length* **unfolding** *s convert-analysis-list-def*
lit-redundant-rec-wl-ref-def
by (*cases x1c rule*: *rev-cases*; *auto*; *fail*)+
then show *?thesis*
supply *convert-analysis-list-def*[*simp*] *hd-rev*[*simp*] *last-map*[*simp*] *rev-map*[*symmetric*, *simp*]
using *x1c xx' length s*
using *Cons-nth-drop-Suc*[*of* ‹*snd* (*snd* (*snd* (*last x1c*)))› ‹*NU* ∝ *fst* (*last x1c*)›, *symmetric*]
unfolding *lit-redundant-rec-wl-ref-def x1d*
by (*cases x1c*; *cases* ‹*last x1c*›)
(*auto simp*: *get-literal-and-remove-of-analyse-wl-def nth-in-sliceI mset-tl*
get-literal-and-remove-of-analyse-def convert-analysis-list-def slice-Suc
slice-head
intro!: *RETURN-SPEC-refine elim*!: *neq-Nil-revE split*: *if-splits*)
qed

have *get-propagation-reason*: ‹*get-propagation-reason M* (− *x1h*)
≤ ⇓ ((‹{(*C'*, *C*). *C* = *mset* (*NU* ∝ *C'*) ∧ *C'* ≠ *0* ∧

Propagated (− *x1g*) (*mset* (*NU*∝*C′*)) ∈ *set M′*
∧ *Propagated* (− *x1g*) *C′* ∈ *set M* ∧ *C′* ∈# *dom-m NU* ∧
length (*NU* ∝ *C′*) ≥ *2*}⟩
option-rel)
(*get-propagation-reason M′* (− *x1g*))⟩
(**is** ‹- ≤ ⇓ (⟨*?get-propagation-reason*⟩*option-rel*) -›)
if
‹(*x*, *x′*) ∈ *?R*› **and**
‹*case x of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*case x′ of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*lit-redundant-rec-wl-inv M NU D x*› **and**
st:
‹*x2* = (*x1a*, *x2a*)›
‹*x′* = (*x1*, *x2*)›
‹*x2d* = (*x1f*, *x2e*)›
‹*x2c* = (*x1e*, *x2d*)›
‹(*fst* (*last x1c*), *fst* (*snd* (*last x1c*)), *fst* (*snd* (*snd* (*last x1c*))),
snd (*snd* (*snd* (*last x1c*)))) =
(*x1d*, *x2c*)›
‹*x2b* = (*x1c*, *x2f*)›
‹*x* = (*x1b*, *x2b*)›
‹*x′a* = (*x1g*, *x2g*)› **and**
‹*x1a* ≠ []› **and**
‹− *fst* (*hd x1a*) ∈ *lits-of-l M′*› **and**
‹*x1c* ≠ []› **and**
x1d: ‹*x1d* ∈# *dom-m NU*› **and**
‹*x1e* < *length* (*NU* ∝ *x1d*)› **and**
‹*NU* ∝ *x1d* ! *x1e* ∈ *lits-of-l M*› **and**
‹¬ *x2e* ≤ *x1f*› **and**
‹*snd* (*hd x1a*) ≠ {#}› **and**
H: ‹(*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c*, *x′a*)
∈ *Id* $\times_f$ *?B*›
‹*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c* = (*x1h*, *x2h*)› **and**
‹− *x1g* ∈ *lits-of-l M′*› **and**
‹− *x1h* ∈ *lits-of-l M*› **and**
‹(*b*, *ba*) ∈ *bool-rel*› **and**
‹*b* ∈ *UNIV*› **and**
‹*ba* ∈ *UNIV*› **and**
‹¬ (*get-level M x1h* = *0* ∨ *x1b* (*atm-of x1h*) = *SEEN-REMOVABLE* ∨ *x1h* ∈# *D*)› **and**
cond: ‹¬ (*get-level M′ x1g* = *0* ∨ *x1* (*atm-of x1g*) = *SEEN-REMOVABLE* ∨ *x1g* ∈# *D*)› **and**
‹¬ (*b* ∨ *x1b* (*atm-of x1h*) = *SEEN-FAILED*)› **and**
‹¬ (*ba* ∨ *x1* (*atm-of x1g*) = *SEEN-FAILED*)›
for *x x′ x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f x′a x1g x2g x1h x2h b ba*
proof −
have [*simp*]: ‹*x1h* = *x1g*›
using *st H* **by** *auto*
have *le2*: ‹*length* (*NU* ∝ *x1d*) ≥ *2*›
using *clauses-length-ge2*[*OF S′-S″ add-inv assms*(*10*), *of x1d*] *x1d st S-S′*
by (*auto simp*: *S*)
have
‹*Propagated* (− *x1g*) (*mset* (*NU* ∝ *a*)) ∈ *set M′*› (**is** *?propa*) **and**
‹*a* ≠ *0*› (**is** *?a*) **and**
‹*a* ∈# *dom-m NU*› (**is** *?L*) **and**
‹*length* (*NU* ∝ *a*) ≥ *2*› (**is** *?len*)
if *x1e-M*: ‹*Propagated* (− *x1g*) *a* ∈ *set M*›

for *a*
proof −
have [*simp*]: ‹*a* ≠ *0*›
proof
assume [*simp*]: ‹*a* = *0*›
obtain E' **where**
x1d-M': ‹*Propagated* (− *x1g*) E' ∈ *set* M'› **and**
‹E' ∈# *NE* + *UE*›
using *x1e-M M'-def* **by** (*auto dest*: *split-list simp*: *convert-lits-l-def p2rel-def convert-lit.simps*
elim!: *list-rel-in-find-correspondanceE split*: *if-splits*)
moreover have ‹*unit-clss* S'' = *NE* + *UE*›
using *S-S' S'-S'' x1d-M'* **by** (*auto simp*: *S*)
moreover have ‹*Propagated* (− *x1g*) E' ∈ *set* (*get-trail* S'')›
using *S-S' S'-S'' x1d-M'* **by** (*auto simp*: *S state-wl-l-def twl-st-l-def M'*)
moreover have ‹*0* < *count-decided* (*get-trail* S'')›
using *cond S-S' S'-S'' count-decided-ge-get-level*[*of M* ‹*x1g*›]
by (*auto simp*: *S M' twl-st*)
ultimately show *False*
using *clauses-in-unit-clss-have-level0*(*1*)[*of* S'' E' ‹− *x1g*›] *cond* ‹*twl-struct-invs* S''›
S-S' S'-S'' M'-def
by (*auto simp*: *S*)
qed
show *?propa* **and** *?a*
using *that M'-def* **by** (*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*
elim!: *list-rel-in-find-correspondanceE split*: *if-splits*)
then show *?L*
using *that add-inv S-S' S'-S'' S* **unfolding** *twl-list-invs-def*
by (*auto 5 5 simp*: *state-wl-l-def twl-st-l-def*)
show *?len*
using *trail-length-ge2*[*OF S'-S'' add-inv assms*(*10*), *of* ‹− *x1g*› *a*] *that S-S'*
by (*force simp*: *S*)
qed
then show *?thesis*
apply (*auto simp*: *get-propagation-reason-def refine-rel-defs intro*!: *RES-refine*)
apply (*case-tac s*)
by *auto*
qed
have *resolve*: ‹((*x1b*, *x2h* @ [*lit-redundant-reason-stack* (− *x1h*) *NU xb*], *False*), *x1*,
(*x1g*, *remove1-mset* (− *x1g*) *x'c*) # *x2g*, *False*)
∈ *Id* $\times_f$
({(*analyse*, *analyse'*).
analyse' = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse} $\times_f$
bool-rel)›
if
xx': ‹(*x*, *x'*) ∈ *Id* $\times_r$ *?A* $\times_r$ *bool-rel*› **and**
‹*case x of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*case x' of* (*cach*, *analyse*, *b*) ⇒ *analyse* ≠ []› **and**
‹*lit-redundant-rec-wl-inv M NU D x*› **and**
s:
‹*x2* = (*x1a*, *x2a*)›
‹*x'* = (*x1*, *x2*)›
‹*x2d* = (*x1f*, *x2e*)›
‹*x2c* = (*x1e*, *x2d*)›
‹(*fst* (*last x1c*), *fst* (*snd* (*last x1c*)), *fst* (*snd* (*snd* (*last x1c*))),

snd (*snd* (*snd* (*last x1c*)))) =
(*x1d*, *x2c*)⟩
⟨*x2b* = (*x1c*, *x2f*)⟩
⟨*x* = (*x1b*, *x2b*)⟩
⟨*x'a* = (*x1g*, *x2g*)⟩ **and**
[*simp*]: ⟨*x1a* ≠ []⟩ **and**
⟨− *fst* (*hd x1a*) ∈ *lits-of-l M'*⟩ **and**
[*simp*]: ⟨*x1c* ≠ []⟩ **and**
⟨*x1d* ∈# *dom-m NU*⟩ **and**
⟨*x1e* < *length* (*NU* ∝ *x1d*)⟩ **and**
⟨*NU* ∝ *x1d* ! *x1e* ∈ *lits-of-l M*⟩ **and**
⟨¬ *x2e* ≤ *x1f*⟩ **and**
⟨*snd* (*hd x1a*) ≠ {#}⟩ **and**
get-literal-and-remove-of-analyse-wl:
⟨(*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c*, *x'a*)
∈ *Id* $\times_f$
{(*analyse*, *analyse'*).
analyse' = *convert-analysis-list NU analyse* ∧
lit-redundant-rec-wl-ref NU analyse ∧
0 < *fst* (*snd* (*snd* (*last analyse*)))}⟩ **and**
get-lit: ⟨*get-literal-and-remove-of-analyse-wl* (*NU* ∝ *x1d*) *x1c* = (*x1h*, *x2h*)⟩ **and**
⟨− *x1g* ∈ *lits-of-l M'*⟩ **and**
⟨*fst* (*snd* (*snd* (*last x2h*))) ≠ *0*⟩ **and**
⟨− *x1h* ∈ *lits-of-l M*⟩ **and**
bba: ⟨(*b*, *ba*) ∈ *bool-rel*⟩ **and**
⟨¬ (*get-level M x1h* = *0* ∨ *x1b* (*atm-of x1h*) = *SEEN-REMOVABLE* ∨ *x1h* ∈# *D*)⟩ **and**
⟨¬ (*get-level M' x1g* = *0* ∨ *x1* (*atm-of x1g*) = *SEEN-REMOVABLE* ∨ *x1g* ∈# *D*)⟩ **and**
⟨¬ (*b* ∨ *x1b* (*atm-of x1h*) = *SEEN-FAILED*)⟩ **and**
⟨¬ (*ba* ∨ *x1* (*atm-of x1g*) = *SEEN-FAILED*)⟩ **and**
xb-x'c: ⟨(*xa*, *x'b*)
∈ ⟨*?get-propagation-reason x1g*⟩*option-rel*⟩ **and**
xa: ⟨*xa* = *Some xb*⟩ ⟨*x'b* = *Some x'c*⟩ **and**
⟨(*xb*, *x'c*)
∈ (*?get-propagation-reason x1g*)⟩ **and**
dist-tauto: ⟨*distinct-mset x'c* ∧ ¬ *tautology x'c*⟩ **and**
⟨*xb* ∈# *dom-m NU*⟩ **and**
⟨*2* ≤ *length* (*NU* ∝ *xb*)⟩
for *x x' x1 x2 x1a x2a x1b x2b x1c x1d x2c x1e x2d x1f x2e x2f x'a x1g x2g x1h*
x2h b ba xa x'b xb x'c
proof −
have [*simp*]: ⟨*mset* (*tl C*) = *remove1-mset* (*C*!*0*) (*mset C*)⟩ **for** *C*
by (*cases C*) *auto*
have [*simp*]:
⟨*x2* = (*x1a*, *x2a*)⟩
⟨*x'* = (*x1*, *x1a*, *x2a*)⟩
⟨*x2d* = (*x1f*, *x2e*)⟩
⟨*x2c* = (*x1e*, *x1f*, *x2e*)⟩
⟨*last x1c* = (*x1d*, *x1e*, *x1f*, *x2e*)⟩
⟨*x2b* = (*x1c*, *x2f*)⟩
⟨*x* = (*x1b*, *x1c*, *x2f*)⟩
⟨*xa* = *Some xb*⟩
⟨*x'b* = *Some x'c*⟩
⟨*x'c* = *mset* (*NU* ∝ *xb*)⟩
using *s get-literal-and-remove-of-analyse-wl xa xb-x'c*
unfolding *get-lit convert-analysis-list-def*
by *auto*

then have *x1d0*: ‹*length* (*NU* ∝ *xb*) > *2* ⟹ *x1g* = −*NU* ∝ *xb* ! *0*› ‹*NU* ∝ *xb* ≠ []› **and**
x1d: ‹−*x1g* ∈ *set* (*watched-l* (*NU* ∝ *xb*))›
using *add-inv xb-x′c S-S′ S′-S′′ S* **unfolding** *twl-list-invs-def*
by (*auto 5 5 simp*: *state-wl-l-def twl-st-l-def*)

have *le2*: ‹*length* (*NU* ∝ *xb*) ≥ *2*›
using *clauses-length-ge2*[*OF S′-S′′ add-inv assms*(*10*)] *xb-x′c S-S′*
by (*auto simp*: *S*)
have *0*: ‹*case lit-redundant-reason-stack* (−*x1g*) *NU xb of* (*i*, *k*, *j*, *ln*) ⇒
j ≤ *ln* ∧ *i* ∈# *dom-m NU* ∧ *0* ≤ *j* ∧ *0* < *i* ∧ *ln* ≤ *length* (*NU* ∝ *i*) ∧
k < *length* (*NU* ∝ *i*) ∧ *distinct* (*NU* ∝ *i*) ∧ ¬ *tautology* (*mset* (*NU* ∝ *i*))›
for *i j ln k*
using *s xx′ get-literal-and-remove-of-analyse-wl xb-x′c x1d le2 dist-tauto*
unfolding *get-lit convert-analysis-list-def lit-redundant-rec-wl-ref-def*
lit-redundant-reason-stack-def
by (*auto split*: *if-splits*)

have ‹(*x1g*, *remove1-mset* (− *x1g*) (*mset* (*NU* ∝ *xb*))) =
convert-analysis-l NU (*lit-redundant-reason-stack* (− *x1g*) *NU xb*)›
using *s xx′ get-literal-and-remove-of-analyse-wl xb-x′c x1d le2*
unfolding *get-lit convert-analysis-list-def lit-redundant-rec-wl-ref-def*
lit-redundant-reason-stack-def
by (*auto split*: *simp*: *Misc.slice-def drop-Suc simp*: *x1d0*(*1*)
dest!: *list-decomp-2*)
then show *?thesis*
using *s xx′ get-literal-and-remove-of-analyse-wl xb-x′c x1d 0*
unfolding *get-lit convert-analysis-list-def lit-redundant-rec-wl-ref-def*
by (*cases x2h rule*: *rev-cases*)
(*auto simp*: *drop-Suc uminus-lit-swap butlast-append*
dest: *list-decomp-2*)
qed
have *mark-failed-lits-wl*: ‹*mark-failed-lits-wl NU x2e x1b* ≤ ⇓ *Id* (*mark-failed-lits NU′ x2d x1*)›
if
‹(*x*, *x′*) ∈ *?R*› **and**
‹*x′* = (*x1*, *x2*)› **and**
‹*x* = (*x1b*, *x2b*)›
for *x x′ x2e x1b x1 x2 x2b x2d*
using *that* **unfolding** *mark-failed-lits-wl-def mark-failed-lits-def* **by** *auto*

have *ana*: ‹*last analyse* = (*fst* (*last analyse*), *fst* (*snd* (*last analyse*)),
fst (*snd* (*snd* (*last analyse*))), *snd* (*snd* (*snd* (*last analyse*))))› **for** *analyse*
by (*cases* ‹*last analyse*›) *auto*

show *?thesis*
supply *convert-analysis-list-def*[*simp*] *hd-rev*[*simp*] *last-map*[*simp*] *rev-map*[*symmetric*, *simp*]
unfolding *lit-redundant-rec-wl-def lit-redundant-rec-def*
apply (*rewrite at* ‹*let* - = - ∝ - *in* -› *Let-def*)
apply (*rewrite* **in** ‹*let* - = - *in* -› *ana*)
apply (*rewrite at* ‹*let* - = (-, -, -) *in* -› *Let-def*)
apply *refine-rcg*
subgoal using *bounds-init* **unfolding** *analyse′-def* **by** *auto*
subgoal for *x x′*
by (*cases x*, *cases x′*)
(*auto simp*: *lit-redundant-rec-wl-inv-def lit-redundant-rec-wl-ref-def*)
subgoal by *auto*
subgoal by *auto*

subgoal using M'*-def* **by** (*auto dest*: *convert-lits-l-imp-same-length*)
subgoal by (*auto simp*: *lit-redundant-rec-wl-inv-def lit-redundant-rec-wl-ref-def elim*!: *neq-Nil-revE*)
subgoal by (*auto simp*: *lit-redundant-rec-wl-inv-def lit-redundant-rec-wl-ref-def elim*!: *neq-Nil-revE*)
subgoal by (*auto simp*: *map-butlast rev-butlast-is-tl-rev lit-redundant-rec-wl-ref-def dest*: *in-set-butlastD*)
subgoal by (*auto simp*: *map-butlast rev-butlast-is-tl-rev lit-redundant-rec-wl-ref-def Misc.slice-def*
dest: *in-set-butlastD*
elim!: *neq-Nil-revE*)
subgoal by (*auto simp*: *map-butlast rev-butlast-is-tl-rev lit-redundant-rec-wl-ref-def Misc.slice-def*
dest: *in-set-butlastD*
elim!: *neq-Nil-revE*)
apply (*rule get-literal-and-remove-of-analyse-wl*; *assumption*)
subgoal by *auto*
subgoal by *auto*
subgoal using M'*-def* **by** *auto*
subgoal by *auto*
subgoal by *auto*
apply (*rule mark-failed-lits-wl*; *assumption*)
subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
subgoal by *auto*
apply (*rule get-propagation-reason*; *assumption*)
apply *assumption*
apply (*rule mark-failed-lits-wl*; *assumption*)
subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
subgoal for $x\ x'\ x1\ x2\ x1a\ x2a\ x1b\ x2b\ x1c\ x1d\ x2c\ x1e\ x2d\ x1f\ x2e\ x2f\ x'a\ x1g\ x2g\ x1h\ x2h\ b\ ba\ xa\ x'b\ xb\ x'c$
by (*rule resolve*)
done
qed

definition *literal-redundant-wl* **where**
⟨*literal-redundant-wl M NU D cach L lbd* = *do* {
ASSERT($-L \in$ *lits-of-l M*);
if get-level M L = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE*
then RETURN (*cach*, [], *True*)
else if cach (*atm-of L*) = *SEEN-FAILED*
then RETURN (*cach*, [], *False*)
else do {
C ← *get-propagation-reason M* ($-L$);
case C of
Some C ⇒ *do*{
ASSERT(*C* ∈# *dom-m NU*);
ASSERT(*length* (*NU* ∝ *C*) ≥ *2*);
ASSERT(*distinct* (*NU* ∝ *C*) ∧ ¬*tautology* (*mset* (*NU* ∝ *C*)));
lit-redundant-rec-wl M NU D cach [*lit-redundant-reason-stack* ($-L$) *NU C*] *lbd*
}

| *None* ⇒ *do* {
    RETURN (*cach*, [], *False*)
  }
 }
}›

lemma *literal-redundant-wl-literal-redundant*:
  fixes *S* :: ‹*nat twl-st-wl*› **and** *S'* :: ‹*nat twl-st-l*› **and** *S''* :: ‹*nat twl-st*› **and** *NU M*
  defines
    [*simp*]: ‹*S'''* ≡ $state_W$*-of S''*›
  defines
    ‹*M* ≡ *get-trail-wl S*› **and**
    M': ‹*M'* ≡ *trail S'''*› **and**
    NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
    NU': ‹*NU'* ≡ *mset '# ran-mf NU*›
  assumes
    S-S': ‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
    S'-S'': ‹(*S'*, *S''*) ∈ *twl-st-l None*› **and**
    ‹*M* ≡ *get-trail-wl S*› **and**
    M': ‹*M'* ≡ *trail S'''*› **and**
    NU: ‹*NU* ≡ *get-clauses-wl S*› **and**
    NU': ‹*NU'* ≡ *mset '# ran-mf NU*›
  assumes
    struct-invs: ‹*twl-struct-invs S''*› **and**
    add-inv: ‹*twl-list-invs S'*› **and**
    L-D: ‹*L* ∈# *D*› **and**
    M-D: ‹*M* ⊨*as CNot D*›
  shows
    ‹*literal-redundant-wl M NU D cach L lbd* ≤ ⇓
      (*Id* $\times_r$ {(*analyse*, *analyse'*). *analyse'* = *convert-analysis-list NU analyse* ∧
        lit-redundant-rec-wl-ref NU analyse} $\times_r$ *bool-rel*)
      (*literal-redundant M' NU' D cach L*)›
   (**is** ‹- ≤ ⇓ (- $\times_r$ *?A* $\times_r$ -) -› **is** ‹- ≤ ⇓ *?R* -›)
proof −
  obtain *D' NE UE Q W* **where**
    S: ‹*S* = (*M*, *NU*, *D'*, *NE*, *UE*, *Q*, *W*)›
    using *M-def NU* **by** (*cases S*) *auto*
  have *M'-def*: ‹(*M*, *M'*) ∈ *convert-lits-l NU* (*NE*+*UE*)›
    using *NU S-S' S'-S'' S M'* **by** (*auto simp*: *twl-st-l-def state-wl-l-def*)
  have [*simp*]: ‹*lits-of-l M'* = *lits-of-l M*›
    using *M'-def* **by** *auto*
  have
    no-smaller-propa: ‹$cdcl_W$*-restart-mset.no-smaller-propa S'''*› **and**
    struct-invs': ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv S'''*›
    using *struct-invs* **unfolding** *twl-struct-invs-def S'''-def*[*symmetric*]
    by *fast*+
  have *annots*: ‹*set* (*get-all-mark-of-propagated* (*trail S'''*)) ⊆
     set-mset ($cdcl_W$*-restart-mset.clauses S'''*)›
    using *struct-invs'*
    unfolding $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
      $cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clause-alt-def*
    by *fast*
  have *n-d*: ‹*no-dup* (*get-trail-wl S*)›
    using *struct-invs' S-S' S'-S''* **unfolding** $cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv-def*
     $cdcl_W$*-restart-mset.*$cdcl_W$*-M-level-inv-def*
    by (*auto simp*: *twl-st-wl twl-st-l twl-st*)

then have *n-d*: ‹*no-dup M*›
  by (*auto simp*: *S*)
then have *n-d'*: ‹*no-dup M'*›
  using *M'-def* **by** (*auto simp*: *S*)
have *uL-M*: ‹$-L \in$ *lits-of-l M*›
  using *L-D M-D* **by** (*auto dest*!: *multi-member-split*)
have *H*: ‹*lit-redundant-rec-wl M NU D cach analyse lbd*
    $\leq \Downarrow$ *?R* (*lit-redundant-rec M' NU' D cach analyse'*)›
  if ‹*analyse'* = *convert-analysis-list NU analyse*› **and**
    ‹*lit-redundant-rec-wl-ref NU analyse*›
   for *analyse analyse'*
  using *lit-redundant-rec-wl*[*of S S' S'' analyse D cach lbd*, *unfolded S'''-def*[*symmetric*],
   unfolded
   M-def[*symmetric*] *M'*[*symmetric*] *NU*[*symmetric*] *NU'*[*symmetric*],
   OF S-S' S'-S'' - *struct-invs add-inv*]
  that **by** (*auto simp*:)
have *get-propagation-reason*: ‹*get-propagation-reason M* ($-L$)
  $\leq \Downarrow$ (⟨{(C', C). C = *mset* (*NU* $\propto$ C') $\wedge$ $C' \neq 0$ $\wedge$ *Propagated* ($-L$) (*mset* (*NU*$\propto$$C'$)) $\in$ *set M'*
        $\wedge$ *Propagated* ($-L$) $C' \in$ *set M* $\wedge$ *length* (*NU*$\propto$$C'$) $\geq$ *2*}⟩
      option-rel)
    (*get-propagation-reason M'* ($-L$))›
  (**is** ‹- $\leq \Downarrow$ (⟨*?get-propagation-reason*⟩*option-rel*) -› **is** *?G1*) **and**
  propagated-L:
   ‹*Propagated* ($-L$) *a* $\in$ *set M* $\Longrightarrow$ *a* $\neq$ *0* $\wedge$ *Propagated* ($-$ *L*) (*mset* (*NU* $\propto$ *a*)) $\in$ *set M'*›
   (**is** ‹*?H2* $\Longrightarrow$ *?G2*›)
  if
   lev0-rem: ‹$\neg$ (*get-level M' L* = *0* $\vee$ *cach* (*atm-of L*) = *SEEN-REMOVABLE*)› **and**
   ux1e-M: ‹$-$ *L* $\in$ *lits-of-l M*›
  for *a*
  proof $-$
   have ‹*Propagated* ($-$ *L*) (*mset* (*NU* $\propto$ *a*)) $\in$ *set M'*› (**is** *?propa*) **and**
    ‹*a* $\neq$ *0*› (**is** *?a*) **and**
‹*length* (*NU* $\propto$ *a*) $\geq$ *2*› (**is** *?len*)
    if *L-M*: ‹*Propagated* ($-L$) *a* $\in$ *set M*›
    for *a*
   proof $-$
    have [*simp*]: ‹*a* $\neq$ *0*›
    proof
     assume [*simp*]: ‹*a* = *0*›
     obtain *E'* **where**
      x1d-M': ‹*Propagated* ($-$ *L*) *E'* $\in$ *set M'*› **and**
      ‹*E'* $\in\#$ *NE* + *UE*›
      using *L-M M'-def* **by** (*auto dest*: *split-list simp*: *convert-lits-l-def p2rel-def*
        convert-lit.simps
        elim!: *list-rel-in-find-correspondanceE split*: *if-splits*)
     moreover have ‹*unit-clss S''* = *NE* + *UE*›
      using *S-S' S'-S'' x1d-M'* **by** (*auto simp*: *S*)
     moreover have ‹*Propagated* ($-$ *L*) *E'* $\in$ *set* (*get-trail S''*)›
      using *S-S' S'-S'' x1d-M'* **by** (*auto simp*: *S state-wl-l-def twl-st-l-def M'*)
     moreover have ‹*0* < *count-decided* (*get-trail S''*)›
      using *lev0-rem S-S' S'-S'' count-decided-ge-get-level*[*of M L*]
      by (*auto simp*: *S M' twl-st*)
     ultimately show *False*
      using *clauses-in-unit-clss-have-level0*(*1*)[*of S'' E'* ‹$-$ *L*›] *lev0-rem* ‹*twl-struct-invs S''*›
       S-S' S'-S'' M'-def
      by (*auto simp*: *S*)

qed

show *?propa* **and** *?a*
using *that M′-def* **by** (*auto simp*: *convert-lits-l-def p2rel-def convert-lit.simps*
elim!: *list-rel-in-find-correspondanceE split*: *if-splits*)
show *?len*
using *trail-length-ge2*[*OF S′-S″ add-inv struct-invs*, *of* ‹− *L*› *a*] *that S-S′*
by (*force simp*: *S*)
qed note $H = this$
show ‹*?H2* ⟹ *?G2*›
using *H* **by** *auto*
show *?G1*
using *H*
apply (*auto simp*: *get-propagation-reason-def refine-rel-defs*
get-propagation-reason-def intro!: *RES-refine*)
apply (*case-tac s*)
by *auto*
qed
have *S‴*: ‹*S‴* = (*get-trail S″*, *get-all-init-clss S″*, *get-all-learned-clss S″*,
get-conflict S″)›
by (*cases S″*) (*auto simp*: *S‴-def*)
have [*simp*]: ‹*mset* (*tl C*) = *remove1-mset* (*C*!*0*) (*mset C*)› **for** *C*
by (*cases C*) *auto*
have *S″-M′*: ‹(*get-trail S″*) = *M′*›
using *M′ S‴* **by** *auto*

have [*simp*]: ‹*length* (*NU* ∝ *C*) > *2* ⟹ *NU* ∝ *C* ! *0* = −*L*› **and**
L-watched: ‹−*L* ∈ *set* (*watched-l* (*NU* ∝ *C*))› **and**
L-dist: ‹*distinct* (*NU* ∝ *C*)› **and**
L-tauto: ‹¬*tautology* (*mset* (*NU* ∝ *C*))›
if
in-trail: ‹*Propagated* (− *L*) *C* ∈ *set M*› **and**
lev: ‹¬ (*get-level M′ L* = *0* ∨ *cach* (*atm-of L*) = *SEEN-REMOVABLE*)›
for *C*
using *add-inv that propagated-L*[*OF lev - in-trail*] *uL-M S-S′ S′-S″*
Propagated-in-trail-entailed[*of* ‹*get-trail S″*› ‹*get-all-init-clss S″*› ‹*get-all-learned-clss S″*›
‹*get-conflict S″*› ‹−*L*› ‹*mset* (*NU* ∝ *C*)›] *struct-invs′* **unfolding** *S‴*[*symmetric*]
by (*auto simp*: *S twl-list-invs-def S″-M′*; *fail*)+

have [*dest*]: ‹*C* ≠ {#}› **if** ‹*Propagated* (− *L*) *C* ∈ *set M′*› **for** *C*
proof −
have ‹*a* @ *Propagated L mark* # *b* = *trail S‴* ⟹ *b* ⊨*as CNot* (*remove1-mset L mark*) ∧ *L* ∈#
mark›
for *L mark a b*
using *struct-invs′* **unfolding** *cdcl*$_W$*-restart-mset.cdcl*$_W$*-all-struct-inv-def*
cdcl$_W$*-restart-mset.cdcl*$_W$*-conflicting-def*
by *fast*
then show *?thesis*
using *that S-S′ S′-S″ M′-def M′*
by (*fastforce simp*: *S state-wl-l-def*
twl-st-l-def convert-lits-l-def convert-lit.simps
list-rel-append2 list-rel-append1
elim!: *list-relE3 list-relE4*
elim: *list-rel-in-find-correspondanceE split*: *if-splits*
dest!: *split-list p2relD*)
qed

have *le2*: ‹*Propagated* $(- L)$ *C* $\in$ *set M* $\Longrightarrow$ $C > 0$ $\Longrightarrow$ *length* $(NU \propto C) \geq 2$› **for** *C*
  using *trail-length-ge2*[*OF S′-S″ add-inv struct-invs*, *of - C*] *S-S′*
  by (*auto simp*: *S*)
have [*simp*]: ‹*Propagated* $(- L)$ *C* $\in$ *set M* $\Longrightarrow$ $C > 0$ $\Longrightarrow$ *C* $\in\#$ *dom-m NU*› **for** *C*
  using *add-inv S-S′ S′-S″ propagated-L*[*of C*]
  by (*auto simp*: *S twl-list-invs-def state-wl-l-def*
    twl-st-l-def)
show *?thesis*
  unfolding *literal-redundant-wl-def literal-redundant-def*
  apply (*refine-rcg H get-propagation-reason*)
  subgoal by *simp*
  subgoal using *M′-def* **by** *simp*
  subgoal using *M′-def* **by** (*auto simp*: *lit-redundant-rec-wl-ref-def*)
  subgoal by *simp*
  subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
  apply (*assumption*)
  subgoal by (*auto simp*: *lit-redundant-rec-wl-ref-def*)
  subgoal by *simp*
  subgoal by *simp*
  subgoal for *x x′ C x′a*
    using *le2*[*of C*] *L-watched*[*of C*] *L-dist*[*of C*] *L-tauto*[*of C*]
    by (*auto simp*: *convert-analysis-list-def drop-Suc slice-0*
      lit-redundant-reason-stack-def slice-Suc slice-head slice-end
      dest!: *list-decomp-2*)
  subgoal for *x x′ C x′a*
    using *le2*[*of C*] *L-watched*[*of C*] *L-dist*[*of C*] *L-tauto*[*of C*]
    by (*auto simp*: *convert-analysis-list-def drop-Suc slice-0*
      lit-redundant-reason-stack-def slice-Suc slice-head slice-end
      dest!: *list-decomp-2*)
  subgoal for *x x′ C x′a*
    using *le2*[*of C*] *L-watched*[*of C*] *L-dist*[*of C*] *L-tauto*[*of C*]
    by (*auto simp*: *convert-analysis-list-def drop-Suc slice-0*
      lit-redundant-reason-stack-def slice-Suc slice-head slice-end
      dest!: *list-decomp-2*)
  subgoal for *x x′ C x′a*
    using *le2*[*of C*] *L-watched*[*of C*] *L-dist*[*of C*] *L-tauto*[*of C*]
    by (*auto simp*: *lit-redundant-reason-stack-def lit-redundant-rec-wl-ref-def*)
  done
qed

definition *mark-failed-lits-stack-inv* **where**
‹*mark-failed-lits-stack-inv NU analyse* = (λ*cach*.
  ($\forall$ (*i*, *k*, *j*, *len*) $\in$ *set analyse*. $j \leq len \wedge len \leq$ *length* $(NU \propto i) \wedge i \in\#$ *dom-m NU* $\wedge$
    $k <$ *length* $(NU \propto i) \wedge j > 0$))›

We mark all the literals from the current literal stack as failed, since every minimisation call will find the same minimisation problem.

definition *mark-failed-lits-stack* **where**
‹*mark-failed-lits-stack* $\mathcal{A}_{in}$ *NU analyse cach* = *do* {
  (-, *cach*) $\leftarrow$ $WHILE_T{}^{\lambda(\text{-},\ cach).\ mark\text{-}failed\text{-}lits\text{-}stack\text{-}inv\ NU\ analyse\ cach}$
    (λ(*i*, *cach*). *i* < *length analyse*)
    (λ(*i*, *cach*). *do* {
      ASSERT(*i* < *length analyse*);
      let (*cls-idx*, -, *idx*, -) = *analyse* ! *i*;
      ASSERT(*atm-of* $(NU \propto$ *cls-idx* ! $(idx - 1)) \in\# \mathcal{A}_{in}$);

RETURN (*i*+*1*, *cach* (*atm-of* (*NU* ∝ *cls-idx* ! (*idx* − *1*)) := *SEEN-FAILED*))
})
(*0*, *cach*);
RETURN cach
}›

lemma *mark-failed-lits-stack-mark-failed-lits-wl*:
shows
‹(*uncurry2* (*mark-failed-lits-stack* $\mathcal{A}$), *uncurry2 mark-failed-lits-wl*) ∈
[λ((*NU*, *analyse*), *cach*). *literals-are-in-*$\mathcal{L}_{in}$*-mm* $\mathcal{A}$ (*mset* '# *ran-mf NU*) ∧
mark-failed-lits-stack-inv NU analyse cach]$_f$
Id $\times_f$ *Id* $\times_f$ *Id* → ⟨*Id*⟩*nres-rel*›
proof −
have ‹*mark-failed-lits-stack* $\mathcal{A}$ *NU analyse cach* ≤ (*mark-failed-lits-wl NU analyse cach*)›
if
NU-$\mathcal{L}_{in}$: ‹*literals-are-in-*$\mathcal{L}_{in}$*-mm* $\mathcal{A}$ (*mset* '# *ran-mf NU*)› **and**
init: ‹*mark-failed-lits-stack-inv NU analyse cach*›
for *NU analyse cach*
proof −
define *I* **where**
‹*I* = (λ(*i* :: *nat*, *cach'*). (∀ *L*. *cach' L* = *SEEN-REMOVABLE* ⟶ *cach L* = *SEEN-REMOVABLE*))›
have *valid-atm*: ‹*atm-of* (*NU* ∝ *cls-idx* ! (*idx* − *1*)) ∈# $\mathcal{A}$›
if
‹*I s*› **and**
‹*case s of* (*i*, *cach*) ⇒ *i* < *length analyse*› **and**
‹*case s of* (*i*, *cach*) ⇒ *mark-failed-lits-stack-inv NU analyse cach*› **and**
‹*s* = (*i*, *cach*)› **and**
i: ‹*i* < *length analyse*› **and**
‹*analyse* ! *i* = (*cls-idx*, *k*)› ‹*k* = (*k0*, *k'*)› ‹*k'* = (*idx*, *len*)›
for *s i cach cls-idx idx k len k' k'' k0*
proof −
have [*iff*]: ‹(∀ *a b*. (*a*, *b*) ∉ *set analyse*) ⟷ *False*›
using *i* **by** (*cases analyse*) *auto*
show *?thesis*
unfolding *in-*$\mathcal{L}_{all}$*-atm-of-in-atms-of-iff*[*symmetric*] *atms-of-*$\mathcal{L}_{all}$*-*$\mathcal{A}_{in}$[*symmetric*]
apply (*rule literals-are-in-*$\mathcal{L}_{in}$*-mm-in-*$\mathcal{L}_{all}$)
using *NU-*$\mathcal{L}_{in}$ *that nth-mem*[*of i analyse*]
by (*auto simp*: *mark-failed-lits-stack-inv-def I-def*)
qed
show *?thesis*
unfolding *mark-failed-lits-stack-def mark-failed-lits-wl-def*
apply (*refine-vcg WHILEIT-rule-stronger-inv*[**where** *R* = ‹*measure* (λ(*i*, -). *length analyse* −*i*)›
and *I'* = *I*])
subgoal by *auto*
subgoal using *init* **by** *simp*
subgoal unfolding *I-def* **by** *auto*
subgoal by *auto*
subgoal for *s i cach cls-idx idx*
by (*rule valid-atm*)
subgoal unfolding *mark-failed-lits-stack-inv-def* **by** *auto*
subgoal unfolding *I-def* **by** *auto*
subgoal by *auto*
subgoal unfolding *I-def* **by** *auto*
done
qed
then show *?thesis*

by (*intro frefI nres-relI*) *auto*
qed

end