Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

January 20, 2020

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 ⟨*proof*⟩

end

lemma *nat-shiftr*[*simp*]:
m >> *0* = *m*
‹((*0*::*nat*) >> *m*) = *0*›
‹(*m* >> *Suc n*) = (*m div 2* >> *n*)› **for** *m* :: *nat*
⟨*proof*⟩

lemma *nat-shifl-div*: ‹*m* >> *n* = *m div* (*2*^*n*)› **for** *m* :: *nat*
  ⟨*proof*⟩

lemma *nat-shiftl*[*simp*]:
  m << *0* = *m*
  ‹((*0*::*nat*) << *m*) = *0*›
  ‹(*m* << *Suc n*) = ((*m* * *2*) << *n*)› **for** *m* :: *nat*
  ⟨*proof*⟩

lemma *nat-shiftr-div2*: ‹*m* >> *1* = *m div 2*› **for** *m* :: *nat*
  ⟨*proof*⟩

lemma *nat-shiftr-div*: ‹*m* << *n* = *m* * (*2*^*n*)› **for** *m* :: *nat*
  ⟨*proof*⟩

definition *shiftl1* :: ‹*nat* ⇒ *nat*› **where**
  ‹*shiftl1 n* = *n* << *1*›

definition *shiftr1* :: ‹*nat* ⇒ *nat*› **where**
  ‹*shiftr1 n* = *n* >> *1*›

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

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

lift-definition *test-bit-natural* :: ‹*natural* ⇒ *nat* ⇒ *bool*› **is** *test-bit* ⟨*proof*⟩

lift-definition *lsb-natural* :: ‹*natural* ⇒ *bool*› **is** *lsb* ⟨*proof*⟩

lift-definition *set-bit-natural* :: *natural* ⇒ *nat* ⇒ *bool* ⇒ *natural* **is**
  set-bit ⟨*proof*⟩

lift-definition *set-bits-natural* :: ‹(*nat* ⇒ *bool*) ⇒ *natural*›
  is ‹*set-bits* :: (*nat* ⇒ *bool*) ⇒ *nat*› ⟨*proof*⟩

lift-definition *shiftl-natural* :: ‹*natural* ⇒ *nat* ⇒ *natural*›
  is ‹*shiftl* :: *nat* ⇒ *nat* ⇒ *nat*› ⟨*proof*⟩

lift-definition *shiftr-natural* :: ‹*natural* ⇒ *nat* ⇒ *natural*›
  is ‹*shiftr* :: *nat* ⇒ *nat* ⇒ *nat*› ⟨*proof*⟩

lift-definition *bitNOT-natural* :: ‹*natural* ⇒ *natural*›
  is ‹*bitNOT* :: *nat* ⇒ *nat*› ⟨*proof*⟩

lift-definition *bitAND-natural* :: ‹*natural* ⇒ *natural* ⇒ *natural*›
  is ‹*bitAND* :: *nat* ⇒ *nat* ⇒ *nat*› ⟨*proof*⟩

lift-definition *bitOR-natural* :: ‹*natural* ⇒ *natural* ⇒ *natural*›
  is ‹*bitOR* :: *nat* ⇒ *nat* ⇒ *nat*› ⟨*proof*⟩

lift-definition *bitXOR-natural* :: ‹*natural* ⇒ *natural* ⇒ *natural*›
  is ‹*bitXOR* :: *nat* ⇒ *nat* ⇒ *nat*› ⟨*proof*⟩

end

instance ⟨*proof*⟩
end

context includes *natural.lifting* **begin**
lemma [*code*]:
  integer-of-natural (*m* >> *n*) = (*integer-of-natural m*) >> *n*
  ⟨*proof*⟩

lemma [*code*]:
  integer-of-natural (*m* << *n*) = (*integer-of-natural m*) << *n*
  ⟨*proof*⟩

end

lemma *bitXOR-1-if-mod-2*: ‹*bitXOR L 1* = (*if L mod 2* = *0 then L* + *1 else L* − *1*)› **for** *L* :: *nat*
  ⟨*proof*⟩

lemma *bitAND-1-mod-2*: ‹*bitAND L 1* = *L mod 2*› **for** *L* :: *nat*
  ⟨*proof*⟩

lemma *shiftl-0-uint32*[*simp*]: ‹*n* << *0* = *n*› **for** *n* :: *uint32*
  ⟨*proof*⟩

lemma *shiftl-Suc-uint32*: ‹*n* << *Suc m* = (*n* << *m*) << *1*› **for** *n* :: *uint32*
  ⟨*proof*⟩

lemma *nat-set-bit-0*: ‹*set-bit x 0 b* = *nat* ((*bin-rest* (*int x*)) *BIT b*)› **for** *x* :: *nat*
  ⟨*proof*⟩

lemma *nat-test-bit0-iff*: ‹*n* !! *0* ⟷ *n mod 2* = *1*› **for** *n* :: *nat*
⟨*proof*⟩
lemma *test-bit-2*: ‹*m* > *0* ⟹ (*2*∗*n*) !! *m* ⟷ *n* !! (*m* − *1*)› **for** *n* :: *nat*
  ⟨*proof*⟩

lemma *test-bit-Suc-2*: ‹*m* > *0* ⟹ *Suc* (*2* ∗ *n*) !! *m* ⟷ (*2* ∗ *n*) !! *m*› **for** *n* :: *nat*
  ⟨*proof*⟩

lemma *bin-rest-prev-eq*:
  assumes [*simp*]: ‹*m* > *0*›
  shows ‹*nat* ((*bin-rest* (*int w*))) !! (*m* − *Suc* (*0*::*nat*)) = *w* !! *m*›
⟨*proof*⟩

lemma *bin-sc-ge0*: ‹*w* >= *0* ==> (*0*::*int*) ≤ *bin-sc n b w*›
  ⟨*proof*⟩

lemma *bin-to-bl-eq-nat*:
  ‹*bin-to-bl* (*size a*) (*int a*) = *bin-to-bl* (*size b*) (*int b*) ==> *a*=*b*›
  ⟨*proof*⟩

lemma *nat-bin-nth-bl*: *n* < *m* ⟹ *w* !! *n* = *nth* (*rev* (*bin-to-bl m* (*int w*))) *n* **for** *w* :: *nat*
  ⟨*proof*⟩

lemma *bin-nth-ge-size*: ‹*nat na* ≤ *n* ⟹ *0* ≤ *na* ⟹ *bin-nth na n* = *False*›

⟨*proof*⟩

lemma *test-bit-nat-outside*: $n > size\ w \Longrightarrow \neg w\ !!\ n$ **for** $w :: nat$
⟨*proof*⟩

lemma *nat-bin-nth-bl'*:
‹$a\ !!\ n \longleftrightarrow (n < size\ a \wedge (rev\ (bin\text{-}to\text{-}bl\ (size\ a)\ (int\ a))\ !\ n))$›
⟨*proof*⟩

lemma *nat-set-bit-test-bit*: ‹$set\text{-}bit\ w\ n\ x\ !!\ m = (if\ m = n\ then\ x\ else\ w\ !!\ m)$› **for** $w\ n :: nat$
⟨*proof*⟩

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* :: $('c \Rightarrow bool) \Rightarrow ('a \times 'c)\ set \Rightarrow ('b \times 'd)\ set$
$\Rightarrow (('a \Rightarrow 'b) \times ('c \Rightarrow 'd))\ set$
$([\text{-}]_f\ \text{-} \rightarrow \text{-}\ [0,60,60]\ 60)$
where $[P]_f\ R \rightarrow S \equiv \{(f,g).\ \forall x\ y.\ P\ y \wedge (x,y)\in R \longrightarrow (f\ x,\ g\ y)\in S\}$

abbreviation *freft* $(\text{-} \rightarrow_f \text{-}\ [60,60]\ 60)$ **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 $(f,g)\in fref\ P\ R\ S$
⟨*proof*⟩
lemma *fref-mono*: $[\![\ \bigwedge x.\ P'\ x \Longrightarrow P\ x;\ R' \subseteq R;\ S \subseteq S'\ ]\!]$
$\Longrightarrow fref\ P\ R\ S \subseteq fref\ P'\ R'\ S'$
⟨*proof*⟩

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

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 ‹$length\ xs \leq CARD('a)$›
⟨*proof*⟩

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*: ‹$(\forall (x,\ y) \in A.\ P\ x\ y) \longleftrightarrow (\forall x\ y.\ (x,\ y) \in A \longrightarrow P\ x\ y)$›
  ⟨*proof*⟩

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

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**
    ‹$V \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \land P\ T1 \land Q'\ T1\ T0\}\ U$› **and**
    ‹$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ V$›
  shows ‹$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \land P\ T1\}\ V$›
  ⟨*proof*⟩

lemma *Down-del-assumption-middle*:
  assumes
    ‹$S1 \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \land P\ T1 \land Q'\ T1\ T0\}\ S0$›
  shows ‹$S1 \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \land Q'\ T1\ T0\}\ S0$›

⟨proof⟩

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 ‹$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$›
⟨proof⟩

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 ‹$W \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1 \wedge P\ T1\}\ V$›
⟨proof⟩

lemma *Down-del-assumption-beginning*:
fixes U :: ‹*'a nres*› **and** V :: ‹*'b nres*› **and** $Q\ Q'$ :: ‹$'b \Rightarrow 'a \Rightarrow bool$›
assumes
‹$V \le \Downarrow \{(T1,\ T0).\ Q\ T1\ T0 \wedge Q'\ T1\ T0\}\ U$›
shows ‹$V \le \Downarrow \{(T1,\ T0).\ Q'\ T1\ T0\}\ U$›
⟨proof⟩

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: ‹$S1 \le \Downarrow R10\ S0$› **and**
J[*thin*]: ‹$S2 \le \Downarrow R21\ S1$›
for *S1*:: ‹*'b nres*› **and** *S0* :: ‹*'a nres*› **and** *S2* :: ‹*'c nres*› **and** *R10 R21* ⇒
‹*insert True-implies-equals*[*where P* = ‹$S2 \le \Downarrow R21\ S1$›, *symmetric*,
THEN equal-elim-rule1, *OF J*]›
| *I*[*thin*]: ‹$S1 \le \Downarrow \{(T1,\ T0).\ P\ T1\}\ S0$› (*multi*) **and**
J[*thin*]: - **for** *S1*:: ‹*'b nres*› **and** *S0* :: ‹*'a nres*› **and** *P* :: ‹$'b \Rightarrow bool$› ⇒
‹*match J*[*uncurry*] *in*
J[*curry*]: ‹$- \Longrightarrow S2 \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ S1$› *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*]: ‹$S1 \le \Downarrow \{(T1,\ T0).\ P\ T1 \wedge Q'\ T1\ T0\}\ S0$› (*multi*) **and**
J[*thin*]: - **for** *S1*:: ‹*'b nres*› **and** *S0* :: ‹*'a nres*› **and** *Q'* **and** *P* :: ‹$'b \Rightarrow bool$› ⇒
‹*match J*[*uncurry*] *in*
J[*curry*]: ‹$- \Longrightarrow S2 \le \Downarrow \{(T2,\ T1).\ R\ T2\ T1\}\ S1$› *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›
⟨*proof*⟩

Inversion Tactics

lemma *refinement-trans-long*:
‹*A* = *A′* ⟹ *B* = *B′* ⟹ *R* ⊆ *R′* ⟹ *A* ≤ ⇓ *R B* ⟹ *A′* ≤ ⇓ *R′ B′*›
⟨*proof*⟩

lemma *mem-set-trans*:
‹*A* ⊆ *B* ⟹ *a* ∈ *A* ⟹ *a* ∈ *B*›
⟨*proof*⟩

lemma *fun-rel-syn-invert*:
‹*a* = *a′* ⟹ *b* ⊆ *b′* ⟹ *a* → *b* ⊆ *a′* → *b′*›
⟨*proof*⟩

lemma *fref-param1*: *R*→*S* = *fref* (λ-. *True*) *R S*
⟨*proof*⟩

lemma *fref-syn-invert*:
‹*a* = *a′* ⟹ *b* ⊆ *b′* ⟹ *a* $\rightarrow_f$ *b* ⊆ *a′* $\rightarrow_f$ *b′*›
⟨*proof*⟩

lemma *nres-rel-mono*:
‹*a* ⊆ *a′* ⟹ ⟨*a*⟩ *nres-rel* ⊆ ⟨*a′*⟩ *nres-rel*›
⟨*proof*⟩

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

method *match-fun-rel* =
((*match* **conclusion in**
‹- → - ⊆ - → -› ⇒ ‹*rule fun-rel-mono*›
| ‹- $\rightarrow_f$ - ⊆ - $\rightarrow_f$ -› ⇒ ‹*rule fref-syn-invert*›

| ‹⟨-⟩*nres-rel* ⊆ ⟨-⟩*nres-rel*› ⇒ ‹*rule nres-rel-mono*›
| ‹$[-]_f$ - → - ⊆ $[-]_f$ - → -› ⇒ ‹*rule fref-mono*›
)+)

lemma *weaken-SPEC2*: ‹$m' \leq SPEC\ \Phi \Longrightarrow m = m' \Longrightarrow (\bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x) \Longrightarrow m \leq SPEC\ \Psi$›
⟨*proof*⟩

method *match-spec-trans* =
(*match* **conclusion in** ‹$f \leq SPEC\ R$› **for** *f* :: ‹'*a nres*› **and** *R* :: ‹'*a* ⇒ *bool*› ⇒
‹*print-term f*; *match premises in I*: ‹- ⟹ - ⟹ $f' \leq SPEC\ R'$› *for f'* :: ‹'*a nres*› *and R'* :: ‹'*a* ⇒ *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* $g \equiv \lambda x.\ f$ *oooo* $(g\ x)$
abbreviation *comp6* (**infixl** *oooooo 55*) **where** f *oooooo* $g \equiv \lambda x.\ f$ *ooooo* $(g\ x)$
abbreviation *comp7* (**infixl** *ooooooo 55*) **where** f *ooooooo* $g \equiv \lambda x.\ f$ *oooooo* $(g\ x)$
abbreviation *comp8* (**infixl** *oooooooo 55*) **where** f *oooooooo* $g \equiv \lambda x.\ f$ *ooooooo* $(g\ x)$
abbreviation *comp9* (**infixl** *ooooooooo 55*) **where** f *ooooooooo* $g \equiv \lambda x.\ f$ *oooooooo* $(g\ x)$
abbreviation *comp10* (**infixl** *oooooooooo 55*) **where** f *oooooooooo* $g \equiv \lambda x.\ f$ *ooooooooo* $(g\ x)$
abbreviation *comp11* (**infixl** o_{11} *55*) **where** $f\ o_{11}\ g \equiv \lambda x.\ f$ *oooooooooo* $(g\ x)$
abbreviation *comp12* (**infixl** o_{12} *55*) **where** $f\ o_{12}\ g \equiv \lambda x.\ f\ o_{11}\ (g\ x)$
abbreviation *comp13* (**infixl** o_{13} *55*) **where** $f\ o_{13}\ g \equiv \lambda x.\ f\ o_{12}\ (g\ x)$
abbreviation *comp14* (**infixl** o_{14} *55*) **where** $f\ o_{14}\ g \equiv \lambda x.\ f\ o_{13}\ (g\ x)$
abbreviation *comp15* (**infixl** o_{15} *55*) **where** $f\ o_{15}\ g \equiv \lambda x.\ f\ o_{14}\ (g\ x)$
abbreviation *comp16* (**infixl** o_{16} *55*) **where** $f\ o_{16}\ g \equiv \lambda x.\ f\ o_{15}\ (g\ x)$
abbreviation *comp17* (**infixl** o_{17} *55*) **where** $f\ o_{17}\ g \equiv \lambda x.\ f\ o_{16}\ (g\ x)$
abbreviation *comp18* (**infixl** o_{18} *55*) **where** $f\ o_{18}\ g \equiv \lambda x.\ f\ o_{17}\ (g\ x)$
abbreviation *comp19* (**infixl** o_{19} *55*) **where** $f\ o_{19}\ g \equiv \lambda x.\ f\ o_{18}\ (g\ x)$
abbreviation *comp20* (**infixl** o_{20} *55*) **where** $f\ o_{20}\ g \equiv \lambda x.\ f\ o_{19}\ (g\ x)$

notation
comp4 (**infixl** ∘∘∘ *55*) **and**
comp5 (**infixl** ∘∘∘∘ *55*) **and**
comp6 (**infixl** ∘∘∘∘∘ *55*) **and**
comp7 (**infixl** ∘∘∘∘∘∘ *55*) **and**
comp8 (**infixl** ∘∘∘∘∘∘∘ *55*) **and**
comp9 (**infixl** ∘∘∘∘∘∘∘∘ *55*) **and**
comp10 (**infixl** ∘∘∘∘∘∘∘∘∘ *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*: ‹$P \Longrightarrow A \leq SPEC\ Q \Longrightarrow A \leq SPEC(\lambda x.\ Q\ x \wedge P)$›
⟨*proof*⟩

lemma *bind-refine-spec*: ‹$(\bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq \Downarrow R\ M) \Longrightarrow M' \leq SPEC\ \Phi \Longrightarrow M' \gg= f \leq \Downarrow R\ M$›
⟨*proof*⟩

lemma *intro-spec-iff*:
‹$(RES\ X \gg= f \leq M) = (\forall x{\in}X.\ f\ x \leq M)$›
⟨*proof*⟩

lemma *case-prod-bind*:
assumes ‹$\bigwedge x1\ x2.\ x = (x1,\ x2) \Longrightarrow f\ x1\ x2 \leq \Downarrow R\ I$›
shows ‹$(case\ x\ of\ (x1,\ x2) \Rightarrow f\ x1\ x2) \leq \Downarrow R\ I$›
⟨*proof*⟩

lemma (**in** *transfer*) *transfer-bool*[*refine-transfer*]:
assumes $\alpha\ fa \leq Fa$
assumes $\alpha\ fb \leq Fb$

shows α (*case-bool fa fb x*) $\leq$ *case-bool Fa Fb x*
⟨*proof*⟩

lemma *ref-two-step'*: ‹$A \leq B \Longrightarrow \Downarrow R\ A \leq\ \Downarrow R\ B$›
⟨*proof*⟩

lemma *RES-RETURN-RES*: ‹$RES\ \Phi \gg= (\lambda T.\ RETURN\ (f\ T)) = RES\ (f\ `\ \Phi)$›
⟨*proof*⟩

lemma *RES-RES-RETURN-RES*: ‹$RES\ A \gg= (\lambda T.\ RES\ (f\ T)) = RES\ (\bigcup (f\ `\ A))$›
⟨*proof*⟩

lemma *RES-RES2-RETURN-RES*: ‹$RES\ A \gg= (\lambda(T,\ T').\ RES\ (f\ T\ T')) = RES\ (\bigcup (uncurry\ f\ `\ A))$›
⟨*proof*⟩

lemma *RES-RES3-RETURN-RES*:
‹$RES\ A \gg= (\lambda(T,\ T',\ T'').\ RES\ (f\ T\ T'\ T'')) = RES\ (\bigcup ((\lambda(a,\ b,\ c).\ f\ a\ b\ c)\ `\ A))$›
⟨*proof*⟩

lemma *RES-RETURN-RES3*:
‹$SPEC\ \Phi \gg= (\lambda(T,\ T',\ T'').\ RETURN\ (f\ T\ T'\ T'')) = RES\ ((\lambda(a,\ b,\ c).\ f\ a\ b\ c)\ `\ \{T.\ \Phi\ T\})$›
⟨*proof*⟩

lemma *RES-RES-RETURN-RES2*: ‹$RES\ A \gg= (\lambda(T,\ T').\ RETURN\ (f\ T\ T')) = RES\ (uncurry\ f\ `$
$A)$›
⟨*proof*⟩

lemma *bind-refine-res*: ‹$(\bigwedge x.\ x \in \Phi \Longrightarrow f\ x \leq \Downarrow R\ M) \Longrightarrow M' \leq RES\ \Phi \Longrightarrow M' \gg= f \leq \Downarrow R\ M$›
⟨*proof*⟩

lemma *RES-RETURN-RES-RES2*:
‹$RES\ \Phi \gg= (\lambda(T,\ T').\ RETURN\ (f\ T\ T')) = RES\ (uncurry\ f\ `\ \Phi)$›
⟨*proof*⟩

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*:
‹REC_T (*WHILEI-body* ($\gg=$) *RETURN* $I'\ b'\ f$) x' =
(REC_T (*WHILEI-body* ($\gg=$) *RETURN* ($\lambda x'.\ I'\ x' \wedge (b'\ x' \longrightarrow f\ x' = FAIL \vee f\ x' \leq SPEC\ I')$) b'
f) x')›
(**is** ‹REC_T *?f* x' = REC_T *?f'* x'›)
⟨*proof*⟩

lemma *WHILEIT-add-post-condition*:
‹(*WHILEIT* $I'\ b'\ f'\ x'$) =
(*WHILEIT* ($\lambda x'.\ I'\ x' \wedge (b'\ x' \longrightarrow f'\ x' = FAIL \vee f'\ x' \leq SPEC\ I')$)
$b'\ f'\ x'$)›
⟨*proof*⟩

lemma *WHILEIT-rule-stronger-inv*:
assumes
‹*wf R*› **and**
‹$I\ s$› **and**
‹$I'\ s$› **and**

$\langle \bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \le SPEC\ (\lambda s'.\ I\ s' \wedge\ \ I'\ s' \wedge (s',\ s) \in R)\rangle$ **and**
$\langle \bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow \neg\ b\ s \Longrightarrow \Phi\ s\rangle$
shows $\langle WHILE_T{}^I\ b\ f\ s \le SPEC\ \Phi\rangle$
$\langle proof\rangle$

lemma *RES-RETURN-RES2*:
$\langle SPEC\ \Phi \gg= (\lambda(T,\ T').\ RETURN\ (f\ T\ T')) = RES\ (uncurry\ f\ `\ \{T.\ \Phi\ T\})\rangle$
$\langle proof\rangle$

lemma *WHILEIT-rule-stronger-inv-RES*:
assumes
$\langle wf\ R\rangle$ **and**
$\langle I\ s\rangle$ **and**
$\langle I'\ s\rangle$
$\langle \bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow b\ s \Longrightarrow f\ s \le SPEC\ (\lambda s'.\ I\ s' \wedge\ \ I'\ s' \wedge (s',\ s) \in R)\rangle$ **and**
$\langle \bigwedge s.\ I\ s \Longrightarrow I'\ s \Longrightarrow \neg\ b\ s \Longrightarrow s \in \Phi\rangle$
shows $\langle WHILE_T{}^I\ b\ f\ s \le RES\ \Phi\rangle$
$\langle proof\rangle$

lemma *fref-weaken-pre-weaken*:
assumes $\bigwedge x.\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in fref\ P'\ R\ S$
assumes $\langle S \subseteq S'\rangle$
shows $(f,h) \in fref\ P\ R\ S'$
$\langle proof\rangle$

lemma *bind-rule-complete-RES*: $\langle (M \gg= f \le RES\ \Phi) = (M \le SPEC\ (\lambda x.\ f\ x \le RES\ \Phi))\rangle$
$\langle proof\rangle$

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

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

lemma *fref-to-Down-curry*:
$\langle (uncurry\ f,\ uncurry\ g) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow f\ x\ y \le \Downarrow B\ (g\ x'\ y'))\rangle$
$\langle proof\rangle$

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

lemma *fref-to-Down-curry2'*:

‹$(\mathit{uncurry2}\ f,\ \mathit{uncurry2}\ g) \in A \rightarrow_f \langle B\rangle \mathit{nres\text{-}rel} \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'\ z\ z'.\ (((x,\ y),\ z),\ ((x',\ y'),\ z')) \in A \Longrightarrow$
$f\ x\ y\ z \le \Downarrow B\ (g\ x'\ y'\ z'))$›
$\langle \mathit{proof} \rangle$

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

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

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

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

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

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

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

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

lemma *fref-to-Down-curry-no-nres*:

‹$(uncurry\ (RETURN\ oo\ f),\ uncurry\ (RETURN\ oo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ P\ (x',\ y') \Longrightarrow ((x,\ y),\ (x',\ y')) \in A \Longrightarrow (f\ x\ y,\ g\ x'\ y') \in B)$›
⟨*proof*⟩

lemma *RES-RETURN-RES4*:
‹$SPEC\ \Phi \gg\!= (\lambda(T,\ T',\ T'',\ T''').\ RETURN\ (f\ T\ T'\ T''\ T''')) =$
$RES\ ((\lambda(a,\ b,\ c,\ d).\ f\ a\ b\ c\ d)\ `\ \{T.\ \Phi\ T\})$›
⟨*proof*⟩

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

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

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

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

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

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
‹$(uncurry2\ (RETURN\ ooo\ f),\ uncurry2\ (RETURN\ ooo\ g)) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge(x :: 'a)\ x'\ y\ y'\ (z :: 'c)\ (z' :: 'c2).$
$P\ ((x',\ y'),\ z') \Longrightarrow (((x,\ y),\ z),\ ((x',\ y'),\ z')) \in A \Longrightarrow$
$(f\ x\ y\ z,\ g\ x'\ y'\ z') \in B)$›
⟨*proof*⟩

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

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

More Simplification Theorems

lemma *nofail-Down-nofail*: ‹*nofail gS* $\Longrightarrow$ *fS* $\leq \Downarrow$ *R gS* $\Longrightarrow$ *nofail fS*›
⟨*proof*⟩

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

lemma *WHILEIT-refine-with-post*:
assumes *R0*: $I'\ x' \Longrightarrow (x,x') \in R$
assumes *IREF*: $\bigwedge x\ x'.\ \llbracket\ (x,x') \in R;\ I'\ x'\ \rrbracket \Longrightarrow I\ x$
assumes *COND-REF*: $\bigwedge x\ x'.\ \llbracket\ (x,x') \in R;\ I\ x;\ I'\ x'\ \rrbracket \Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.\ \llbracket\ (x,x') \in R;\ b\ x;\ b'\ x';\ I\ x;\ I'\ x';\ f'\ x' \leq SPEC\ I'\ \rrbracket \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')$
shows $WHILEIT\ I\ b\ f\ x \leq \Downarrow R\ (WHILEIT\ I'\ b'\ f'\ x')$
⟨*proof*⟩

0.0.4 Some Refinement

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

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

lemma *no-fail-spec-le-RETURN-itself*: ‹*nofail f* $\Longrightarrow f \leq SPEC(\lambda x.\ RETURN\ x \leq f)$›
⟨*proof*⟩

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 ‹$y \leq \Downarrow \{((i,\ S),\ S').\ P\ i\ S\ S' \wedge Q\ S'\}\ (f\ S)$›
⟨*proof*⟩

lemma *weaken-*$\Downarrow$: ‹$R' \subseteq R \Longrightarrow f \leq \Downarrow R'\ g \Longrightarrow f \leq \Downarrow R\ g$›
⟨*proof*⟩

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-*$\Downarrow$*[OF - I]*››)

lemma *refine-SPEC-refine-Down*:
‹$f \leq SPEC\ C \longleftrightarrow f \leq \Downarrow \{(T',\ T).\ T = T' \wedge C\ T'\}\ (SPEC\ C)$›
⟨*proof*⟩

0.0.5 More declarations

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

lemma *diff-add-mset-remove1*: ‹*NO-MATCH* {#} $N \Longrightarrow M$ − *add-mset* a N = *remove1-mset* a (M − N)›
⟨*proof*⟩

0.0.6 List relation

lemma *list-rel-take*:
‹(ba, ab) ∈ ⟨A⟩*list-rel* ⟹ (*take* b ba, *take* b ab) ∈ ⟨A⟩*list-rel*›
⟨*proof*⟩

lemma *list-rel-update'*:
fixes R
assumes *rel*: ‹(xs, ys) ∈ ⟨R⟩*list-rel*› **and**
h: ‹(bi, b) ∈ R›
shows ‹(*list-update* xs ba bi, *list-update* ys ba b) ∈ ⟨R⟩*list-rel*›
⟨*proof*⟩

lemma *list-rel-in-find-correspondanceE*:
assumes ‹(M, M') ∈ ⟨R⟩*list-rel*› **and** ‹L ∈ *set* M›
obtains L' **where** ‹(L, L') ∈ R› **and** ‹L' ∈ *set* M'›
⟨*proof*⟩

0.0.7 More Functions, Relations, and Theorems

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

lemma *Down-id-eq*: ⇓ *Id* a = a
⟨*proof*⟩

lemma *Down-itself-via-SPEC*:
assumes ‹I ≤ *SPEC* P› **and** ‹⋀x. P x ⟹ (x, x) ∈ R›
shows ‹I ≤ ⇓ R I›
⟨*proof*⟩
lemma *RES-ASSERT-moveout*:
(⋀a. a ∈ P ⟹ Q a) ⟹ *do* {a ← *RES* P; *ASSERT*(Q a); (f a)} =
do {a← *RES* P; (f a)}
⟨*proof*⟩

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*›
⟨*proof*⟩

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*: ‹(⋀*x*. *g x* = *g′ x*) ⟹ (*do* {*a* ← *f* :: ′*a nres*; *g a*}) = (*do* {*a* ← *f* :: ′*a nres*; *g′ a*})›
  ⟨*proof*⟩

lemma *case-prod-cong*:
  ‹(⋀*a b*. *f a b* = *g a b*) ⟹ (*case x of* (*a*, *b*) ⇒ *f a b*) = (*case x of* (*a*, *b*) ⇒ *g a b*)›
  ⟨*proof*⟩

lemma *if-replace-cond*: ‹(*if b then P b else Q b*) = (*if b then P True else Q False*)›
  ⟨*proof*⟩

lemma *foldli-cong2*:
  assumes
    le: ‹*length l* = *length l′*› **and**
    σ: ‹*σ* = *σ′*› **and**
    c: ‹*c* = *c′*› **and**
    H: ‹⋀*σ x*. *x* < *length l* ⟹ *c′ σ* ⟹ *f* (*l* ! *x*) *σ* = *f′* (*l′* ! *x*) *σ*›
  shows ‹*foldli l c f σ* = *foldli l′ c′ f′ σ′*›
⟨*proof*⟩

lemma *foldli-foldli-list-nth*:
  ‹*foldli xs c P a* = *foldli* [*0*..<*length xs*] *c* (*λi*. *P* (*xs* ! *i*)) *a*›
⟨*proof*⟩

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)›
  ⟨*proof*⟩

lemma *RES-SPEC-conv*: ‹*RES P* = *SPEC* (*λv*. *v* ∈ *P*)›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

lemma (**in** −) *WHILEIT-rule-stronger-inv-RES′*:
  assumes

⟨*wf R*⟩ **and**
⟨*I s*⟩ **and**
⟨*I′ s*⟩
⟨$\bigwedge$*s. I s* ⟹ *I′ s* ⟹ *b s* ⟹ *f s* ≤ *SPEC* (λ*s′. I s′* ∧ *I′ s′* ∧ (*s′*, *s*) ∈ *R*)⟩ **and**
⟨$\bigwedge$*s. I s* ⟹ *I′ s* ⟹ ¬ *b s* ⟹ *RETURN s* ≤ ⇓ *H* (*RES* Φ)⟩
shows ⟨*WHILE*$_T$^{I} *b f s* ≤ ⇓ *H* (*RES* Φ)⟩
⟨*proof*⟩

lemma *same-in-Id-option-rel*:
⟨*x* = *x′* ⟹ (*x*, *x′*) ∈ ⟨*Id*⟩*option-rel*⟩
⟨*proof*⟩

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$$^{λ(found,\ i).\ i ≥ a ∧ i ≤ length\ C ∧ i ≤ b ∧ (∀j∈\{a..<i\}.\ ¬P\ (C!j)) ∧}$ (∀ *j. found* = *Some j* ⟶ (*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* ≤ *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*))))⟩
⟨*proof*⟩

lemma *nfoldli-cong2*:
assumes
le: ⟨*length l* = *length l′*⟩ **and**
σ: ⟨σ = σ′⟩ **and**
c: ⟨*c* = *c′*⟩ **and**
H: ⟨$\bigwedge$σ *x*. *x* < *length l* ⟹ *c′* σ ⟹ *f* (*l* ! *x*) σ = *f′* (*l′* ! *x*) σ⟩
shows ⟨*nfoldli l c f* σ = *nfoldli l′ c′ f′* σ′⟩
⟨*proof*⟩

lemma *nfoldli-nfoldli-list-nth*:
⟨*nfoldli xs c P a* = *nfoldli* [*0*..<*length xs*] *c* (λ*i*. *P* (*xs* ! *i*)) *a*⟩
⟨*proof*⟩

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* = []⟩
⟨*proof*⟩

definition *list-rel-mset-rel* **where** *list-rel-mset-rel-internal*:
‹*list-rel-mset-rel* ≡ λR. ⟨*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*›
⟨*proof*⟩

lemma *list-rel-mset-rel-imp-same-length*: ‹(*a*, *b*) ∈ ⟨*R*⟩*list-rel-mset-rel* ⟹ *length a* = *size b*›
⟨*proof*⟩

lemma *while-upt-while-direct1*:
b ≥ *a* ⟹
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
}
⟨*proof*⟩

lemma *while-upt-while-direct2*:
b ≥ *a* ⟹
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
}
⟨*proof*⟩

lemma *while-upt-while-direct*:
b ≥ *a* ⟹
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
}
⟨*proof*⟩

lemma *while-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*
⟨*proof*⟩
lemma *nfoldli-while*: *nfoldli l c f σ*
≤
($WHILE_T{}^I$
(*FOREACH-cond c*) (λ*x*. *do* {*ASSERT* (*FOREACH-cond c x*); *FOREACH-body f x*}) (*l*, σ)
≫=
(λ(-, σ). *RETURN* σ))
⟨*proof*⟩

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*
⟨*proof*⟩

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* ≡ *l*[*i* := *l*!*j*, *j*:=*l*!*i*]

lemma *swap-nth*[*simp*]: ⟦*i* < *length l*; *j*<*length l*; *k*<*length l*⟧ ⟹
swap l i j!*k* = (
if k=*i then l*!*j*
else if k=*j then l*!*i*
else l!*k*
)
⟨*proof*⟩

lemma *swap-set*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ ⟹ *set* (*swap l i j*) = *set l*
⟨*proof*⟩

lemma *swap-multiset*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ ⟹ *mset* (*swap l i j*) = *mset l*
⟨*proof*⟩

lemma *swap-length*[*simp*]: *length* (*swap l i j*) = *length l*
⟨*proof*⟩

lemma *swap-same*[*simp*]: *swap l i i* = *l*
⟨*proof*⟩

lemma *distinct-swap*[*simp*]:
$[\![i<length\ l;\ j<length\ l]\!] \Longrightarrow distinct\ (swap\ l\ i\ j) = distinct\ l$
⟨*proof*⟩

lemma *map-swap*: $[\![i<length\ l;\ j<length\ l]\!]$
$\Longrightarrow map\ f\ (swap\ l\ i\ j) = swap\ (map\ f\ l)\ i\ j$
⟨*proof*⟩

lemma *swap-nth-irrelevant*:
‹$k \neq i \Longrightarrow k \neq j \Longrightarrow swap\ xs\ i\ j\ !\ k = xs\ !\ k$›
⟨*proof*⟩

lemma *swap-nth-relevant*:
‹$i < length\ xs \Longrightarrow j < length\ xs \Longrightarrow swap\ xs\ i\ j\ !\ i = xs\ !\ j$›
⟨*proof*⟩

lemma *swap-nth-relevant2*:
‹$i < length\ xs \Longrightarrow j < length\ xs \Longrightarrow swap\ xs\ j\ i\ !\ i = xs\ !\ j$›
⟨*proof*⟩

lemma *swap-nth-if*:
‹$i < length\ xs \Longrightarrow j < length\ xs \Longrightarrow swap\ xs\ i\ j\ !\ k =$
$(if\ k = i\ then\ xs\ !\ j\ else\ if\ k = j\ then\ xs\ !\ i\ else\ xs\ !\ k)$›
⟨*proof*⟩

lemma *drop-swap-irrelevant*:
‹$k > i \Longrightarrow k > j \Longrightarrow drop\ k\ (swap\ outl'\ j\ i) = drop\ k\ outl'$›
⟨*proof*⟩

lemma *take-swap-relevant*:
‹$k > i \Longrightarrow k > j \Longrightarrow take\ k\ (swap\ outl'\ j\ i) = swap\ (take\ k\ outl')\ i\ j$›
⟨*proof*⟩

lemma *tl-swap-relevant*:
‹$i > 0 \Longrightarrow j > 0 \Longrightarrow tl\ (swap\ outl'\ j\ i) = swap\ (tl\ outl')\ (i - 1)\ (j - 1)$›
⟨*proof*⟩

lemma *swap-only-first-relevant*:
‹$b \geq i \Longrightarrow a < length\ xs \Longrightarrow take\ i\ (swap\ xs\ a\ b) = take\ i\ (xs[a := xs\ !\ b])$›
⟨*proof*⟩

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*:
‹$[\![from \leq length\ xs;\ i < to - from]\!] \Longrightarrow Misc.slice\ from\ to\ xs\ !\ i = xs\ !\ (from + i)$›
⟨*proof*⟩

lemma *slice-irrelevant*[*simp*]:
‹$i < from \Longrightarrow Misc.slice\ from\ to\ (xs[i := C]) = Misc.slice\ from\ to\ xs$›
‹$i \geq to \Longrightarrow Misc.slice\ from\ to\ (xs[i := C]) = Misc.slice\ from\ to\ xs$›
‹$i \geq to \lor i < from \Longrightarrow Misc.slice\ from\ to\ (xs[i := C]) = Misc.slice\ from\ to\ xs$›
⟨*proof*⟩

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*]›
⟨*proof*⟩

lemma *drop-slice*[*simp*]:
‹*drop n* (*Misc.slice from to xs*) = *Misc.slice* (*from* + *n*) *to xs*› **for** *from n to xs*
⟨*proof*⟩

lemma *take-slice*[*simp*]:
‹*take n* (*Misc.slice from to xs*) = *Misc.slice from* (*min to* (*from* + *n*)) *xs*› **for** *from n to xs*
⟨*proof*⟩

lemma *slice-append*[*simp*]:
‹*to* ≤ *length xs* ⟹ *Misc.slice from to* (*xs* @ *ys*) = *Misc.slice from to xs*›
⟨*proof*⟩

lemma *slice-prepend*[*simp*]:
‹*from* ≥ *length xs* ⟹
Misc.slice from to (*xs* @ *ys*) = *Misc.slice* (*from* − *length xs*) (*to* − *length xs*) *ys*›
⟨*proof*⟩

lemma *slice-len-min-If*:
‹*length* (*Misc.slice from to xs*) =
(*if from* < *length xs then min* (*length xs* − *from*) (*to* − *from*) *else 0*)›
⟨*proof*⟩

lemma *slice-start0*: ‹*Misc.slice 0 to xs* = *take to xs*›
⟨*proof*⟩

lemma *slice-end-length*: ‹*n* ≥ *length xs* ⟹ *Misc.slice to n xs* = *drop to xs*›
⟨*proof*⟩

lemma *slice-swap*[*simp*]:
‹*l* ≥ *from* ⟹ *l* < *to* ⟹ *k* ≥ *from* ⟹ *k* < *to* ⟹ *from* < *length arena* ⟹
Misc.slice from to (*swap arena l k*) = *swap* (*Misc.slice from to arena*) (*k* − *from*) (*l* − *from*)›
⟨*proof*⟩

lemma *drop-swap-relevant*[*simp*]:
‹*i* ≥ *k* ⟹ *j* ≥ *k* ⟹ *j* < *length outl′* ⟹*drop k* (*swap outl′ j i*) = *swap* (*drop k outl′*) (*j* − *k*) (*i* − *k*)›
⟨*proof*⟩

lemma *swap-swap*: ‹*k* < *length xs* ⟹ *l* < *length xs* ⟹ *swap xs k l* = *swap xs l k*›
⟨*proof*⟩

lemma *list-rel-append-single-iff*:
‹(*xs* @ [*x*], *ys* @ [*y*]) ∈ ⟨*R*⟩*list-rel* ⟷
(*xs*, *ys*) ∈ ⟨*R*⟩*list-rel* ∧ (*x*, *y*) ∈ *R*›
⟨*proof*⟩

lemma *nth-in-sliceI*:
‹*i* ≥ *j* ⟹ *i* < *k* ⟹ *k* ≤ *length xs* ⟹ *xs* ! *i* ∈ *set* (*Misc.slice j k xs*)›
⟨*proof*⟩

lemma *slice-Suc*:

‹*Misc.slice* (*Suc j*) *k xs* = *tl* (*Misc.slice j k xs*)›
⟨*proof*⟩

lemma *slice-0*:
‹*Misc.slice 0 b xs* = *take b xs*›
⟨*proof*⟩

lemma *slice-end*:
‹*c* = *length xs* ⟹ *Misc.slice b c xs* = *drop b xs*›
⟨*proof*⟩

lemma *slice-append-nth*:
‹*a* ≤ *b* ⟹ *Suc b* ≤ *length xs* ⟹ *Misc.slice a* (*Suc b*) *xs* = *Misc.slice a b xs* @ [*xs* ! *b*]›
⟨*proof*⟩

lemma *take-set*: *set* (*take n l*) = { *l*!*i* | *i*. *i*<*n* ∧ *i*<*length l* }
⟨*proof*⟩

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

lemma (**in** −) *delete-index-and-swap-alt-def*:
‹*delete-index-and-swap S i* =
(*let x* = *last S in butlast* (*S*[*i* := *x*]))›
⟨*proof*⟩

lemma *swap-param*[*param*]: ⟦ *i*<*length l*; *j*<*length l*; (*l'*,*l*)∈⟨*A*⟩*list-rel*; (*i'*,*i*)∈*nat-rel*; (*j'*,*j*)∈*nat-rel*⟧
⟹ (*swap l' i' j'*, *swap l i j*)∈⟨*A*⟩*list-rel*
⟨*proof*⟩

lemma *mset-tl-delete-index-and-swap*:
assumes
‹*0* < *i*› **and**
‹*i* < *length outl'*›
shows ‹*mset* (*tl* (*delete-index-and-swap outl' i*)) =
remove1-mset (*outl'* ! *i*) (*mset* (*tl outl'*))›
⟨*proof*⟩

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

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

definition *append-ll* :: *'a list list* ⇒ *nat* ⇒ *'a* ⇒ *'a list list* **where**
‹*append-ll xs i x* = *list-update xs i* (*xs* ! *i* @ [*x*])›

definition (**in** −)*length-uint32-nat* **where**
[*simp*]: ‹*length-uint32-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* (- →$_f$ - [*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›
⟨*proof*⟩

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*) ⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *list-mset-assn-empty-nil*: ‹*list-mset-assn R* {#} [] = *emp*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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'*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma
Nil-list-mset-rel-iff:
‹([], *aaa*) ∈ *list-mset-rel* ⟷ *aaa* = {#}› **and**
empty-list-mset-rel-iff:
‹(*a*, {#}) ∈ *list-mset-rel* ⟷ *a* = []›
⟨*proof*⟩

lemma *snd-hnr-pure*:
‹*CONSTRAINT is-pure B* ⟹ (*return* ∘ *snd*, *RETURN* ∘ *snd*) ∈ $A^d *_a B^k \rightarrow_a B$›
⟨*proof*⟩

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* (λ*a b*. *Pair a b*)), *uncurry* (*RETURN oo* (λ*a b*. *Pair a b*))) ∈
$A^d *_a B^d \rightarrow_a$ *prod-assn A B*›
⟨*proof*⟩

This version works only for *pure* refinement relations:

lemma *the-hnr-keep*:
‹*CONSTRAINT is-pure A* ⟹ (*return o the*, *RETURN o the*) ∈ $[\lambda D.\ D \neq None]_a$ $(option\text{-}assn\ A)^k$
→ *A*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *list-mset-assn-pure-conv*:
‹*list-mset-assn* (*pure R*) = *pure* (⟨*R*⟩*list-rel-mset-rel*)›
⟨*proof*⟩

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*⟩

lemma *id-ref*: ‹(*return o id*, *RETURN o id*) ∈ $R^d \rightarrow_a R$›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *bool-assn-alt-def*: ‹*bool-assn a b* = ↑ (*a* = *b*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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^λ(j, ys). j ≥ 0 ∧ mset xs = mset ys ∧ 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 ‹$RETURN\ [Suc\ 0,\ 2,\ 0] = insert\text{-}sort\text{-}inner\ (<)\ (\lambda remove\ n.\ remove\ !\ n)\ [2::nat,\ 1,\ 0]\ 1$›
⟨*proof*⟩

definition *insert-sort* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow nat \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'a\ list\ nres$› **where**
‹$insert\text{-}sort\ R\ f\ xs = do\ \{$
$(i,\ ys) \leftarrow WHILE_T{}^{\lambda(i,\ ys).\ (ys = [] \lor i \le length\ ys) \land mset\ xs = mset\ ys}$
$(\lambda(i,\ ys).\ i < length\ ys)$
$(\lambda(i,\ ys).\ do\ \{$
$ASSERT(i < length\ ys);$
$ys \leftarrow insert\text{-}sort\text{-}inner\ R\ f\ ys\ i;$
$RETURN\ (i+1,\ ys)$
$\})$
$(1,\ xs);$
$RETURN\ ys$
$\}$›

lemma *insert-sort-inner*:
‹$(uncurry\ (insert\text{-}sort\text{-}inner\ R\ f),\ uncurry\ (\lambda m\ m'.\ reorder\text{-}list\ m'\ m)) \in$
$[\lambda(xs,\ i).\ i < length\ xs]_f\ \langle Id::\ ('a \times 'a)\ set\rangle list\text{-}rel \times_r nat\text{-}rel \rightarrow \langle Id\rangle\ nres\text{-}rel$›
⟨*proof*⟩

lemma *insert-sort-reorder-list*:
‹$(insert\text{-}sort\ R\ f,\ reorder\text{-}list\ vm) \in \langle Id\rangle list\text{-}rel \rightarrow_f \langle Id\rangle\ nres\text{-}rel$›
⟨*proof*⟩

definition *arl-replicate* **where**
$arl\text{-}replicate\ init\text{-}cap\ x \equiv do\ \{$
$let\ n = max\ init\text{-}cap\ minimum\text{-}capacity;$
$a \leftarrow Array.new\ n\ x;$
$return\ (a, init\text{-}cap)$
$\}$

definition ‹*op-arl-replicate = op-list-replicate*›
lemma *arl-fold-custom-replicate*:
‹*replicate = op-arl-replicate*›
⟨*proof*⟩

lemma *list-replicate-arl-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*CONSTRAINT is-pure R*›
shows ‹$(uncurry\ arl\text{-}replicate,\ uncurry\ (RETURN\ oo\ op\text{-}arl\text{-}replicate)) \in nat\text{-}assn^k *_a R^k \rightarrow_a arl\text{-}assn$
R›
⟨*proof*⟩

lemma *option-bool-assn-direct-eq-hnr*:
‹$(uncurry\ (return\ oo\ (=)),\ uncurry\ (RETURN\ oo\ (=))) \in$
$(option\text{-}assn\ bool\text{-}assn)^k *_a (option\text{-}assn\ bool\text{-}assn)^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

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*]:
‹$(return\ o\ (\lambda(a,\ \text{-}).\ (a,\ 1::nat)),\ RETURN\ o\ take1) \in [\lambda xs.\ xs \ne []]_a\ (arl\text{-}assn\ R)^d \rightarrow arl\text{-}assn\ R$›
⟨*proof*⟩

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 ≡ *uncurry2* (*uncurry2 f*)

abbreviation *uncurry6'* **where**
  uncurry6' f ≡ *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* ≤ *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*))))›
  ⟨*proof*⟩

lemma *list-assn-map-list-assn*: ‹*list-assn g* (*map f x*) *xi* = *list-assn* (λ*a c*. *g* (*f a*) *c*) *x xi*›
  ⟨*proof*⟩

lemma *hfref-imp2*: $(\bigwedge x\ y.\ S\ x\ y \Longrightarrow_t S'\ x\ y) \Longrightarrow [P]_a\ RR \rightarrow S \subseteq [P]_a\ RR \rightarrow S'$
  ⟨*proof*⟩

lemma *hr-comp-mono-entails*: ‹$B \subseteq C \Longrightarrow hr\text{-}comp\ a\ B\ x\ y \Longrightarrow_A hr\text{-}comp\ a\ C\ x\ y$›
  ⟨*proof*⟩

lemma *hfref-imp-mono-result*:
  $B \subseteq C \Longrightarrow [P]_a\ RR \rightarrow hr\text{-}comp\ a\ B \subseteq [P]_a\ RR \rightarrow hr\text{-}comp\ a\ C$
  ⟨*proof*⟩

lemma *hfref-imp-mono-result2*:
  $(\bigwedge x.\ P\ L\ x \Longrightarrow B\ L \subseteq C\ L) \Longrightarrow [P\ L]_a\ RR \rightarrow hr\text{-}comp\ a\ (B\ L) \subseteq [P\ L]_a\ RR \rightarrow hr\text{-}comp\ a\ (C\ L)$
  ⟨*proof*⟩

lemma *ex-assn-up-eq2*: ‹$(\exists_A ba.\ f\ ba * \uparrow (ba = c)) = (f\ c)$›
  ⟨*proof*⟩

lemma *ex-assn-pair-split*: ‹$(\exists_A b.\ P\ b) = (\exists_A a\ b.\ P\ (a,\ b))$›
⟨*proof*⟩

lemma *ex-assn-swap*: ‹$(\exists_A a\ b.\ P\ a\ b) = (\exists_A b\ a.\ P\ a\ b)$›
⟨*proof*⟩

lemma *ent-ex-up-swap*: ‹$(\exists_A aa.\ \uparrow (P\ aa)) = (\uparrow(\exists\, aa.\ P\ aa))$›
⟨*proof*⟩

lemma *ex-assn-def-pure-eq-middle3*:
‹$(\exists_A ba\ b\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb) * P\ b\ ba\ bb) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * P\ b\ (h\ b\ bb)\ bb)$›
‹$(\exists_A b\ ba\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb) * P\ b\ ba\ bb) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * P\ b\ (h\ b\ bb)\ bb)$›
‹$(\exists_A b\ bb\ ba.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb) * P\ b\ ba\ bb) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * P\ b\ (h\ b\ bb)\ bb)$›
‹$(\exists_A ba\ b\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb \wedge Q\ b\ ba\ bb)) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * \uparrow(Q\ b\ (h\ b\ bb)\ bb))$›
‹$(\exists_A b\ ba\ bb.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb \wedge Q\ b\ ba\ bb)) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * \uparrow(Q\ b\ (h\ b\ bb)\ bb))$›
‹$(\exists_A b\ bb\ ba.\ f\ b\ ba\ bb * \uparrow (ba = h\ b\ bb \wedge Q\ b\ ba\ bb)) = (\exists_A b\ bb.\ f\ b\ (h\ b\ bb)\ bb * \uparrow(Q\ b\ (h\ b\ bb)\ bb))$›
⟨*proof*⟩

lemma *ex-assn-def-pure-eq-middle2*:
‹$(\exists_A ba\ b.\ f\ b\ ba * \uparrow (ba = h\ b) * P\ b\ ba) = (\exists_A b\ .\ f\ b\ (h\ b) * P\ b\ (h\ b))$›
‹$(\exists_A b\ ba.\ f\ b\ ba * \uparrow (ba = h\ b) * P\ b\ ba) = (\exists_A b\ .\ f\ b\ (h\ b) * P\ b\ (h\ b))$›
‹$(\exists_A b\ ba.\ f\ b\ ba * \uparrow (ba = h\ b \wedge Q\ b\ ba)) = (\exists_A b.\ f\ b\ (h\ b) * \uparrow(Q\ b\ (h\ b)))$›
‹$(\exists_A\ ba\ b.\ f\ b\ ba * \uparrow (ba = h\ b \wedge Q\ b\ ba)) = (\exists_A b.\ f\ b\ (h\ b) * \uparrow(Q\ b\ (h\ b)))$›
⟨*proof*⟩

lemma *ex-assn-skip-first2*:
‹$(\exists_A ba\ bb.\ f\ bb * \uparrow(P\ ba\ bb)) = (\exists_A bb.\ f\ bb * \uparrow(\exists\, ba.\ P\ ba\ bb))$›
‹$(\exists_A bb\ ba.\ f\ bb * \uparrow(P\ ba\ bb)) = (\exists_A bb.\ f\ bb * \uparrow(\exists\, ba.\ P\ ba\ bb))$›
⟨*proof*⟩

lemma *fr-refl'*: ‹$A \Longrightarrow_A B \Longrightarrow C * A \Longrightarrow_A C * B$›
⟨*proof*⟩

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

lemma *hn-ctxt-prod-assn-prod*:
‹$hn\text{-}ctxt\ (R *a\ S)\ (a,\ b)\ (a',\ b') = hn\text{-}ctxt\ R\ a\ a' * hn\text{-}ctxt\ S\ b\ b'$›
⟨*proof*⟩

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*⟩

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*)xyzabcdegh= (*RETURN*$(f$xyzabcdeg$h))
$\bigwedge f.$ (*RETURN* $\circ_{11}$ *f*)xyzabcdegh$i= (*RETURN*$(fxyzabcdegh$i))
$\bigwedge f.$ (*RETURN* $\circ_{12}$ *f*)xyzabcdeghij= (*RETURN*$(f$xyzabcdeghi$j))
$\bigwedge f.$ (*RETURN* $\circ_{13}$ *f*)xyzabcdeghij$l= (*RETURN*$(fxyzabcdeghij$l))
$\bigwedge f.$ (*RETURN* $\circ_{14}$ *f*)xyzabcdeghijlm= (*RETURN*$(f$xyzabcdeghijl$m))
$\bigwedge f.$ (*RETURN* $\circ_{15}$ *f*)xyzabcdeghijlm$n= (*RETURN*$(fxyzabcdeghijlm$n))
$\bigwedge f.$ (*RETURN* $\circ_{16}$ *f*)xyzabcdeghijlmnp= (*RETURN*$(f$xyzabcdeghijlmn$p))
$\bigwedge f.$ (*RETURN* $\circ_{17}$ *f*)xyzabcdeghijlmnp$r=
(*RETURN*$(f$xyzabcdeghijlmnpr))
$\bigwedge f.$ (*RETURN* $\circ_{18}$ *f*)xyzabcdeghijlmnprs=
(*RETURN*$(f$xyzabcdeghijlmnpr$s))
$\bigwedge f.$ (*RETURN* $\circ_{19}$ *f*)xyzabcdeghijlmnprs$t=
(*RETURN*$(f$xyzabcdeghijlmnprst))
$\bigwedge f.$ (*RETURN* $\circ_{20}$ *f*)xyzabcdeghijlmnprstu=
(*RETURN*$(f$xyzabcdeghijlmnprst$u))
⟨*proof*⟩

lemma *norm-return-o*[*to-hnr-post*]:
$\bigwedge f.$ (*return oooo f*)xyza = (*return*$(f$xyz$a))
$\bigwedge f.$ (*return ooooo f*)xyza$b = (*return*$(fxyza$b))
$\bigwedge f.$ (*return oooooo f*)xyzabc = (*return*$(f$xyzab$c))
$\bigwedge f.$ (*return ooooooo f*)xyzabc$d = (*return*$(fxyzabc$d))
$\bigwedge f.$ (*return oooooooo f*)xyzabcde = (*return*$(f$xyzabcd$e))
$\bigwedge f.$ (*return ooooooooo f*)xyzabcde$g = (*return*$(fxyzabcde$g))
$\bigwedge f.$ (*return oooooooooo f*)xyzabcdegh= (*return*$(f$xyzabcdeg$h))
$\bigwedge f.$ (*return* $\circ_{11}$ *f*)xyzabcdegh$i= (*return*$(fxyzabcdegh$i))
$\bigwedge f.$ (*return* $\circ_{12}$ *f*)xyzabcdeghij= (*return*$(f$xyzabcdeghi$j))
$\bigwedge f.$ (*return* $\circ_{13}$ *f*)xyzabcdeghij$l= (*return*$(fxyzabcdeghij$l))
$\bigwedge f.$ (*return* $\circ_{14}$ *f*)xyzabcdeghijlm= (*return*$(f$xyzabcdeghijl$m))
$\bigwedge f.$ (*return* $\circ_{15}$ *f*)xyzabcdeghijlm$n= (*return*$(fxyzabcdeghijlm$n))
$\bigwedge f.$ (*return* $\circ_{16}$ *f*)xyzabcdeghijlmnp= (*return*$(f$xyzabcdeghijlmn$p))
$\bigwedge f.$ (*return* $\circ_{17}$ *f*)xyzabcdeghijlmnp$r=
(*return*$(f$xyzabcdeghijlmnpr))
$\bigwedge f.$ (*return* $\circ_{18}$ *f*)xyzabcdeghijlmnprs=
(*return*$(f$xyzabcdeghijlmnpr$s))
$\bigwedge f.$ (*return* $\circ_{19}$ *f*)xyzabcdeghijlmnprs$t=
(*return*$(f$xyzabcdeghijlmnprst))
$\bigwedge f.$ (*return* $\circ_{20}$ *f*)xyzabcdeghijlmnprstu=
(*return*$(f$xyzabcdeghijlmnprst$u))
⟨*proof*⟩

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*⟩

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* :: $('a \Rightarrow 'b \Rightarrow assn) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow assn$ **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*:: $('a \Rightarrow 'b \Rightarrow assn) \Rightarrow nat\ list \Rightarrow\ 'a\ list \Rightarrow 'b\ list \Rightarrow assn$ **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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *heap-list-all-nth-single*: ‹*heap-list-all-nth R* [*a*] *xs ys* = *R* (*xs* ! *a*) (*ys* ! *a*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *heap-list-add-same-length*:
‹*h* ⊨ *heap-list-all R*′ *xs p* ⟹ *length p* = *length xs*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

definition *arrayO-assn* :: ‹('*a* ⇒ '*b*::*heap* ⇒ *assn*) ⇒ '*a list* ⇒ '*b array* ⇒ *assn*› **where**

‹*arrayO-assn R′ xs axs ≡ ∃$_A$ 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 ≡*
∃$_A$ *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*⟩

lemma *arrayO-except-assn-array0-index*:
‹*i < length xs* ⟹ *arrayO-except-assn R* [*i*] *xs asx* (λ*p*. *R* (*xs* ! *i*) (*p* ! *i*)) = *arrayO-assn R xs asx*›
⟨*proof*⟩

lemma *arrayO-nth-rule*[*sep-heap-rules*]:
assumes *i*: ‹*i < length a*›
shows ‹ <*arrayO-assn* (*arl-assn R*) *a ai*> *Array.nth ai i* <λ*r*. *arrayO-except-assn* (*arl-assn R*) [*i*] *a ai*
(λ*r′*. *arl-assn R* (*a* ! *i*) *r* ∗ ↑(*r* = *r′* ! *i*))>›
⟨*proof*⟩

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* <λ*r*. *arrayO-assn R x xi* ∗ ↑(*r* = *length x*)>$_t$›
⟨*proof*⟩

lemma *length-a-hnr*[*sepref-fr-rules*]:
‹(*length-a*, *RETURN o op-list-length*) ∈ (*arrayO-assn R*)^k $\rightarrow_a$ *nat-assn*›
⟨*proof*⟩

lemma *le-length-ll-nemptyD*: ‹*b < length-ll a ba* ⟹ *a* ! *ba* ≠ []›
⟨*proof*⟩

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*)>$_t$›
⟨*proof*⟩

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*›
⟨*proof*⟩

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)›
⟨proof⟩

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⟩

definition *append-el-aa* :: ('a::{default,heap} array-list) array ⇒
nat ⇒ 'a ⇒ ('a array-list) array Heap**where**
append-el-aa ≡ λa i x. do {
j ← Array.nth a i;
a' ← arl-append j x;
Array.upd i a' a
}

lemma *sep-auto-is-stupid*:
fixes *R* :: ‹'a ⇒ 'b::{heap,default} ⇒ assn›
assumes *p*: ‹is-pure R›
shows
‹<∃$_A$p. R1 p ∗ R2 p ∗ arl-assn R l' aa ∗ R x x' ∗ R4 p>
arl-append aa x' <λr. (∃$_A$p. arl-assn R (l' @ [x]) r ∗ R1 p ∗ R2 p ∗ R x x' ∗ R4 p ∗ true) >›
⟨proof⟩

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

lemma *heap-list-all-nth-cong*:
assumes
‹∀ i ∈ set is. xs ! i = xs' ! i› **and**
‹∀ i ∈ set is. ys ! i = ys' ! i›
shows ‹heap-list-all-nth R is xs ys = heap-list-all-nth R is xs' ys'›
⟨proof⟩

lemma *append-aa-hnr*[*sepref-fr-rules*]:
fixes *R* :: ‹'a ⇒ 'b :: {heap, default} ⇒ assn›
assumes *p*: ‹is-pure R›
shows
‹(uncurry2 append-el-aa, uncurry2 (RETURN ∘∘∘ append-ll)) ∈
[λ((l,i),x). i < length l]$_a$ (arrayO-assn (arl-assn R))^d $*_a$ nat-assn^k $*_a$ R^k → (arrayO-assn (arl-assn R))›
⟨proof⟩

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 ← Array.nth a i;
a' ← 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 ↑ ((*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* (λ*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*
(λ*r'*. *arl-assn R* ((*a* ! *bb*)[*ba* := *b*]) (*aa*, *bc*)) ∗ *R b bi* ∗ *true*>›
⟨*proof*⟩

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* ∗ (∃$_A$*x*. *arrayO-assn* (*arl-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*⟩

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*⟩

lemma *set-butlast-aa-rule*[*sep-heap-rules*]:
assumes ‹*is-pure R*› **and**
‹*b* < *length a*› **and**
‹*a* ! *b* ≠ []›
shows ‹<*arrayO-assn* (*arl-assn R*) *a ai*> *set-butlast-aa ai b*
<λ*r*. (∃$_A$*x*. *arrayO-assn* (*arl-assn R*) *x r* ∗ ↑ (*x* = *set-butlast-ll a b*))>$_t$›
⟨*proof*⟩

lemma *set-butlast-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry set-butlast-aa*, *uncurry* (*RETURN oo set-butlast-ll*)) ∈
[λ(*l*,*i*). *i* < *length l* ∧ *l* ! *i* ≠ []]$_a$ (*arrayO-assn* (*arl-assn R*))^d ∗$_a$ *nat-assn*^{k} → (*arrayO-assn* (*arl-assn R*))›
⟨*proof*⟩

definition *last-aa* :: ('*a*::*heap array-list*) *array* ⇒ *nat* ⇒ '*a Heap* **where**
‹*last-aa xs i* = *do* {
x ← *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
<λ*r*. *arrayO-assn* (*arl-assn R*) *a ai* ∗ (∃$_A$*x*. *R x r* ∗ ↑ (*x* = *last-ll a b*))>$_t$›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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. ∃A x. arrayO-assn (arl-assn R) aa a ∗
              (R x r ∗
               ↑ (x = nth-ll aa b ba)) ∗
              true>›
⟨proof⟩

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
          <λr. ∃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⟩

lemma length-update-ll[simp]: ‹length (update-ll a bb b c) = length a›
  ⟨proof⟩

lemma length-ll-update-ll:
  ‹bb < length a ⟹ length-ll (update-ll a bb b c) bb = length-ll a bb›
  ⟨proof⟩

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⟩
```

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›
  ⟨proof⟩
```

lemma *of-list-op-list-copy-arrayO*[*sepref-fr-rules*]:
‹(*Array.of-list*, *RETURN* ∘ *op-list-copy*) ∈ (*list-assn* (*arl-assn R*))^d $\to_a$ *arrayO-assn* (*arl-assn R*)›
⟨*proof*⟩

sepref-definition
arrayO-ara-empty-sz-code
is *RETURN o arrayO-ara-empty-sz*
:: ‹*nat-assn*^{k} $\to_a$ *arrayO-assn* (*arl-assn* (*R*::'*a* ⇒ '*b*::{*heap*, *default*} ⇒ *assn*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *arrayO-raa-empty-sz-init-lrl*[*sepref-fr-rules*]:
‹(*arrayO-ara-empty-sz-code*, *RETURN o init-lrl*) ∈
nat-assn^k $\to_a$ *arrayO-assn* (*arl-assn R*)›
⟨*proof*⟩

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* ≤ *length* (*b* ! *a*)› **and**
‹*a* < *length b*›
shows ‹<*arrayO-except-assn* (*arl-assn R*) [*a*] *b bi*
(λ*r*'. *arl-assn R* (*b* ! *a*) (*aaa*, *n*) ∗ ↑ ((*aaa*, *n*) = *r*' ! *a*))>
Array.upd a (*aaa*, *ba*) *bi*
<λ*r*. $\exists_A$*x*. *arrayO-assn* (*arl-assn R*) *x r* ∗ *true* ∗
↑ (*x* = *b*[*a* := *take ba* (*b* ! *a*)])>›
⟨*proof*⟩

lemma *shorten-take-aa-hnr*[*sepref-fr-rules*]:
‹(*uncurry2 shorten-take-aa*, *uncurry2* (*RETURN ooo shorten-take-ll*)) ∈
[λ((*L*, *j*), *W*). *j* ≤ *length* (*W* ! *L*) ∧ *L* < *length W*]$_a$
nat-assn^k $*_a$ *nat-assn*^{k} $*_a$ (*arrayO-assn* (*arl-assn R*))^d → *arrayO-assn* (*arl-assn R*)›
⟨*proof*⟩

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* ≡ ∃$_A$*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* ≡
∃$_A$ *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*⟩

lemma *arlO-assn-except-array0-index*:
‹*i* < *length xs* ⟹ *arlO-assn-except R* [*i*] *xs asx* (λ*p*. *R* (*xs* ! *i*) (*p* ! *i*)) = *arlO-assn R xs asx*›
⟨*proof*⟩

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* <λ*r*. *arlO-assn-except* (*array-assn R*) [*i*] *a ai*
(λ*r′*. *array-assn R* (*a* ! *i*) *r* ∗ ↑(*r* = *r′* ! *i*))>›
⟨*proof*⟩

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* <λ*r*. *arlO-assn R x xi* ∗ ↑(*r* = *length x*)>$_t$›
⟨*proof*⟩

lemma *length-ra-hnr*[*sepref-fr-rules*]:
‹(*length-ra*, *RETURN o op-list-length*) ∈ (*arlO-assn R*)^k →$_a$ *nat-assn*›
⟨*proof*⟩

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* ≠ []›
⟨*proof*⟩

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*
<λ*r*. *arlO-assn* (*array-assn R*) *xs a* ∗ ↑ (*r* = *length-rll xs b*)>$_t$›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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* ∗
↑ (*aa* = *r'* ! *bb*)) ∗ *R b bi*>
Array.upd ba bi aa
<λ*aa*. *arlO-assn-except* (*array-assn R*) [*bb*] *a ai*
(λ*r'*. *array-assn R* ((*a* ! *bb*)[*ba* := *b*]) *aa*) ∗ *R b bi* ∗ *true*>›
⟨*proof*⟩

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*
<λ*r*. *R b bi* ∗ (∃$_A$*x*. *arlO-assn* (*array-assn R*) *x r* ∗ ↑ (*x* = *update-rll a bb ba b*))>$_t$›
⟨*proof*⟩

lemma *update-raa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-raa*, *uncurry3* (*RETURN oooo update-rll*)) ∈
[λ(((*l*,*i*), *j*), *x*). *i* < *length l* ∧ *j* < *length-rll l i*]$_a$ (*arlO-assn* (*array-assn R*))^d $*_a$ *nat-assn*^{k} $*_a$

nat-assn^k $*_a$ *R*^{k} → (*arlO-assn* (*array-assn R*))›
⟨*proof*⟩

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
<λ*r*. ∃$_A$*x*. *arlO-assn* (*array-assn R*) *aa a* ∗
(*R x r* ∗
↑ (*x* = *nth-rll aa b ba*)) ∗
true>›
⟨*proof*⟩

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
<λ*r*. ∃$_A$*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*⟩

lemma *length-update-rll*[*simp*]: ‹*length* (*update-rll a bb b c*) = *length a*›
⟨*proof*⟩

lemma *length-rll-update-rll*:
‹*bb* < *length a* ⟹ *length-rll* (*update-rll a bb b c*) *bb* = *length-rll a bb*›
⟨*proof*⟩

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-rll xs k* ∧ *j* < *length-rll xs k*]$_a$
(*arlO-assn* (*array-assn R*))^d $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} → (*arlO-assn* (*array-assn R*))›
⟨*proof*⟩

definition *update-ra* :: ‹'*a arrayO-raa* ⇒ *nat* ⇒ '*a array* ⇒ '*a arrayO-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-assn R l xs*> *update-ra xs n x* <*arlO-assn R* (*l*[*n*:=*y*])>$_t$›
⟨*proof*⟩
lemma *ex-assn-up-eq*: ‹($\exists_A$*x*. *P x* ∗ ↑(*x* = *a*) ∗ *Q*) = (*P a* ∗ *Q*)›
  ⟨*proof*⟩
lemma *update-ra-list-update*[*sepref-fr-rules*]:
  ‹(*uncurry2 update-ra*, *uncurry2* (*RETURN ooo list-update*)) ∈
   [λ((*xs*, *n*), -). *n* < *length xs*]$_a$ (*arlO-assn R*)^d $*_a$ *nat-assn*^{k} $*_a$ R^d → (*arlO-assn R*)›
⟨*proof*⟩
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* ≠ [] ⟹ *heap-list-all R* (*va* @ *y*) [] = *false*›
  ⟨*proof*⟩

lemma *heap-list-all-Nil-append*:
  ‹*y* ≠ [] ⟹ *heap-list-all R* [] (*va* @ *y*) = *false*›
  ⟨*proof*⟩

lemma *heap-list-all-append*: ‹*heap-list-all R* (*l* @ [*y*]) (*l'* @ [*x*])
  = *heap-list-all R* (*l*) (*l'*) ∗ *R y x*›
  ⟨*proof*⟩
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*⟩

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*›
  ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

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*>
 ⟨*proof*⟩

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*⟩

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*›
 ⟨*proof*⟩

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*)›
 ⟨*proof*⟩

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

lemma *less-upper-bintrunc-id*: ‹*n* < *2* ^*b* ⟹ *n* ≥ *0* ⟹ *bintrunc b n* = *n*›
 ⟨*proof*⟩

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⟩

lemma *and-eq-bits-eqI*: ‹$(\bigwedge n.\ c\ !!\ n = (a\ !!\ n \wedge b\ !!\ n)) \Longrightarrow a\ AND\ b = c$› **for** *a b c* :: ‹- *word*›
⟨proof⟩

lemma *pow2-mono-word-less*:
‹$m < LENGTH('a) \Longrightarrow n < LENGTH('a) \Longrightarrow m < n \Longrightarrow (2 :: 'a :: len\ word)\ \hat{}\ m < 2\ \hat{}\ n$›
⟨proof⟩

lemma *pow2-mono-word-le*:
‹$m < LENGTH('a) \Longrightarrow n < LENGTH('a) \Longrightarrow m \leq n \Longrightarrow (2 :: 'a :: len\ word)\ \hat{}\ m \leq 2\ \hat{}\ n$›
⟨proof⟩

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⟩

definition *uint32-max'* **where**
[*simp*, *symmetric*, *code*]: ‹*uint32-max'* = *uint32-max*›

lemma [*code*]: ‹*uint32-max'* = *4294967295*›
⟨proof⟩

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⟩

32-bits

lemma *word-nat-of-uint32-Rep-inject*[*simp*]: ‹nat-of-uint32 ai = nat-of-uint32 bi ⟷ ai = bi›
⟨proof⟩

lemma *nat-of-uint32-012*[*simp*]: ‹nat-of-uint32 0 = 0› ‹nat-of-uint32 2 = 2› ‹nat-of-uint32 1 = 1›
⟨proof⟩

lemma *nat-of-uint32-3*: ‹nat-of-uint32 3 = 3›
⟨proof⟩

lemma *nat-of-uint32-Suc03-iff*:
‹nat-of-uint32 a = Suc 0 ⟷ a = 1›
‹nat-of-uint32 a = 3 ⟷ a = 3›
⟨proof⟩

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)
⟨proof⟩

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⟩

instantiation *uint32* :: *default*
begin
definition *default-uint32* :: *uint32* **where**
‹default-uint32 = 0›
instance
⟨proof⟩
end

instance *uint32* :: *heap*
⟨proof⟩

instance *uint32* :: *semiring-numeral*
⟨proof⟩

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
⟨*proof*⟩
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*))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *nat-of-uint32-mod-2*:
‹*nat-of-uint32 L mod 2* = *nat-of-uint32* (*L mod 2*)›
⟨*proof*⟩

lemma *bitAND-1-mod-2-uint32*: ‹*bitAND L 1* = *L mod 2*› **for** *L* :: *uint32*
⟨*proof*⟩

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*⟩

lemma *nat-of-uint32-XOR*: ‹*nat-of-uint32* (*a XOR b*) = *nat-of-uint32 a XOR nat-of-uint32 b*›
⟨*proof*⟩

lemma *nat-of-uint32-0-iff*: ‹*nat-of-uint32 xi* = *0* ⟷ *xi* = *0*› **for** *xi*
⟨*proof*⟩

lemma *nat-0-AND*: ‹*0 AND n* = *0*› **for** *n* :: *nat*
⟨*proof*⟩

lemma *uint32-0-AND*: ‹*0 AND n* = *0*› **for** *n* :: *uint32*
⟨*proof*⟩

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*›
⟨*proof*⟩

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

lemma *nat-of-uint32-uint32-of-nat-id*: ‹*n* ≤ *uint32-max* ⟹ *nat-of-uint32* (*uint32-of-nat n*) = *n*›
⟨*proof*⟩

lemma *uint32-less-than-0*[*iff*]: ‹(*a*::*uint32*) ≤ *0* ⟷ *a* = *0*›
⟨*proof*⟩

lemma *nat-of-uint32-less-iff*: ‹*nat-of-uint32 a* < *nat-of-uint32 b* ⟷ *a* < *b*›
⟨*proof*⟩

lemma *nat-of-uint32-le-iff*: ‹*nat-of-uint32 a* ≤ *nat-of-uint32 b* ⟷ *a* ≤ *b*›
⟨*proof*⟩

lemma *nat-of-uint32-max*:
‹*nat-of-uint32* (*max ai bi*) = *max* (*nat-of-uint32 ai*) (*nat-of-uint32 bi*)›
⟨*proof*⟩

lemma *mult-mod-mod-mult*:
‹*b* < *n div a* ⟹ *a* > *0* ⟹ *b* > *0* ⟹ *a* ∗ *b mod n* = *a* ∗ (*b mod n*)› **for** *a b n* :: *int*
⟨*proof*⟩

lemma *nat-of-uint32-distrib-mult2*:
assumes ‹*nat-of-uint32 xi* ≤ *uint32-max div 2*›
shows ‹*nat-of-uint32* (*2* ∗ *xi*) = *2* ∗ *nat-of-uint32 xi*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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′*›
⟨*proof*⟩

lemma *word-of-int-int-unat*[*simp*]: ‹*word-of-int* (*int* (*unat x*)) = *x*›
⟨*proof*⟩

lemma *uint32-of-nat-nat-of-uint32*[*simp*]: ‹*uint32-of-nat* (*nat-of-uint32 x*) = *x*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *nat-of-uint32-mod-232*:
shows ‹*nat-of-uint32 xi* = *nat-of-uint32 xi mod 2*^*32*›
⟨*proof*⟩

lemma *transfer-pow-uint32*:
‹*Transfer.Rel* (*rel-fun cr-uint32* (*rel-fun* (=) *cr-uint32*)) ((^)) ((^))›
⟨*proof*⟩

lemma *uint32-mod-232-eq*:
fixes *xi* :: *uint32*
shows ‹*xi* = *xi mod 2*^*32*›
⟨*proof*⟩

lemma *nat-of-uint32-numeral-mod-232*:
‹*nat-of-uint32* (*numeral n*) = *numeral n mod 2*^*32*›
⟨*proof*⟩

lemma *int-of-uint32-alt-def*: ‹*int-of-uint32 n* = *int* (*nat-of-uint32 n*)›
⟨*proof*⟩

lemma *int-of-uint32-numeral*[*simp*]:
‹*numeral n* ≤ ((*2* ^ *32* − *1*)::*nat*) ⟹ *int-of-uint32* (*numeral n*) = *numeral n*›
⟨*proof*⟩

lemma *nat-of-uint32-numeral-iff* [*simp*]:
‹*numeral n* ≤ ((*2* ^ *32* − *1*)::*nat*) ⟹ *nat-of-uint32 a* = *numeral n* ⟷ *a* = *numeral n*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nat-of-uint32-div*:
‹*nat-of-uint32* (*a div b*) = *nat-of-uint32 a div nat-of-uint32 b*›
⟨*proof*⟩

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*›
⟨*proof*⟩

instantiation *uint64* :: *default*
begin
definition *default-uint64* :: *uint64* **where**
‹*default-uint64* = *0*›
instance
⟨*proof*⟩
end

instance *uint64* :: *heap*
⟨*proof*⟩

instance *uint64* :: *semiring-numeral*
⟨*proof*⟩

lemma *nat-of-uint64-012*[*simp*]: ‹*nat-of-uint64 0 = 0*› ‹*nat-of-uint64 2 = 2*› ‹*nat-of-uint64 1 = 1*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *nat-of-uint64-uint64-of-nat-id*: ‹*n* ≤ *uint64-max* ⟹ *nat-of-uint64* (*uint64-of-nat n*) = *n*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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

lemma *uint64-less-than-0*[*iff*]: ‹(*a*::*uint64*) ≤ *0* ⟷ *a* = *0*›
⟨*proof*⟩

lemma *nat-of-uint64-less-iff*: ‹*nat-of-uint64 a* < *nat-of-uint64 b* ⟷ *a* < *b*›
⟨*proof*⟩

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*⟩

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*⟩

lemma *nat-of-uint64-numeral*[*simp*]:
‹*numeral n* ≤ ((*2* ^ *64* − *1*)::*nat*) ⟹ *nat-of-uint64* (*numeral n*) = *numeral n*›
⟨*proof*⟩

lemma *int-of-uint64-alt-def*: ‹*int-of-uint64 n* = *int* (*nat-of-uint64 n*)›

⟨*proof*⟩

lemma *int-of-uint64-numeral*[*simp*]:
‹*numeral n* ≤ ((*2* ^ *64* − *1*)::*nat*) ⟹ *int-of-uint64* (*numeral n*) = *numeral n*›
⟨*proof*⟩

lemma *nat-of-uint64-numeral-iff* [*simp*]:
‹*numeral n* ≤ ((*2* ^ *64* − *1*)::*nat*) ⟹ *nat-of-uint64 a* = *numeral n* ⟷ *a* = *numeral n*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *numeral-uint64-eq0-iff* [*simp*]:
‹*numeral n* ≤ (*2*^*64*−*1* :: *nat*) ⟹ ((*0* :: *uint64*) = *numeral n*) ⟷ *0* = (*numeral n* :: *nat*)›
⟨*proof*⟩

lemma *transfer-pow-uint64*: ‹*Transfer.Rel* (*rel-fun cr-uint64* (*rel-fun* (=) *cr-uint64*)) (^) (^)›
⟨*proof*⟩

lemma *shiftl-t2n-uint64*: ‹*n* << *m* = *n* ∗ *2* ^ *m*› **for** *n* :: *uint64*
⟨*proof*⟩

lemma *mod2-bin-last*: ‹*a mod 2* = *0* ⟷ ¬*bin-last a*›
⟨*proof*⟩

lemma *bitXOR-1-if-mod-2-int*: ‹*bitOR L 1* = (*if L mod 2* = *0 then L* + *1 else L*)› **for** *L* :: *int*
⟨*proof*⟩

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*⟩

lemma *uint64-max-uint-def*: ‹*unat* (−*1* :: *64 Word.word*) = *uint64-max*›
⟨*proof*⟩

lemma *nat-of-uint64-le-uint64-max*: ‹*nat-of-uint64 x* ≤ *uint64-max*›
⟨*proof*⟩

lemma *bitOR-1-if-mod-2-uint64*: ‹*bitOR L 1* = (*if L mod 2* = *0 then L* + *1 else L*)› **for** *L* :: *uint64*
⟨*proof*⟩

lemma *nat-of-uint64-plus*:
‹*nat-of-uint64* (*a* + *b*) = (*nat-of-uint64 a* + *nat-of-uint64 b*) *mod* (*uint64-max* + *1*)›
⟨*proof*⟩

lemma *nat-and*:
‹*ai*≥ *0* ⟹ *bi* ≥ *0* ⟹ *nat* (*ai AND bi*) = *nat ai AND nat bi*›
⟨*proof*⟩

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*⟩
⟨*proof*⟩

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*⟩
⟨*proof*⟩

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*⟩
⟨*proof*⟩

lemma *Suc-0-le-uint64-max*: ⟨*Suc 0* ≤ *uint64-max*⟩
⟨*proof*⟩

lemma *nat-of-uint64-le-iff*: ⟨*nat-of-uint64 a* ≤ *nat-of-uint64 b* ⟷ *a* ≤ *b*⟩
⟨*proof*⟩

lemma *nat-of-uint64-notle-minus*:
⟨¬ *ai* < *bi* ⟹
nat-of-uint64 (*ai* − *bi*) = *nat-of-uint64 ai* − *nat-of-uint64 bi*⟩
⟨*proof*⟩

lemma *le-uint32-max-le-uint64-max*: ⟨*a* ≤ *uint32-max* + *2* ⟹ *a* ≤ *uint64-max*⟩
⟨*proof*⟩

lemma *nat-of-uint64-ge-minus*:
⟨*ai* ≥ *bi* ⟹
nat-of-uint64 (*ai* − *bi*) = *nat-of-uint64 ai* − *nat-of-uint64 bi*⟩
⟨*proof*⟩

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*⟩

lemma *sum-mod-uint64-max-le-uint64-max*[*simp*]: ⟨*sum-mod-uint64-max a b* ≤ *uint64-max*⟩
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *nat-of-uint32-le-uint64-max*: ‹*nat-of-uint32 n* ≤ *uint64-max*›
⟨*proof*⟩

lemma *nat-of-uint64-uint64-of-uint32*: ‹*nat-of-uint64* (*uint64-of-uint32 n*) = *nat-of-uint32 n*›
  ⟨*proof*⟩

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*: ‹$j \neq (0{::}uint64) \longleftrightarrow j > 0$›
  ⟨*proof*⟩

lemma *uint64-gt0-ge1*: ‹$j > 0 \longleftrightarrow j \geq (1{::}uint64)$›
  ⟨*proof*⟩

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*) ← $WHILE_T$ λ(*i*,*zs*). *i* ≤ *length xs* ∧ *take i zs* = *map R* (*take i xs*) ∧   *length zs* = *length xs* ∧ (∀ *k*≥*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*)›
  ⟨*proof*⟩

lemma *op-map-map-rel*:
  ‹(*op-map R e*, *RETURN o* (*map R*)) ∈ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *param-max-uint32*[*param*,*sepref-import-param*]:
(*max*,*max*)∈*uint32-rel* → *uint32-rel* → *uint32-rel* ⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*safe-constraint-rules*]:
‹*CONSTRAINT IS-LEFT-UNIQUE uint32-nat-rel*›
‹*CONSTRAINT IS-RIGHT-UNIQUE uint32-nat-rel*›
⟨*proof*⟩

lemma *shiftr1*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* ((>>))), *uncurry* (*RETURN oo* (>>))) ∈ $uint32\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a$ *uint32-assn*›
⟨*proof*⟩

lemma *shiftl1*[*sepref-fr-rules*]: ‹(*return o shiftl1*, *RETURN o shiftl1*) ∈ $nat\text{-}assn^k \rightarrow_a$ *nat-assn*›
⟨*proof*⟩

lemma *nat-of-uint32-rule*[*sepref-fr-rules*]:
‹(*return o nat-of-uint32*, *RETURN o nat-of-uint32*) ∈ $uint32\text{-}assn^k \rightarrow_a$ *nat-assn*›
⟨*proof*⟩

lemma *max-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo max*), *uncurry* (*RETURN oo max*)) ∈ $uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

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*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(array\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \rightarrow$ *array-assn A*›
⟨*proof*⟩

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*)) ∈ $[pre\text{-}list\text{-}get]_a$ $(array\text{-}assn\ A)^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

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*))
∈ $[pre\text{-}list\text{-}get]_a$ $(arl\text{-}assn\ A)^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

lemma *uint32-nat-assn-plus*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ $[\lambda(m, n).\ m + n \le uint32\text{-}max]_a$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow$ *uint32-nat-assn*›
⟨*proof*⟩

lemma *uint32-nat-assn-one*:
‹(*uncurry0* (*return 1*), *uncurry0* (*RETURN 1*)) ∈ $unit\text{-}assn^k \rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

lemma *uint32-nat-assn-zero*:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN 0*)) ∈ $unit\text{-}assn^k \rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

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

lemma *uint32-nat-assn-zero-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint32-nat*)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

lemma *nat-assn-zero*:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN 0*)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *nat-assn*›
⟨*proof*⟩

lemma *one-uint32-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 1*), *uncurry0* (*RETURN one-uint32-nat*)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

lemma *uint32-nat-assn-less*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (<)), *uncurry* (*RETURN oo* (<))) $\in$
uint32-nat-assn^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

lemma *uint32-2-hnr*[*sepref-fr-rules*]: ‹(*uncurry0* (*return two-uint32*), *uncurry0* (*RETURN two-uint32-nat*)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

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

lemma *le-uint32-nat-hnr*:
‹(*uncurry* (*return oo* ($\lambda a\ b.$ *nat-of-uint32 a* < *b*)), *uncurry* (*RETURN oo* (<))) $\in$
uint32-nat-assn^k $*_a$ *nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

lemma *le-nat-uint32-hnr*:
‹(*uncurry* (*return oo* ($\lambda a\ b.$ *a* < *nat-of-uint32 b*)), *uncurry* (*RETURN oo* (<))) $\in$
nat-assn^k $*_a$ *uint32-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

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*)) $\in$
$[\lambda(m, n).\ m \geq n]_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow$ *nat-assn*›
⟨*proof*⟩

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*)) $\in$
$[\lambda(m, n).\ m \geq n]_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} $\rightarrow$ *uint32-nat-assn*›
⟨*proof*⟩

lemma *uint32-nat-assn-0-eq*: ‹*uint32-nat-assn 0 a* = ↑ (*a* = *0*)›
⟨*proof*⟩

lemma *uint32-nat-assn-nat-assn-nat-of-uint32*:
‹$uint32\text{-}nat\text{-}assn\ aa\ a = nat\text{-}assn\ aa\ (nat\text{-}of\text{-}uint32\ a)$›
⟨*proof*⟩

lemma *sum-mod-uint32-max*: ‹$(uncurry\ (return\ oo\ (+)),\ uncurry\ (RETURN\ oo\ sum\text{-}mod\text{-}uint32\text{-}max))$
$\in$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a$
$uint32\text{-}nat\text{-}assn$›
⟨*proof*⟩

lemma *le-uint32-nat-rel-hnr*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (\le)),\ uncurry\ (RETURN\ oo\ (\le))) \in$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

lemma *one-uint32-hnr*[*sepref-fr-rules*]:
‹$(uncurry0\ (return\ 1),\ uncurry0\ (RETURN\ one\text{-}uint32)) \in unit\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
⟨*proof*⟩

lemma *sum-uint32-assn*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (+)),\ uncurry\ (RETURN\ oo\ (+))) \in uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
⟨*proof*⟩

lemma *Suc-uint32-nat-assn-hnr*:
‹$(return\ o\ (\lambda n.\ n + 1),\ RETURN\ o\ Suc) \in [\lambda n.\ n < uint32\text{-}max]_a\ uint32\text{-}nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn$›
⟨*proof*⟩

lemma *minus-uint32-assn*:
‹$(uncurry\ (return\ oo\ (-)),\ uncurry\ (RETURN\ oo\ (-))) \in uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
⟨*proof*⟩

lemma *bitAND-uint32-nat-assn*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (AND)),\ uncurry\ (RETURN\ oo\ (AND))) \in$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a uint32\text{-}nat\text{-}assn$›
⟨*proof*⟩

lemma *bitAND-uint32-assn*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (AND)),\ uncurry\ (RETURN\ oo\ (AND))) \in$
$uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
⟨*proof*⟩

lemma *bitOR-uint32-nat-assn*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (OR)),\ uncurry\ (RETURN\ oo\ (OR))) \in$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow_a uint32\text{-}nat\text{-}assn$›
⟨*proof*⟩

lemma *bitOR-uint32-assn*[*sepref-fr-rules*]:
‹$(uncurry\ (return\ oo\ (OR)),\ uncurry\ (RETURN\ oo\ (OR))) \in$
$uint32\text{-}assn^k *_a uint32\text{-}assn^k \rightarrow_a uint32\text{-}assn$›
⟨*proof*⟩

lemma *uint32-nat-assn-mult*:
‹$(uncurry\ (return\ oo\ ((*))),\ uncurry\ (RETURN\ oo\ ((*)))) \in [\lambda(a,\ b).\ a * b \le uint32\text{-}max]_a$
$uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow uint32\text{-}nat\text{-}assn$›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*div*)), *uncurry* (*RETURN oo* (*div*))) ∈
$\textit{uint32-nat-assn}^k *_a \textit{uint32-nat-assn}^k \rightarrow_a \textit{uint32-nat-assn}$›
⟨*proof*⟩

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*
⟨*proof*⟩

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* (=))) ∈
$\textit{uint64-assn}^k *_a \textit{uint64-assn}^k \rightarrow_a \textit{bool-assn}$›
⟨*proof*⟩

lemma *op-eq-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* ((=) :: *uint64* ⇒ -)), *uncurry* (*RETURN oo* (=))) ∈
$\textit{uint64-nat-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow_a \textit{bool-assn}$›
⟨*proof*⟩

lemma *uint64-nat-assn-zero-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint64-nat*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint64-nat-assn}$›
⟨*proof*⟩

lemma *uint64-nat-assn-plus*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ $[\lambda(m, n).\ m + n \le \textit{uint64-max}]_a$
$\textit{uint64-nat-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow \textit{uint64-nat-assn}$›
⟨*proof*⟩

lemma *one-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 1*), *uncurry0* (*RETURN one-uint64-nat*)) ∈ $\textit{unit-assn}^k \rightarrow_a \textit{uint64-nat-assn}$›
⟨*proof*⟩

lemma *uint64-nat-assn-less*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (<)), *uncurry* (*RETURN oo* (<))) ∈
$\textit{uint64-nat-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow_a \textit{bool-assn}$›
⟨*proof*⟩

lemma *mult-uint64*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (∗)), *uncurry* (*RETURN oo* (∗)))

$\in$ *uint64-assn*^{k} $*_a$ *uint64-assn*^{k} $\rightarrow_a$ *uint64-assn*›
⟨*proof*⟩

lemma *shiftr-uint64* [*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (>>)), *uncurry* (*RETURN oo* (>>)))
$\in$ *uint64-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *uint64-assn*›
⟨*proof*⟩

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*)))
$\in [\lambda(a, b).\ a \leq \mathit{uint64\text{-}max} \wedge b \leq \mathit{uint64\text{-}max}]_a$
$\mathit{uint64\text{-}nat\text{-}assn}^k *_a \mathit{uint64\text{-}nat\text{-}assn}^k \rightarrow \mathit{uint64\text{-}nat\text{-}assn}$›

⟨*proof*⟩

lemma *two-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint64-nat*))
∈ *unit-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *bitOR-uint64-max-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*)))
∈ [λ(*a*, *b*). *a* ≤ *uint64-max* ∧ *b* ≤ *uint64-max*]$_a$
uint64-nat-assn^k $*_a$ *uint64-nat-assn*^{k} → *uint64-nat-assn*›
⟨*proof*⟩

lemma *fast-minus-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo fast-minus*), *uncurry* (*RETURN oo fast-minus*))
∈ [λ(*a*, *b*). *a* ≥ *b*]$_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *uint64-nat-assn*›
⟨*proof*⟩

lemma *fast-minus-uint64*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo fast-minus*), *uncurry* (*RETURN oo fast-minus*))
∈ [λ(*a*, *b*). *a* ≥ *b*]$_a$ *uint64-assn*^{k} $*_a$ *uint64-assn*^{k} → *uint64-assn*›
⟨*proof*⟩

lemma *minus-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (−)), *uncurry* (*RETURN oo* (−))) ∈
[λ(*a*, *b*). *a* ≥ *b*]$_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *uint64-nat-assn*›
⟨*proof*⟩

lemma *le-uint64-nat-assn-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (≤)), *uncurry* (*RETURN oo* (≤))) ∈ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

lemma *sum-mod-uint64-max-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo sum-mod-uint64-max*))
∈ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *zero-uint64-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint64*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint64-assn*›
⟨*proof*⟩

lemma *zero-uint32-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN zero-uint32*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-assn*›
⟨*proof*⟩

lemma *zero-uin64-hnr*: ‹(*uncurry0* (*return 0*), *uncurry0* (*RETURN 0*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint64-assn*›
⟨*proof*⟩

lemma *two-uin64-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint64*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint64-assn*›
⟨*proof*⟩

lemma *two-uint32-hnr*[*sepref-fr-rules*]:

‹(*uncurry0* (*return 2*), *uncurry0* (*RETURN two-uint32*)) ∈ *unit-assn*^{k} →$_a$ *uint32-assn*›
⟨*proof*⟩

lemma *sum-uint64-assn*:
‹(*uncurry* (*return oo* (+)), *uncurry* (*RETURN oo* (+))) ∈ *uint64-assn*^{k} ∗$_a$ *uint64-assn*^{k} →$_a$ *uint64-assn*›
⟨*proof*⟩

lemma *bitAND-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*))) ∈
uint64-nat-assn^k ∗$_a$ *uint64-nat-assn*^{k} →$_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *bitAND-uint64-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*AND*)), *uncurry* (*RETURN oo* (*AND*))) ∈
uint64-assn^k ∗$_a$ *uint64-assn*^{k} →$_a$ *uint64-assn*›
⟨*proof*⟩

lemma *bitOR-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*))) ∈
uint64-nat-assn^k ∗$_a$ *uint64-nat-assn*^{k} →$_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *bitOR-uint64-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*OR*)), *uncurry* (*RETURN oo* (*OR*))) ∈
uint64-assn^k ∗$_a$ *uint64-assn*^{k} →$_a$ *uint64-assn*›
⟨*proof*⟩

lemma *nat-of-uint64-mult-le*:
‹*nat-of-uint64 ai* ∗ *nat-of-uint64 bi* ≤ *uint64-max* ⟹
nat-of-uint64 (*ai* ∗ *bi*) = *nat-of-uint64 ai* ∗ *nat-of-uint64 bi*›
⟨*proof*⟩

lemma *uint64-nat-assn-mult*:
‹(*uncurry* (*return oo* ((∗))), *uncurry* (*RETURN oo* ((∗)))) ∈ [λ(*a*, *b*). *a* ∗ *b* ≤ *uint64-max*]$_a$
uint64-nat-assn^k ∗$_a$ *uint64-nat-assn*^{k} → *uint64-nat-assn*›
⟨*proof*⟩

lemma *uint64-max-uint64-nat-assn*:
‹(*uncurry0* (*return 18446744073709551615*), *uncurry0* (*RETURN uint64-max*)) ∈
unit-assn^k →$_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *uint64-max-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry0* (*return 18446744073709551615*), *uncurry0* (*RETURN uint64-max*)) ∈
unit-assn^k →$_a$ *nat-assn*›
⟨*proof*⟩

Conversions

From nat to 64 bits **lemma** *uint64-of-nat-conv-hnr*[*sepref-fr-rules*]:
‹(*return o uint64-of-nat*, *RETURN o uint64-of-nat-conv*) ∈
[λ*n*. *n* ≤ *uint64-max*]$_a$ *nat-assn*^{k} → *uint64-nat-assn*›
⟨*proof*⟩

From nat to 32 bits **lemma** *nat-of-uint32-spec-hnr*[*sepref-fr-rules*]:
‹(*return o uint32-of-nat*, *RETURN o nat-of-uint32-spec*) ∈

$[\lambda n.\ n \le$ *uint32-max*$]_a$ *nat-assn*^{k} $\to$ *uint32-nat-assn*›
⟨*proof*⟩

From 64 to nat bits **lemma** *nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
‹(*return o nat-of-uint64*, *RETURN o nat-of-uint64-conv*) ∈ *uint64-nat-assn*^{k} $\to_a$ *nat-assn*›
⟨*proof*⟩

lemma *nat-of-uint64*[*sepref-fr-rules*]:
‹(*return o nat-of-uint64*, *RETURN o nat-of-uint64*) ∈
(*uint64-assn*)^k $\to_a$ *nat-assn*›
⟨*proof*⟩

From 32 to nat bits **lemma** *nat-of-uint32-conv-hnr*[*sepref-fr-rules*]:
‹(*return o nat-of-uint32*, *RETURN o nat-of-uint32-conv*) ∈ *uint32-nat-assn*^{k} $\to_a$ *nat-assn*›
⟨*proof*⟩

lemma *convert-to-uint32-hnr*[*sepref-fr-rules*]:
‹(*return o uint32-of-nat*, *RETURN o convert-to-uint32*)
∈ $[\lambda n.\ n \le$ *uint32-max*$]_a$ *nat-assn*^{k} $\to$ *uint32-nat-assn*›
⟨*proof*⟩

From 32 to 64 bits **lemma** *uint64-of-uint32-hnr*[*sepref-fr-rules*]:
‹(*return o uint64-of-uint32*, *RETURN o uint64-of-uint32*) ∈ *uint32-assn*^{k} $\to_a$ *uint64-assn*›
⟨*proof*⟩

lemma *uint64-of-uint32-conv-hnr*[*sepref-fr-rules*]:
‹(*return o uint64-of-uint32*, *RETURN o uint64-of-uint32-conv*) ∈
uint32-nat-assn^k $\to_a$ *uint64-nat-assn*›
⟨*proof*⟩

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*) ∈
$[\lambda a.\ a \le$ *uint32-max*$]_a$ *uint64-nat-assn*^{k} $\to$ *uint32-nat-assn*›
⟨*proof*⟩

From nat to 32 bits **lemma** (**in** −) *uint32-of-nat*[*sepref-fr-rules*]:
‹(*return o uint32-of-nat*, *RETURN o uint32-of-nat*) ∈ $[\lambda n.\ n \le$ *uint32-max*$]_a$ *nat-assn*^{k} $\to$ *uint32-assn*›
⟨*proof*⟩

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* ⇒ *uint32*)]
itypeI[*Pure.of numeral TYPE* (*num* ⇒ *uint64*)]

lemma *id-uint32-const*[*id-rules*]: (*PR-CONST* (*a*::*uint32*)) $::_i$ *TYPE*(*uint32*) ⟨*proof*⟩
lemma *id-uint64-const*[*id-rules*]: (*PR-CONST* (*a*::*uint64*)) $::_i$ *TYPE*(*uint64*) ⟨*proof*⟩

lemma *param-uint32-numeral*[*sepref-import-param*]:
‹(*numeral n*, *numeral n*) ∈ *uint32-rel*›
⟨*proof*⟩

lemma *param-uint64-numeral*[*sepref-import-param*]:
‹(*numeral n*, *numeral n*) ∈ *uint64-rel*›
⟨*proof*⟩

locale *nat-of-uint64-loc* =
fixes *n* :: *num*
assumes *le-uint64-max*: ‹*numeral n* ≤ *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*›
⟨*proof*⟩
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*›
⟨*proof*⟩

lemma *uint32-max-uint32-nat-assn*:
‹(*uncurry0* (*return 4294967295*), *uncurry0* (*RETURN uint32-max*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-nat-assn*›
⟨*proof*⟩

lemma *minus-uint64-assn*:
‹(*uncurry* (*return oo* (−)), *uncurry* (*RETURN oo* (−))) ∈ *uint64-assn*^{k} $*_a$ *uint64-assn*^{k} $\rightarrow_a$ *uint64-assn*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *three-uint32-hnr*:
‹(*uncurry0* (*return 3*), *uncurry0* (*RETURN* (*three-uint32* :: *uint32*))) ∈ *unit-assn*^{k} $\rightarrow_a$ *uint32-assn*›
⟨*proof*⟩

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*›

⟨*proof*⟩

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*
⟨*proof*⟩

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*) $\in [\lambda xs.\ xs \neq []]_a\ (arl\text{-}assn\ A)^d \rightarrow arl\text{-}assn\ A$›
⟨*proof*⟩

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′*›
⟨*proof*⟩

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*)) ∈
$[\lambda((x, L), L').\ L < length\ x \wedge L' < length\ (x\ !\ L)]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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›
⟨*proof*⟩

lemma *array-replicate-custom-hnr-u*[*sepref-fr-rules*]:
‹CONSTRAINT is-pure A ⟹
(uncurry (λn. Array.new (nat-of-uint32 n)), uncurry (RETURN ∘∘ op-array-replicate)) ∈
uint32-nat-assn^k $*_a$ A^k $\rightarrow_a$ array-assn A›
⟨*proof*⟩

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*⟩

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*⟩

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 ≡ λa 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' <λr. arlO-assn-except (array-assn R) [i] a ai
(λr'. array-assn R (a ! i) r * ↑(r = r' ! i))>›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *arl-get-hnr-u*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry arl-get-u*, *uncurry* (*RETURN* ∘∘ *op-list-get*))
∈ $[pre\text{-}list\text{-}get]_a$ $(arl\text{-}assn\ A)^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

lemma *nth-nat-of-uint32-nth′*: ‹*Array.nth x* (*nat-of-uint32 L*) = *Array.nth′ x* (*integer-of-uint32 L*)›
⟨*proof*⟩

lemma *nth-aa-u-code*[*code*]:
‹*nth-aa-u x L L′* = *nth-u-code x L* ⤜ (λ*x*. *arl-get x L′* ⤜ *return*)›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *array-get-hnr-u64*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry nth-u64-code*,
uncurry (*RETURN* ∘∘ *op-list-get*)) ∈ [*pre-list-get*]$_a$ (*array-assn A*)^k $*_a$ *uint64-nat-assn*^{k} → *A*›
⟨*proof*⟩

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-list-set*]$_a$ (*array-assn A*)^d $*_a$ *uint32-nat-assn*^{k} $*_a$ A^k → *array-assn A*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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
}›
⟨*proof*⟩

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 ⟹*
(*uncurry2 arl-set-u*, *uncurry2* (*RETURN ∘∘∘ op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(arl\text{-}assn\ A)^d *_a uint32\text{-}nat\text{-}assn^k *_a A^k \rightarrow arl\text{-}assn\ A$›
⟨*proof*⟩

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 ⟹*
(*uncurry2 heap-array-set-u64*, *uncurry2* (*RETURN ∘∘∘ op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(array\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a A^k \rightarrow array\text{-}assn\ A$›
⟨*proof*⟩

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 ⟹*
(*uncurry2 arl-set-u64*, *uncurry2* (*RETURN ∘∘∘ op-list-set*)) ∈
$[pre\text{-}list\text{-}set]_a$ $(arl\text{-}assn\ A)^d *_a uint64\text{-}nat\text{-}assn^k *_a A^k \rightarrow arl\text{-}assn\ A$›
⟨*proof*⟩

lemma *nth-nat-of-uint64-nth'*: ‹*Array.nth x (nat-of-uint64 L) = Array.nth' x (integer-of-uint64 L)*›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*))
∈ $[pre\text{-}list\text{-}get]_a$ $(arl\text{-}assn\ A)^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow A$›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}rll\ l\ i]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*) ›
⟨*proof*⟩

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′ ≡ *λa i x.*
  Array.nth′ a (integer-of-uint32 i) ≫=
  (λj. arl-append j x ≫=
    (λa′. 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*›
  ⟨*proof*⟩

lemma *append-aa-hnr-u*:
  fixes *R* :: ‹*′a ⇒ ′b* :: {*heap, default*} ⇒ *assn*›
  assumes *p*: ‹*is-pure R*›
  shows
    ‹*(uncurry2 (λxs i. append-el-aa xs (nat-of-uint32 i)), uncurry2 (RETURN ∘∘∘ (λxs i. append-ll xs (nat-of-uint32 i))))* ∈
      $[\lambda((l,i),x).\ nat\text{-}of\text{-}uint32\ i < length\ l]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a uint32\text{-}assn^k *_a R^k \rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
⟨*proof*⟩

lemma *append-el-aa-hnr′*[*sepref-fr-rules*]:
  shows ‹*(uncurry2 append-el-aa-u′, uncurry2 (RETURN ooo append-ll))*
    $\in [\lambda((W,L),\ j).\ L < length\ W]_a$
      $(arrayO\text{-}assn\ (arl\text{-}assn\ nat\text{-}assn))^d *_a uint32\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ nat\text{-}assn))$›
    (**is** ‹$?a \in [?pre]_a\ ?init \rightarrow ?post$›)
  ⟨*proof*⟩

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))*
    $\in [\lambda((W,L),\ j).\ L < length\ W]_a$
      $(arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a uint32\text{-}nat\text{-}assn^k *_a R^k \rightarrow$
      $(arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
    (**is** ‹$?a \in [?pre]_a\ ?init \rightarrow ?post$›)
  ⟨*proof*⟩

lemma *append-el-aa-u′-code*[*code*]:
  append-el-aa-u′ = (λa i x. nth-u-code a i ≫=
    (λj. arl-append j x ≫=
    (λa′. heap-array-set′-u a i a′ ≫= *(λ-. return a))))*
  ⟨*proof*⟩

definition *update-raa-u32* **where**
‹*update-raa-u32 a i j y = do* {
  x ← arl-get-u a i;
  Array.upd j y x ≫= *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) ∈ 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 ∗ (∃$_A$x. arlO-assn (array-assn R) x r ∗ ↑ (x = update-rll a bb ba b))>$_t$›
⟨proof⟩

lemma *update-raa-u32-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-raa-u32*, *uncurry3* (*RETURN oooo update-rll*)) ∈ [λ(((l,i), j), x). i < length l ∧ j < length-rll l i]$_a$ (arlO-assn (array-assn R))^d ∗$_a$ uint32-nat-assn^k ∗$_a$ nat-assn^k ∗$_a$ R^k → (arlO-assn (array-assn R))›
⟨proof⟩

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*) ∈ *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 ∗ (∃$_A$x. arrayO-assn (arl-assn R) x r ∗ ↑ (x = update-ll a bb ba b))>$_t$›
solve-direct
⟨proof⟩

lemma *update-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa-u*, *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 ∗$_a$ uint32-nat-assn^k ∗$_a$ nat-assn^k ∗$_a$ R^k → (arrayO-assn (arl-assn R))›
⟨proof⟩

Length

32-bits **definition** (**in** −)*length-u-code* **where**
‹*length-u-code C* = *do* { *n* ← *Array.len C*; *return* (*uint32-of-nat n*)}›

lemma (**in** −)*length-u-hnr*[*sepref-fr-rules*]:
‹(*length-u-code*, *RETURN o length-uint32-nat*) ∈ [λC. length C ≤ uint32-max]$_a$ (array-assn R)^k → uint32-nat-assn›
⟨proof⟩

definition *length-arl-u-code* :: ‹(′*a*::*heap*) *array-list* ⇒ *uint32 Heap*› **where**
‹*length-arl-u-code xs* = *do* {
n ← *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*) ∈
[λxs. length xs ≤ uint32-max]$_a$ (arl-assn R)^k → uint32-nat-assn›
⟨proof⟩

64-bits **definition** (**in** −)*length-u64-code* **where**
‹*length-u64-code C* = *do* { *n* ← *Array.len C*; *return* (*uint64-of-nat n*)}›

lemma (**in** −)*length-u64-hnr*[*sepref-fr-rules*]:
‹(*length-u64-code*, *RETURN o length-uint64-nat*)
∈ [λC. length C ≤ uint64-max]$_a$ (array-assn R)^k → uint64-nat-assn›
⟨proof⟩

Length for arrays in arrays

32-bits **definition** (**in** −)*length-aa-u* :: ‹(′*a*::*heap array-list*) *array* ⇒ *uint32* ⇒ *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* ⤜ *arl-length*›
⟨*proof*⟩

lemma *length-aa-u-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-aa-u*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈
$[\lambda(xs, i).\ i < length\ xs]_a$ (*arrayO-assn* (*arl-assn R*))^k $*_a$ *uint32-nat-assn*^{k} → *nat-assn*›
⟨*proof*⟩

definition *length-raa-u* :: ‹′*a*::*heap arrayO-raa* ⇒ *nat* ⇒ *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*)}›
⟨*proof*⟩

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*))>$_t$›
⟨*proof*⟩

lemma *length-raa-u-hnr*[*sepref-fr-rules*]:
shows ‹(*uncurry length-raa-u*, *uncurry* (*RETURN* ∘∘ *length-rll-n-uint32*)) ∈
$[\lambda(xs, i).\ i < length\ xs \wedge length\ (xs\ !\ i) \le uint32\text{-}max]_a$
(*arlO-assn* (*array-assn R*))^k $*_a$ *nat-assn*^{k} → *uint32-nat-assn*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *length-aa-u64-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-aa-u64*, *uncurry* (*RETURN* ∘∘ *length-ll*)) ∈
$[\lambda(xs, i).\ i < length\ xs]_a$ (*arrayO-assn* (*arl-assn R*))^k $*_a$ *uint64-nat-assn*^{k} → *nat-assn*›
⟨*proof*⟩

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)}›
⟨proof⟩

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)) ∈
$[\lambda(xs, i).\ i < length\ xs \wedge length\ (xs\ !\ i) \leq uint64\text{-}max]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨proof⟩

Delete at index

definition *delete-index-and-swap-aa* **where**
‹delete-index-and-swap-aa xs i j = do {
x ← last-aa xs i;
xs ← 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))
$\in [\lambda((l,i), j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k$
$\rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
⟨proof⟩

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›
⟨proof⟩

lemma *length-delete-index-and-swap-ll*[*simp*]:
‹length (delete-index-and-swap-ll s i j) = length s›
⟨proof⟩

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 ← nth-u-code a i;
a′ ← arl-butlast x;

Array-upd-u i a′ a
}› — Replace the *i*-th element by the itself execpt the last element.
⟨*proof*⟩

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
}›
⟨*proof*⟩

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*))
$\in$ [λ((*l*,*i*), *j*). *i* < *length l* ∧ *j* < *length-ll l i*]$_a$ (*arrayO-assn* (*arl-assn R*))^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *nat-assn*^{k}
→ (*arrayO-assn* (*arl-assn R*))›
⟨*proof*⟩

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*)) ∈
[λ((*xs*, *i*), *j*). *i* < *length xs* ∧ *j* < *length xs*]$_a$
(*array-assn R*)^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint32-nat-assn*^{k} → *array-assn R*›
⟨*proof*⟩

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*)) ∈

[λ((xs, i), j). i < length xs ∧ j < length xs]$_a$
(array-assn R)^d $*_a$ uint64-nat-assn^k $*_a$ uint64-nat-assn^k → array-assn R›
⟨proof⟩

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)) ∈
[λ(((xs, k), i), j). k < length xs ∧ i < length-rll xs k ∧ j < length-rll xs k]$_a$
(arlO-assn (array-assn R))^d $*_a$ nat-assn^k $*_a$ uint64-nat-assn^k $*_a$ uint64-nat-assn^k →
(arlO-assn (array-assn R))›
⟨proof⟩

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)) ∈
[λ((xs, i), j). i < length xs ∧ j < length xs]$_a$
(arl-assn R)^d $*_a$ uint32-nat-assn^k $*_a$ uint32-nat-assn^k → arl-assn R›
⟨proof⟩

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›
⟨proof⟩

lemma *shorten-take-aa-u32-hnr*[sepref-fr-rules]:
‹(uncurry2 shorten-take-aa-u32, uncurry2 (RETURN ooo shorten-take-ll)) ∈
[λ((L, j), W). j ≤ length (W ! L) ∧ L < length W]$_a$

$uint32\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d \to arrayO\text{-}assn\ (arl\text{-}assn\ R)$›
⟨*proof*⟩

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*)) ∈
$[\lambda((xs,\ i),\ j).\ i < length\ xs \land j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k \to R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((xs,\ i),\ j).\ i < length\ xs \land j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((xs,\ i),\ j).\ i < length\ xs \land j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \to R$›
⟨*proof*⟩

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′*›
⟨*proof*⟩

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*)) ∈
$[\lambda((x, L), L'). L < length\ x \wedge L' < length\ (x\ !\ L)]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a uint32\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \to R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((xs, i), j). i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint32\text{-}nat\text{-}assn^k \to R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j). i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \to R$›
⟨*proof*⟩

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*)) ∈
$[\lambda((xs, i), j). i < length\ xs \wedge j < length\ (xs\ !i)]_a$
$(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \to R$›
⟨*proof*⟩

lemma *nth-aa-i64-u64-code*[*code*]:

‹nth-aa-i64-u64 x L L′ = nth-u64-code x L ≫= (λx. arl-get-u64 x L′ ≫= return)›
⟨proof⟩

lemma *nth-aa-i64-u32-code*[*code*]:
‹nth-aa-i64-u32 x L L′ = nth-u64-code x L ≫= (λx. arl-get-u x L′ ≫= return)›
⟨proof⟩

lemma *nth-aa-i32-u64-code*[*code*]:
‹nth-aa-i32-u64 x L L′ = nth-u-code x L ≫= (λx. arl-get-u64 x L′ ≫= return)›
⟨proof⟩

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)}›
⟨proof⟩

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)))>$_t$›
⟨proof⟩

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›
⟨proof⟩

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)}›
⟨proof⟩

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)))>$_t$›
⟨proof⟩

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›
⟨proof⟩

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)}›
⟨proof⟩

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)) ∈
[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⟩

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›
⟨proof⟩

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⟩

lemma *length-raa-i32-u-hnr*[sepref-fr-rules]:
shows ‹(uncurry length-raa-i32-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 $*_a$ uint32-nat-assn^k → uint32-nat-assn›
⟨proof⟩

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 >>= (λn. 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›
⟨*proof*⟩

lemma *length-aa-u64-o64-hnr*[*sepref-fr-rules*]:
‹(uncurry length-aa-u64-o64, uncurry (RETURN ∘∘ length-ll)) ∈
[λ(xs, i). i < length xs ∧ length (xs ! i) ≤ uint64-max]$_a$
(arrayO-assn (arl-assn R))^k $*_a$ uint64-nat-assn^k → uint64-nat-assn›
⟨*proof*⟩

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 >>= (λn. 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›
⟨*proof*⟩

lemma *length-aa-u32-o64-hnr*[*sepref-fr-rules*]:
‹(uncurry length-aa-u32-o64, uncurry (RETURN ∘∘ length-ll)) ∈
[λ(xs, i). i < length xs ∧ length (xs ! i) ≤ uint64-max]$_a$
(arrayO-assn (arl-assn R))^k $*_a$ uint32-nat-assn^k → uint64-nat-assn›
⟨*proof*⟩

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)>$_t$›
⟨*proof*⟩

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›
⟨*proof*⟩

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*⟩

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*)) ∈
    $[\lambda(xs, i).\ i < length\ xs \wedge length\ (xs\ !\ i) \le uint64\text{-}max]_a$
      $(arlO\text{-}assn\ (array\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow uint64\text{-}nat\text{-}assn$›
⟨*proof*⟩

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*) ∈ $[\lambda xs.\ length\ xs \le uint32\text{-}max]_a$ $(arlO\text{-}assn\ R)^k \rightarrow$ *uint32-nat-assn*›
  ⟨*proof*⟩

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*) ∈
    $[\lambda xs.\ length\ xs \le uint64\text{-}max]_a$ $(arl\text{-}assn\ R)^k \rightarrow uint64\text{-}nat\text{-}assn$›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

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
  }›
  ⟨*proof*⟩

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;

$a' \leftarrow$ *arl-butlast x*;
Array-upd-u64 i a' a
}› — Replace the *i*-th element by the itself except the last element.
⟨*proof*⟩

lemma *delete-index-and-swap-aa-i64-code*[*code*]:
‹*delete-index-and-swap-aa-i64 xs i j = do* {
$x \leftarrow$ *last-aa-u64 xs i*;
$xs \leftarrow$ *update-aa-u64 xs i j x*;
set-butlast-aa-u64 xs i
}›
⟨*proof*⟩

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*))
$\in [\lambda((l,i), j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a uint64\text{-}nat\text{-}assn^k *_a nat\text{-}assn^k$
$\rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
⟨*proof*⟩

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* {
$x \leftarrow$ *nth-u-code a i*;
$a' \leftarrow$ *arl-set-u64 x j y*;
Array-upd-u i a' a
}›
⟨*proof*⟩

lemma *delete-index-and-swap-aa-i32-u64-code*[*code*]:
‹*delete-index-and-swap-aa-i32-u64 xs i j = do* {
$x \leftarrow$ *last-aa-u xs i*;
$xs \leftarrow$ *update-aa-u32-i64 xs i j x*;
set-butlast-aa-u xs i
}›
⟨*proof*⟩

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*))
$\in [\lambda((l,i), j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a\ (arrayO\text{-}assn\ (arl\text{-}assn\ R))^d *_a$
$uint32\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k$
$\rightarrow (arrayO\text{-}assn\ (arl\text{-}assn\ R))$›
⟨*proof*⟩

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*)) ∈
[λ(((*xs*, *k*), *i*), *j*). *k* < *length xs* ∧ *i* < *length-rll xs k* ∧ *j* < *length-rll xs k*]$_a$
(*arlO-assn* (*array-assn R*))^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} →
(*arlO-assn* (*array-assn R*))›
⟨*proof*⟩

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-assn uint64-nat-assn*)^k $\rightarrow_a$ *array-assn nat-assn*›
⟨*proof*⟩

lemma *array-nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
‹(*array-nat-of-uint64-code*, (*RETURN* ∘ *array-nat-of-uint64-conv*))
∈ (*array-assn uint64-nat-assn*)^k $\rightarrow_a$ *array-assn nat-assn*›
⟨*proof*⟩

sepref-definition *array-uint64-of-nat-code*
is *array-uint64-of-nat*
:: ‹[λ*xs*. ∀ *a*∈*set xs*. *a* ≤ *uint64-max*]$_a$
(*array-assn nat-assn*)^k → *array-assn uint64-nat-assn*›
⟨*proof*⟩

lemma *array-uint64-of-nat-conv-alt-def*:
‹*array-uint64-of-nat-conv* = *map uint64-of-nat-conv*›
⟨*proof*⟩

lemma *array-uint64-of-nat-conv-hnr*[*sepref-fr-rules*]:
‹(*array-uint64-of-nat-code*, (*RETURN* ∘ *array-uint64-of-nat-conv*))
∈ [λ*xs*. ∀ *a*∈*set xs*. *a* ≤ *uint64-max*]$_a$
(*array-assn nat-assn*)^k → *array-assn uint64-nat-assn*›
⟨*proof*⟩

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-list-swap*]$_a$ (*arl-assn A*)^d $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *arl-assn A*›
⟨*proof*⟩

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*) ∈
[λ*xs*. *xs* ≠ []]$_a$ (*arl-assn R*)^d → *arl-assn R*›
⟨*proof*⟩

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* ∗ (∃$_A$*x*. *arrayO-assn* (*arl-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
⟨*proof*⟩

lemma *update-aa-u32-i64-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa-u32-i64*, *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 $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$ R^k → (*arrayO-assn* (*arl-assn R*))›
⟨*proof*⟩

lemma *min-uint64-nat-assn*:
‹(*uncurry* (*return oo min*), *uncurry* (*RETURN oo min*)) ∈ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma *nat-of-uint64-shiftl*: ‹*nat-of-uint64* (*xs* >> *a*) = *nat-of-uint64 xs* >> *a*›
⟨*proof*⟩

lemma *bit-lshift-uint64-nat-assn*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (>>)), *uncurry* (*RETURN oo* (>>))) ∈
uint64-nat-assn^k $*_a$ *nat-assn*^{k} $\rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma [*code*]: *uint32-max-uint32* = *4294967295*
⟨*proof*⟩

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* $l \equiv \lambda(a,n).\ \exists_A l'.\ a \mapsto_a l' * \uparrow(\textit{nat-of-uint64}\ n \leq \textit{length}\ l' \wedge l = \textit{take}\ (\textit{nat-of-uint64}\ n)\ l' \wedge \textit{length}\ l'>0 \wedge \textit{nat-of-uint64}\ n \leq \textit{uint64-max} \wedge \textit{length}\ l' \leq \textit{uint64-max})$

lemma *is-array-list64-prec*[*safe-constraint-rules*]: *precise is-array-list64*
⟨*proof*⟩

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 []>
⟨proof⟩

lemma arl64-empty-sz-rule[sep-heap-rules]: < emp > arl64-empty-sz N <is-array-list64 []>
⟨proof⟩

lemma arl64-copy-rule[sep-heap-rules]: < is-array-list64 l a > arl64-copy a <λr. is-array-list64 l a * is-array-list64 l r>
⟨proof⟩
lemma [simp]: ‹nat-of-uint64 uint64-max-uint64 = uint64-max›
⟨proof⟩
lemma ‹2 * (uint64-max div 2) = uint64-max − 1›
⟨proof⟩

lemma nat-of-uint64-0-iff: ‹nat-of-uint64 x2 = 0 ⟷ x2 = 0›
⟨proof⟩

lemma arl64-append-rule[sep-heap-rules]:
assumes ‹length l < uint64-max›
shows < is-array-list64 l a >
arl64-append a x
<λa. is-array-list64 (l@[x]) a >$_t$
⟨proof⟩

lemma arl64-length-rule[sep-heap-rules]:
<is-array-list64 l a>
arl64-length a
<λr. is-array-list64 l a * ↑(nat-of-uint64 r=length l)>
⟨proof⟩

lemma arl64-is-empty-rule[sep-heap-rules]:
<is-array-list64 l a>
arl64-is-empty a
<λr. is-array-list64 l a * ↑(r⟷(l=[]))>
⟨proof⟩

lemma arl64-last-rule[sep-heap-rules]:
l≠[] ⟹
<is-array-list64 l a>
arl64-last a
<λr. is-array-list64 l a * ↑(r=last l)>

⟨proof⟩

lemma arl64-get-rule[sep-heap-rules]:
  i<length l ⟹ (i', i) ∈ uint64-nat-rel ⟹
  <is-array-list64 l a>
    arl64-get a i'
  <λr. is-array-list64 l a ∗ ↑(r=l!i)>
  ⟨proof⟩

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])>
  ⟨proof⟩

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)
  ⟨proof⟩

lemma arl64-assn-comp': hr-comp (arl64-assn id-assn) (⟨B⟩list-rel) = arl64-assn (pure B)
  ⟨proof⟩

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
    ⟨proof⟩
  sepref-decl-impl (no-register) arl64-empty: arl64-empty-hnr-aux ⟨proof⟩

  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
    ⟨proof⟩

  sepref-decl-impl (no-register) arl64-empty-sz: arl64-empty-sz-hnr-aux ⟨proof⟩

  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
    ⟨proof⟩
  sepref-decl-impl arl64-copy: arl64-copy-hnr-aux ⟨proof⟩

  lemma arl64-append-hnr-aux: (uncurry arl64-append,uncurry (RETURN oo op-list-append)) ∈ $[\lambda(xs, x).\ length\ xs < uint64\text{-}max]_a\ (is\text{-}array\text{-}list64^d *_a id\text{-}assn^k) \rightarrow$ is-array-list64
    ⟨proof⟩

sepref-decl-impl *arl64-append*: *arl64-append-hnr-aux*
⟨*proof*⟩

lemma *arl64-length-hnr-aux*: (*arl64-length*,*RETURN o op-list-length*) ∈ $is\text{-}array\text{-}list64^k \rightarrow_a uint64\text{-}nat\text{-}assn$
⟨*proof*⟩
sepref-decl-impl *arl64-length*: *arl64-length-hnr-aux* ⟨*proof*⟩

lemma *arl64-is-empty-hnr-aux*: (*arl64-is-empty*,*RETURN o op-list-is-empty*) ∈ $is\text{-}array\text{-}list64^k \rightarrow_a bool\text{-}assn$
⟨*proof*⟩
sepref-decl-impl *arl64-is-empty*: *arl64-is-empty-hnr-aux* ⟨*proof*⟩

lemma *arl64-last-hnr-aux*: (*arl64-last*,*RETURN o op-list-last*) ∈ $[pre\text{-}list\text{-}last]_a\ is\text{-}array\text{-}list64^k \rightarrow id\text{-}assn$
⟨*proof*⟩
sepref-decl-impl *arl64-last*: *arl64-last-hnr-aux* ⟨*proof*⟩

lemma *arl64-get-hnr-aux*: (*uncurry arl64-get*,*uncurry* (*RETURN oo op-list-get*)) ∈ $[\lambda(l,i).\ i<length\ l]_a\ (is\text{-}array\text{-}list64^k *_a uint64\text{-}nat\text{-}assn^k) \rightarrow id\text{-}assn$
⟨*proof*⟩
sepref-decl-impl *arl64-get*: *arl64-get-hnr-aux* ⟨*proof*⟩

lemma *arl64-set-hnr-aux*: (*uncurry2 arl64-set*,*uncurry2* (*RETURN ooo op-list-set*)) ∈ $[\lambda((l,i),\text{-}).\ i<length\ l]_a\ (is\text{-}array\text{-}list64^d *_a uint64\text{-}nat\text{-}assn^k *_a id\text{-}assn^k) \rightarrow is\text{-}array\text{-}list64$
⟨*proof*⟩
sepref-decl-impl *arl64-set*: *arl64-set-hnr-aux* ⟨*proof*⟩

sepref-definition *arl64-swap* **is** *uncurry2 mop-list-swap* :: $((arl64\text{-}assn\ id\text{-}assn)^d *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a arl64\text{-}assn\ id\text{-}assn)$
⟨*proof*⟩
sepref-decl-impl (*ismop*) *arl64-swap*: *arl64-swap.refine* ⟨*proof*⟩
end

interpretation *arl64*: *list-custom-empty arl64-assn A arl64-empty op-arl64-empty*
⟨*proof*⟩

lemma [*def-pat-rules*]: *op-arl64-empty-sz*$*N* ≡ *UNPROTECT* (*op-arl64-empty-sz N*) ⟨*proof*⟩

interpretation *arl64-sz*: *list-custom-empty arl64-assn A arl64-empty-sz N PR-CONST* (*op-arl64-empty-sz N*)
⟨*proof*⟩

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*) ∈ $(arl64\text{-}assn\ R)^d \rightarrow_a arl\text{-}assn\ R$›
⟨*proof*⟩

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*)) ∈
$[\lambda(n, xs).\ n \leq length\ xs]_a$ *uint64-nat-assn*^{k} $*_a$ (*arl64-assn R*)^d → *arl64-assn R*›
⟨*proof*⟩
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*) ∈ $[\lambda n.\ length\ n \leq uint64\text{-}max]_a$ (*arl-assn R*)^d → *arl64-assn R*›
⟨*proof*⟩

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*›
⟨*proof*⟩

sepref-definition *arl-nat-of-uint64-code*
is *array-nat-of-uint64*
:: ‹(*arl-assn uint64-nat-assn*)^k $\rightarrow_a$ *arl-assn nat-assn*›
⟨*proof*⟩

lemma *arl-nat-of-uint64-conv-hnr*[*sepref-fr-rules*]:
‹(*arl-nat-of-uint64-code*, (*RETURN* ∘ *arl-nat-of-uint64-conv*))
∈ (*arl-assn uint64-nat-assn*)^k $\rightarrow_a$ *arl-assn nat-assn*›
⟨*proof*⟩

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*

<λp. arrayO-except-assn (arl64-assn R) [b] xs a (λp′. ↑(p=p′!b))∗
arl64-assn R (xs ! b) (p)>›
⟨proof⟩

lemma *arl64-length*[*sep-heap-rules*]:
‹<arl64-assn R b a> arl64-length a< λr. arl64-assn R b a ∗ ↑(nat-of-uint64 r = length b)>›
⟨proof⟩

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′
<λr. arrayO-assn (arl64-assn R) xs a ∗ ↑ (nat-of-uint64 r = length-ll xs b)>$_t$›
⟨proof⟩

lemma *length-aa64-hnr*[*sepref-fr-rules*]: ‹(uncurry length-aa64, uncurry (RETURN ∘∘ length-ll)) ∈
[λ(xs, i). i < length xs]$_a$ (arrayO-assn (arl64-assn R))^k ∗$_a$ uint64-nat-assn^k → uint64-nat-assn›
⟨proof⟩

lemma *arl64-get-hnr*[*sep-heap-rules*]:
assumes ‹(n′, n) ∈ uint64-nat-rel› **and** ‹n < length a› **and** ‹CONSTRAINT is-pure R›
shows ‹<arl64-assn R a b>
arl64-get b n′
<λr. arl64-assn R a b ∗ R (a ! n) r>›
⟨proof⟩

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)) ∈
[λ((l,i),j). i < length l ∧ j < length-ll l i]$_a$
(arrayO-assn (arl64-assn R))^k ∗$_a$ nat-assn^k ∗$_a$ uint64-nat-assn^k → R›
⟨proof⟩

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⟩

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⟩

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⟩

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*. ∃$_A$*x*. *R b bi* ∗ *arrayO-assn* (*arl64-assn R*) *x r* ∗ *true* ∗
            ↑ (*x* = *a*[*bb* := *xu*])>›
⟨proof⟩

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*
    <λ*r*. *R b bi* ∗ (∃$_A$*x*. *arrayO-assn* (*arl64-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
  ⟨proof⟩

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*))›
⟨*proof*⟩

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*
<λ*r*. *arl64-assn R ai a* ∗ *R* (*last ai*) *r*>$_t$›
⟨*proof*⟩

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*⟩

lemma *last-aa-hnr*[*sepref-fr-rules*]:
assumes *p*: ‹*is-pure R*›
shows ‹(*uncurry last-aa64*, *uncurry* (*RETURN oo last-ll*)) ∈
[λ(*l*,*i*). *i* < *length l* ∧ *l* ! *i* ≠ []]$_a$ (*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint64-nat-assn*^{k} → *R*›
⟨*proof*⟩

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*. ∃$_A$*x*. *arrayO-assn* (*arl64-assn R*) *aa a* ∗
(*R x r* ∗
↑ (*x* = *nth-ll aa b ba*)) ∗
true>›

⟨proof⟩

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*. ∃$_A$*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⟩

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*])>
⟨proof⟩

lemma *swap-aa-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 swap-aa64*, *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* (*arl64-assn R*))^d ∗$_a$ *nat-assn*^{k} ∗$_a$ *uint64-nat-assn*^{k} ∗$_a$ *uint64-nat-assn*^{k} → (*arrayO-assn* (*arl64-assn R*))›
⟨proof⟩

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 →$_a$ *arrayO-assn* (*arl64-assn R*)›
⟨proof⟩

sepref-definition
arrayO-ara-empty-sz-code
is *RETURN o arrayO-ara-empty-sz*
:: ‹*nat-assn*^{k} →$_a$ *arrayO-assn* (*arl64-assn* (*R*::′*a* ⇒ ′*b*::{*heap*, *default*} ⇒ *assn*))›
⟨proof⟩

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*›
⟨proof⟩

lemma *arrayO-raa-empty-sz-init-lrl*[*sepref-fr-rules*]:
‹(*arrayO-ara-empty-sz-code*, *RETURN o init-lrl64*) ∈
nat-assn^k →$_a$ *arrayO-assn* (*arl64-assn R*)›
⟨proof⟩

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*
(λ*r*′. ↑ ((*aaa*, *n*) = *r*′ ! *a*)) ∗ *arl64-assn R* (*b* ! *a*) (*aaa*, *n*)>
Array.upd a (*aaa*, *ba*′) *bi*
<λ*r*. $\exists_A$*x*. *arrayO-assn* (*arl64-assn R*) *x r* ∗ *true* ∗
↑ (*x* = *b*[*a* := *take ba* (*b* ! *a*)])>›
⟨*proof*⟩

lemma *shorten-take-aa-hnr*[*sepref-fr-rules*]:
‹(*uncurry2 shorten-take-aa64*, *uncurry2* (*RETURN ooo shorten-take-ll*)) ∈
[λ((*L*, *j*), *W*). *j* ≤ *length* (*W* ! *L*) ∧ *L* < *length W*]$_a$
nat-assn^k $*_a$ *uint64-nat-assn*^{k} $*_a$ (*arrayO-assn* (*arl64-assn R*))^d → *arrayO-assn* (*arl64-assn R*)›
⟨*proof*⟩

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* ∧ *L*′ < *length* (*x* ! *L*)]$_a$
(*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
⟨*proof*⟩

lemma *nth-aa64-u-code*[*code*]:
‹*nth-aa64-u x L L*′ = *nth-u-code x L* ≫= (λ*x*. *arl64-get x L*′ ≫= *return*)›
⟨*proof*⟩

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* ∧ *j* < *length-rll l i*]$_a$
(*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
⟨*proof*⟩

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* ∧ *j* < *length-rll l i*]$_a$
(*arrayO-assn* (*arl64-assn R*))^k $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} → *R*›
⟨*proof*⟩

definition *append64-el-aa32* :: ('*a*::{*default*,*heap*} *array-list64*) *array* ⇒
uint32 ⇒ '*a* ⇒ ('*a array-list64*) *array Heap***where**
append64-el-aa32 ≡ λ*a 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*)) ∈
[λ((*l*,*i*),*x*). *i* < *length l* ∧ *length* (*l* ! *i*) < *uint64-max*]$_a$ (*arrayO-assn* (*arl64-assn R*))^d $*_a$ *uint32-nat-assn*^{k} $*_a$ *R*^{k} → (*arrayO-assn* (*arl64-assn R*))›
⟨*proof*⟩

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' ← *arl64-set x j y*;
Array-upd-u i a' a
}›
⟨*proof*⟩

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*) ∈ *uint64-nat-rel*› ‹(*bb'*, *bb*) ∈ *uint32-nat-rel*›
shows ‹<*R b bi* ∗ *arrayO-assn* (*arl64-assn R*) *a ai*> *update-aa64-u32 ai bb' ba' bi*
<λ*r*. *R b bi* ∗ (∃$_A$*x*. *arrayO-assn* (*arl64-assn R*) *x r* ∗ ↑ (*x* = *update-ll a bb ba b*))>$_t$›
⟨*proof*⟩

lemma *update-aa64-u32-hnr*[*sepref-fr-rules*]:
assumes ‹*is-pure R*›
shows ‹(*uncurry3 update-aa64-u32*, *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 $*_a$ *uint32-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$ *R*^{k} → (*arrayO-assn* (*arl64-assn R*))›
⟨*proof*⟩

definition *nth-aa64-u64* **where**
‹*nth-aa64-u64 xs i j* = *do* {
x ← *nth-u64-code xs i*;
y ← *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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl64\text{-}assn\ R))^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*)>
⟨*proof*⟩

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*)>
⟨*proof*⟩

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*)) ∈
$[\lambda((l,i),j).\ i < length\ l \wedge j < length\text{-}ll\ l\ i]_a$
$(arrayO\text{-}assn\ (arl64\text{-}assn\ R))^k *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow R$›
⟨*proof*⟩

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*
<λ*r*. *arrayO-assn* (*arl64-assn R*) *xs a* ∗ ↑ (*r* = *length-ll xs b*)>$_t$›
⟨*proof*⟩

lemma *length-aa64-nat-hnr*[*sepref-fr-rules*]: ‹(*uncurry length-aa64-nat*, *uncurry* (*RETURN* ∘∘ *length-ll*))
∈
$[\lambda(xs,\ i).\ i < length\ xs]_a\ (arrayO\text{-}assn\ (arl64\text{-}assn\ R))^k *_a nat\text{-}assn^k \rightarrow nat\text{-}assn$›
⟨*proof*⟩

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* $l \equiv \lambda(a,n).\ \exists_A l'.\ a \mapsto_a l' * \uparrow(\textit{nat-of-uint32}\ n \leq \textit{length}\ l' \wedge l = \textit{take}\ (\textit{nat-of-uint32}\ n)\ l' \wedge \textit{length}\ l'{>}0 \wedge \textit{nat-of-uint32}\ n \leq \textit{uint32-max} \wedge \textit{length}\ l' \leq \textit{uint32-max})$

lemma *is-array-list32-prec*[*safe-constraint-rules*]: *precise is-array-list32*
⟨*proof*⟩

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 []>
  ⟨proof⟩

lemma arl32-empty-sz-rule[sep-heap-rules]: < emp > arl32-empty-sz N <is-array-list32 []>
  ⟨proof⟩

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>
  ⟨proof⟩

lemma nat-of-uint32-shiftl: ‹nat-of-uint32 (xs >> a) = nat-of-uint32 xs >> a›
  ⟨proof⟩

lemma [simp]: ‹nat-of-uint32 uint32-max-uint32 = uint32-max›
  ⟨proof⟩

lemma ‹2 * (uint32-max div 2) = uint32-max − 1›
  ⟨proof⟩

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⟩


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)>
  ⟨proof⟩

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*=[]))>
⟨*proof*⟩

lemma *nat-of-uint32-ge-minus*:
‹*ai* ≥ *bi* ⟹
nat-of-uint32 (*ai* − *bi*) = *nat-of-uint32 ai* − *nat-of-uint32 bi*›
⟨*proof*⟩

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*)>
⟨*proof*⟩

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*)>
⟨*proof*⟩

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*])>
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *arl32-assn-comp'*: *hr-comp* (*arl32-assn id-assn*) (⟨*B*⟩*list-rel*) = *arl32-assn* (*pure B*)
⟨*proof*⟩

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*
⟨*proof*⟩
sepref-decl-impl (*no-register*) *arl32-empty*: *arl32-empty-hnr-aux* ⟨*proof*⟩

lemma *arl32-empty-sz-hnr-aux*: (*uncurry0* (*arl32-empty-sz N*),*uncurry0* (*RETURN op-list-empty*)) ∈

unit-assn^k $\rightarrow_a$ *is-array-list32*
⟨*proof*⟩

sepref-decl-impl (*no-register*) *arl32-empty-sz*: *arl32-empty-sz-hnr-aux* ⟨*proof*⟩

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*
⟨*proof*⟩
sepref-decl-impl *arl32-copy*: *arl32-copy-hnr-aux* ⟨*proof*⟩

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*
⟨*proof*⟩
sepref-decl-impl *arl32-append*: *arl32-append-hnr-aux*
⟨*proof*⟩

lemma *arl32-length-hnr-aux*: (*arl32-length*,*RETURN o op-list-length*) ∈ *is-array-list32*^{k} $\rightarrow_a$ *uint32-nat-assn*
⟨*proof*⟩
sepref-decl-impl *arl32-length*: *arl32-length-hnr-aux* ⟨*proof*⟩

lemma *arl32-is-empty-hnr-aux*: (*arl32-is-empty*,*RETURN o op-list-is-empty*) ∈ *is-array-list32*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩
sepref-decl-impl *arl32-is-empty*: *arl32-is-empty-hnr-aux* ⟨*proof*⟩

lemma *arl32-last-hnr-aux*: (*arl32-last*,*RETURN o op-list-last*) ∈ [*pre-list-last*]$_a$ *is-array-list32*^{k} → *id-assn*
⟨*proof*⟩
sepref-decl-impl *arl32-last*: *arl32-last-hnr-aux* ⟨*proof*⟩

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*
⟨*proof*⟩
sepref-decl-impl *arl32-get*: *arl32-get-hnr-aux* ⟨*proof*⟩

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*
⟨*proof*⟩
sepref-decl-impl *arl32-set*: *arl32-set-hnr-aux* ⟨*proof*⟩

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*)
⟨*proof*⟩
sepref-decl-impl (*ismop*) *arl32-swap*: *arl32-swap.refine* ⟨*proof*⟩
end

interpretation *arl32*: *list-custom-empty arl32-assn A arl32-empty op-arl32-empty*
⟨*proof*⟩

lemma [*def-pat-rules*]: *op-arl32-empty-sz*$*N* ≡ *UNPROTECT* (*op-arl32-empty-sz N*) ⟨*proof*⟩

interpretation *arl32-sz*: *list-custom-empty arl32-assn A arl32-empty-sz N PR-CONST* (*op-arl32-empty-sz N*)
⟨*proof*⟩

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*) ∈ $(arl32\text{-}assn\ R)^d \rightarrow_a arl\text{-}assn\ R$›
⟨*proof*⟩

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*)) ∈
$[\lambda(n, xs).\ n \leq length\ xs]_a\ uint32\text{-}nat\text{-}assn^k *_a (arl32\text{-}assn\ R)^d \rightarrow arl32\text{-}assn\ R$›
⟨*proof*⟩

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*) ∈
$[\lambda xs.\ xs \neq []]_a\ (arl32\text{-}assn\ R)^d \rightarrow arl32\text{-}assn\ R$›
⟨*proof*⟩

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* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ ′*a list* ⇒ *nat* ⇒ *nat* ⇒ *nat nres*› **where**
‹*choose-pivot - - - lo hi* = *SPEC*(λ*k*. *k* ≥ *lo* ∧ *k* ≤ *hi*)›

The element at index *p* partitions the subarray *lo..hi*. This means that every element

definition *isPartition-wrt* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ ′*b list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*isPartition-wrt R xs lo hi p* ≡ (∀ *i*. *i* ≥ *lo* ∧ *i* < *p* ⟶ *R* (*xs*!*i*) (*xs*!*p*)) ∧ (∀ *j*. *j* > *p* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*p*) (*xs*!*j*))›

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

definition *isPartition* :: ‹′*a* :: *order list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*isPartition xs lo hi p* ≡ *isPartition-wrt* (≤) *xs lo hi p*›

abbreviation *isPartition-map* :: ‹(′b ⇒ ′b ⇒ bool) ⇒ (′a ⇒ ′b) ⇒ ′a list ⇒ nat ⇒ nat ⇒ nat ⇒ bool›
where
‹*isPartition-map R h xs i j k* ≡ *isPartition-wrt* (λ*a b*. *R* (*h a*) (*h b*)) *xs i j k*›

lemma *isPartition-map-def′*:
‹*lo* ≤ *p* ⟹ *p* ≤ *hi* ⟹ *hi* < *length xs* ⟹ *isPartition-map R h xs lo hi p* = *isPartition-wrt R* (*map h xs*) *lo hi p*›
⟨*proof*⟩

Example: 6 is the pivot element (with index 4); *7*::′*a* is equal to the *length xs* − *1*.

lemma ‹*isPartition* [*0*,*5*,*3*,*4*,*6*,*9*,*8*,*10*::*nat*] *0 7 4*›
⟨*proof*⟩

definition *sublist* :: ‹′*a list* ⇒ *nat* ⇒ *nat* ⇒ ′*a list*› **where**
‹*sublist xs i j* ≡ *take* (*Suc j* − *i*) (*drop i xs*)›

lemma *take-Suc0*:
l≠[] ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
0 < *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
Suc n ≤ *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
⟨*proof*⟩

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

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

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

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

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

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

definition *sorted-sublist-wrt* :: ‹(′b ⇒ ′b ⇒ bool) ⇒ ′b list ⇒ nat ⇒ nat ⇒ bool› **where**
‹*sorted-sublist-wrt R xs lo hi* = *sorted-wrt R* (*sublist xs lo hi*)›

definition *sorted-sublist* :: ‹′*a* :: *linorder list* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*sorted-sublist xs lo hi* = *sorted-sublist-wrt* (≤) *xs lo hi*›

abbreviation *sorted-sublist-map* :: ‹(′b ⇒ ′b ⇒ bool) ⇒ (′a ⇒ ′b) ⇒ ′a list ⇒ nat ⇒ nat ⇒ bool›
where
‹*sorted-sublist-map R h xs lo hi* ≡ *sorted-sublist-wrt* (λ*a b*. *R* (*h a*) (*h b*)) *xs lo hi*›

lemma *sorted-sublist-map-def′*:

‹$lo < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi \equiv sorted\text{-}sublist\text{-}wrt\ R\ (map\ h\ xs)\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-sublist-wrt-refl*: ‹$i < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ i\ i$›
⟨*proof*⟩

lemma *sorted-sublist-refl*: ‹$i < length\ xs \Longrightarrow sorted\text{-}sublist\ xs\ i\ i$›
⟨*proof*⟩

lemma *sorted-sublist-map-refl*: ‹$i < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ i\ i$›
⟨*proof*⟩

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

lemma *take-set*: ‹$j \le length\ xs \Longrightarrow x \in set\ (take\ j\ xs) \equiv (\exists\ k.\ k < j \wedge xs!k = x)$›
⟨*proof*⟩

lemma *drop-set*: ‹$j \le length\ xs \Longrightarrow x \in set\ (drop\ j\ xs) \equiv (\exists k.\ j{\le}k{\wedge}k{<}length\ xs \wedge xs!k{=}x)$›
⟨*proof*⟩

lemma *sublist-el*: ‹$i \le j \Longrightarrow j < length\ xs \Longrightarrow x \in set\ (sublist\ xs\ i\ j) \equiv (\exists\ k.\ k < Suc\ j{-}i \wedge xs!(i{+}k){=}x)$›
⟨*proof*⟩

lemma *sublist-el′*: ‹$i \le j \Longrightarrow j < length\ xs \Longrightarrow x \in set\ (sublist\ xs\ i\ j) \equiv (\exists\ k.\ i{\le}k{\wedge}k{\le}j \wedge xs!k{=}x)$›
⟨*proof*⟩

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

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

lemma *sorted-sublist-wrt-le*: ‹$hi \le lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨*proof*⟩

Elements in a sorted sublists are actually sorted

lemma *sorted-sublist-wrt-nth-le*:
assumes ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$› **and** ‹$lo \le hi$› **and** ‹$hi < length\ xs$› **and**
‹$lo \le i$› **and** ‹$i < j$› **and** ‹$j \le hi$›
shows ‹$R\ (xs!i)\ (xs!j)$›
⟨*proof*⟩

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

lemma *sorted-sublist-wrt-nth-le′*:
assumes *ref*: ‹$\bigwedge\ x.\ R\ x\ x$›
and ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$› **and** ‹$lo \le hi$› **and** ‹$hi < length\ xs$›
and ‹$lo \le i$› **and** ‹$i \le j$› **and** ‹$j \le hi$›
shows ‹$R\ (xs!i)\ (xs!j)$›
⟨*proof*⟩

lemma *sorted-sublist-le*: ‹$hi \leq lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-sublist-map-le*: ‹$hi \leq lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi$›
⟨*proof*⟩

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

lemma *sorted-sublist-wrt-cons'*:
‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ (lo+1)\ hi \Longrightarrow lo \leq hi \Longrightarrow hi < length\ xs \Longrightarrow (\forall j.\ lo{<}j{\wedge}j{\leq}hi \longrightarrow R\ (xs!lo)\ (xs!j)) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-sublist-wrt-cons*:
assumes *trans*: ‹$(\bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z)$› **and**
‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ (lo+1)\ hi$› **and**
‹$lo \leq hi$› **and** ‹$hi < length\ xs$ › **and** ‹$R\ (xs!lo)\ (xs!(lo+1))$›
shows ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-sublist-map-cons*:
‹$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ (lo+1)\ hi \Longrightarrow lo \leq hi \Longrightarrow hi < length\ xs \Longrightarrow R\ (h\ (xs!lo))\ (h\ (xs!(lo+1)))$
$\Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi$›
⟨*proof*⟩

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

lemma *sorted-sublist-wrt-snoc'*:
‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (hi-1) \Longrightarrow lo \leq hi \Longrightarrow hi < length\ xs \Longrightarrow (\forall j.\ lo{\leq}j{\wedge}j{<}hi \longrightarrow R\ (xs!j)\ (xs!hi)) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-sublist-wrt-snoc*:
assumes *trans*: ‹$(\bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z)$› **and**
‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (hi-1)$› **and**
‹$lo \leq hi$› **and** ‹$hi < length\ xs$› **and** ‹$(R\ (xs!(hi-1))\ (xs!hi))$›
shows ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-sublist-map-snoc*:
‹$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ (hi-1) \Longrightarrow$
$lo \leq hi \Longrightarrow hi < length\ xs \Longrightarrow (R\ (h\ (xs!(hi-1)))\ (h\ (xs!hi))) \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sublist-split*: ‹$lo \leq hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow sublist\ xs\ lo\ p\ @\ sublist\ xs\ (p+1)\ hi = sublist\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sublist-split-part*: ‹$lo \leq hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow sublist\ xs\ lo\ (p-1)\ @\ xs!p\ \#\ sublist\ xs\ (p+1)\ hi = sublist\ xs\ lo\ hi$›
⟨*proof*⟩

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

lemma *isPartition-wrt-trans*:
‹$(\bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z) \Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$(\forall\ i\ j.\ lo \leq i \wedge i < p \wedge p < j \wedge j \leq hi \longrightarrow R\ (xs!i)\ (xs!j))$›
⟨*proof*⟩

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

lemma *merge-sorted-wrt-partitions-between'*:
‹$lo \leq hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (p-1) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ (p+1)\ hi \Longrightarrow$
$(\forall\ i\ j.\ lo \leq i \wedge i < p \wedge p < j \wedge j \leq hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨*proof*⟩

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

The main theorem to merge sorted lists

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

theorem *merge-sorted-map-partitions*:
‹$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
$isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ (p - Suc\ 0) \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ (Suc\ p)\ hi \Longrightarrow$

lo ≤ *hi* ⟹ *lo* ≤ *p* ⟹ *p* ≤ *hi* ⟹ *hi* < *length xs* ⟹
sorted-sublist-map R h xs lo hi›
⟨*proof*⟩

lemma *partition-wrt-extend*:
‹*isPartition-wrt R xs lo′ hi′ p* ⟹
hi < *length xs* ⟹
lo ≤ *lo′* ⟹ *lo′* ≤ *hi* ⟹ *hi′* ≤ *hi* ⟹
lo′ ≤ *p* ⟹ *p* ≤ *hi′* ⟹
(⋀ *i*. *lo*≤*i* ⟹ *i* <*lo′* ⟹ *R* (*xs*!*i*) (*xs*!*p*)) ⟹
(⋀ *j*. *hi′*<*j* ⟹ *j*≤*hi* ⟹ *R* (*xs*!*p*) (*xs*!*j*)) ⟹
isPartition-wrt R xs lo hi p›
⟨*proof*⟩

lemma *partition-map-extend*:
‹*isPartition-map R h xs lo′ hi′ p* ⟹
hi < *length xs* ⟹
lo ≤ *lo′* ⟹ *lo′* ≤ *hi* ⟹ *hi′* ≤ *hi* ⟹
lo′ ≤ *p* ⟹ *p* ≤ *hi′* ⟹
(⋀ *i*. *lo*≤*i* ⟹ *i* <*lo′* ⟹ *R* (*h* (*xs*!*i*)) (*h* (*xs*!*p*))) ⟹
(⋀ *j*. *hi′*<*j* ⟹ *j*≤*hi* ⟹ *R* (*h* (*xs*!*p*)) (*h* (*xs*!*j*))) ⟹
isPartition-map R h xs lo hi p›
⟨*proof*⟩

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

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

lemma *drop-ext′*:
‹(∀ *i*. *i*≥*k* ∧ *i*<*length xs* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
0<*k* ⟹ *xs*≠[] ⟹ — These corner cases will be dealt with in the next lemma
length xs′=*length xs* ⟹
drop k xs′ = *drop k xs*›
⟨*proof*⟩

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

lemma *sublist-ext′*:
‹(∀ *i*. *lo*≤*i*∧*i*≤*hi* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
length xs′ = *length xs* ⟹

lo ≤ *hi* ⟹ *Suc hi* < *length xs* ⟹
sublist xs′ lo hi = *sublist xs lo hi*›
⟨*proof*⟩

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

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

lemma *sublist-ext*:
‹(∀ *i*. *lo*≤*i*∧*i*≤*hi* ⟶ *xs′*!*i*=*xs*!*i*) ⟹
length xs′ = *length xs* ⟹
lo ≤ *hi* ⟹ *hi* < *length xs* ⟹
sublist xs′ lo hi = *sublist xs lo hi*›
⟨*proof*⟩

lemma *sorted-wrt-lower-sublist-still-sorted*:
assumes ‹*sorted-sublist-wrt R xs lo* (*lo′* − *Suc 0*)› **and**
‹*lo* ≤ *lo′*› **and** ‹*lo′* < *length xs*› **and**
‹(∀ *i*. *lo*≤*i*∧*i*<*lo′* ⟶ *xs′*!*i*=*xs*!*i*)› **and** ‹*length xs′* = *length xs*›
shows ‹*sorted-sublist-wrt R xs′ lo* (*lo′* − *Suc 0*)›
⟨*proof*⟩

lemma *sorted-map-lower-sublist-still-sorted*:
assumes ‹*sorted-sublist-map R h xs lo* (*lo′* − *Suc 0*)› **and**
‹*lo* ≤ *lo′*› **and** ‹*lo′* < *length xs*› **and**
‹(∀ *i*. *lo*≤*i*∧*i*<*lo′* ⟶ *xs′*!*i*=*xs*!*i*)› **and** ‹*length xs′* = *length xs*›
shows ‹*sorted-sublist-map R h xs′ lo* (*lo′* − *Suc 0*)›
⟨*proof*⟩

lemma *sorted-wrt-upper-sublist-still-sorted*:
assumes ‹*sorted-sublist-wrt R xs* (*hi′*+*1*) *hi*› **and**
‹*lo* ≤ *lo′*› **and** ‹*hi* < *length xs*› **and**
‹∀ *j*. *hi′*<*j*∧*j*≤*hi* ⟶ *xs′*!*j*=*xs*!*j*› **and** ‹*length xs′* = *length xs*›
shows ‹*sorted-sublist-wrt R xs′* (*hi′*+*1*) *hi*›
⟨*proof*⟩

lemma *sorted-map-upper-sublist-still-sorted*:
assumes ‹*sorted-sublist-map R h xs* (*hi′*+*1*) *hi*› **and**
‹*lo* ≤ *lo′*› **and** ‹*hi* < *length xs*› **and**
‹∀ *j*. *hi′*<*j*∧*j*≤*hi* ⟶ *xs′*!*j*=*xs*!*j*› **and** ‹*length xs′* = *length xs*›
shows ‹*sorted-sublist-map R h xs′* (*hi′*+*1*) *hi*›
⟨*proof*⟩

The specification of the partition function

definition *partition-spec* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ ′*a list* ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ *nat* ⇒ *bool*› **where**
‹*partition-spec R h xs lo hi xs′ p* ≡
mset xs′ = *mset xs* ∧ — The list is a permutation
isPartition-map R h xs′ lo hi p ∧ — We have a valid partition on the resulting list
lo ≤ *p* ∧ *p* ≤ *hi* ∧ — The partition index is in bounds
(∀ *i*. *i*<*lo* ⟶ *xs′*!*i*=*xs*!*i*) ∧ (∀ *i*. *hi*<*i*∧*i*<*length xs′* ⟶ *xs′*!*i*=*xs*!*i*)› — Everything else is unchanged.

lemma *mathias*:
assumes
Perm: ‹*mset xs′* = *mset xs*›
and *I*: ‹*lo*≤*i*› ‹*i*≤*hi*› ‹*xs′*!*i*=*x*›
and *Bounds*: ‹*hi* < *length xs*›
and *Fix*: ‹⋀ *i*. *i*<*lo* ⟹ *xs′*!*i* = *xs*!*i*› ‹⋀ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs′*!*j* = *xs*!*j*›
shows ‹∃ *j*. *lo*≤*j*∧*j*≤*hi* ∧ *xs*!*j* = *x*›
⟨*proof*⟩

If we fix the left and right rest of two permutated lists, then the sublists are also permutations.

But we only need that the sets are equal.

lemma *mset-sublist-incl*:
assumes *Perm*: ‹*mset xs′* = *mset xs*›
and *Fix*: ‹⋀ *i*. *i*<*lo* ⟹ *xs′*!*i* = *xs*!*i*› ‹⋀ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs′*!*j* = *xs*!*j*›
and *bounds*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›
shows ‹*set* (*sublist xs′ lo hi*) ⊆ *set* (*sublist xs lo hi*)›
⟨*proof*⟩

lemma *mset-sublist-eq*:
assumes ‹*mset xs′* = *mset xs*›
and ‹⋀ *i*. *i*<*lo* ⟹ *xs′*!*i* = *xs*!*i*›
and ‹⋀ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs′*!*j* = *xs*!*j*›
and *bounds*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›
shows ‹*set* (*sublist xs′ lo hi*) = *set* (*sublist xs lo hi*)›
⟨*proof*⟩

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

definition *quicksort* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ *nat* × *nat* × ′*a list* ⇒ ′*a list nres*› **where**
‹*quicksort R h* = (λ(*lo*,*hi*,*xs0*). *do* {
RECT (λ*f* (*lo*,*hi*,*xs*). *do* {
ASSERT(*lo* ≤ *hi* ∧ *hi* < *length xs* ∧ *mset xs* = *mset xs0*); — Premise for a partition function
(*xs*, *p*) ← *SPEC*(*uncurry* (*partition-spec R h xs lo hi*)); — Abstract partition function
ASSERT(*mset xs* = *mset xs0*);
xs ← (*if p*−*1*≤*lo then RETURN xs else f* (*lo*, *p*−*1*, *xs*));
ASSERT(*mset xs* = *mset xs0*);
if hi≤*p*+*1 then RETURN xs else f* (*p*+*1*, *hi*, *xs*)
}) (*lo*,*hi*,*xs0*)
})›

As premise for quicksor, we only need that the indices are ok.

definition *quicksort-pre* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ ′*a list* ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ *bool*›
where
‹*quicksort-pre R h xs0 lo hi xs* ≡ *lo* ≤ *hi* ∧ *hi* < *length xs* ∧ *mset xs* = *mset xs0*›

definition *quicksort-post* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ ′*a list* ⇒ *bool*›
where
‹*quicksort-post R h lo hi xs xs′* ≡
mset xs′ = *mset xs* ∧
sorted-sublist-map R h xs′ lo hi ∧
(∀ *i*. *i*<*lo* ⟶ *xs′*!*i* = *xs*!*i*) ∧
(∀ *j*. *hi*<*j*∧*j*<*length xs* ⟶ *xs′*!*j* = *xs*!*j*)›

Convert Pure to HOL

lemma *quicksort-postI*:
‹⟦*mset xs′* = *mset xs*; *sorted-sublist-map R h xs′ lo hi*; (⋀ *i*. ⟦*i*<*lo*⟧ ⟹ *xs′*!*i* = *xs*!*i*); (⋀ *j*. ⟦*hi*<*j*; *j*<*length xs*⟧ ⟹ *xs′*!*j* = *xs*!*j*)⟧ ⟹ *quicksort-post R h lo hi xs xs′*›
⟨*proof*⟩

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

lemma *quicksort-correct-case1*:
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*−*1* ≤ *lo*› ‹*hi* ≤ *p*+*1*›
shows ‹*quicksort-post R h lo hi xs xs′*›
⟨*proof*⟩

In the second case, we have to show that the precondition still holds for (p+1, hi, x') after the partition.

lemma *quicksort-correct-case2*:
assumes
pre: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
and *ifs*: ‹¬ *hi* ≤ *p* + *1*›
shows ‹*quicksort-pre R h xs0* (*Suc p*) *hi xs′*›
⟨*proof*⟩

lemma *quicksort-post-set*:
assumes ‹*quicksort-post R h lo hi xs xs′*›
and *bounds*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›
shows ‹*set* (*sublist xs′ lo hi*) = *set* (*sublist xs lo hi*)›
⟨*proof*⟩

In the third case, we have run quicksort recursively on (p+1, hi, xs') after the partition, with hi<=p+1 and p-1<=lo.

lemma *quicksort-correct-case3*:
assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
and *ifs*: ‹*p* − *Suc 0* ≤ *lo*› ‹¬ *hi* ≤ *Suc p*›
and *IH1′*: ‹*quicksort-post R h* (*Suc p*) *hi xs′ xs′′*›
shows ‹*quicksort-post R h lo hi xs xs′′*›
⟨*proof*⟩

In the 4th case, we have to show that the premise holds for $(lo, p - (1::'b), xs')$, in case $\neg\ p - (1::'a) \leq lo$

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*: ‹¬ $p - Suc\ 0 \le lo$ ›
shows ‹*quicksort-pre R h xs0 lo* $(p-Suc\ 0)$ *xs'*›
⟨*proof*⟩

In the 5th case, we have run quicksort recursively on (lo, p-1, xs').

lemma *quicksort-correct-case5*:
assumes *trans*: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs' p*›
and *ifs*: ‹¬ $p - Suc\ 0 \le lo$› ‹$hi \le Suc\ p$›
and *IH1'*: ‹*quicksort-post R h lo* $(p - Suc\ 0)$ *xs' xs''*›
shows ‹*quicksort-post R h lo hi xs xs''*›
⟨*proof*⟩

In the 6th case, we have run quicksort recursively on (lo, p-1, xs'). We show the precondition on the second call on (p+1, hi, xs")

lemma *quicksort-correct-case6*:
assumes
pre: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs' p*›
and *ifs*: ‹¬ $p - Suc\ 0 \le lo$› ‹¬ $hi \le Suc\ p$›
and *IH1*: ‹*quicksort-post R h lo* $(p - Suc\ 0)$ *xs' xs''*›
shows ‹*quicksort-pre R h xs0* $(Suc\ p)$ *hi xs''*›
⟨*proof*⟩

In the 7th (and last) case, we have run quicksort recursively on (lo, p-1, xs'). We show the postcondition on the second call on (p+1, hi, xs")

lemma *quicksort-correct-case7*:
assumes *trans*: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
and *part*: ‹*partition-spec R h xs lo hi xs' p*›
and *ifs*: ‹¬ $p - Suc\ 0 \le lo$› ‹¬ $hi \le Suc\ p$›
and *IH1'*: ‹*quicksort-post R h lo* $(p - Suc\ 0)$ *xs' xs''*›
and *IH2'*: ‹*quicksort-post R h* $(Suc\ p)$ *hi xs'' xs'''*›
shows ‹*quicksort-post R h lo hi xs xs'''*›
⟨*proof*⟩

We can now show the correctness of the abstract quicksort procedure, using the refinement framework and the above case lemmas.

lemma *quicksort-correct*:
assumes *trans*: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *Pre*: ‹$lo0 \le hi0$› ‹$hi0 < length\ xs0$›
shows ‹*quicksort R h* $(lo0,hi0,xs0) \le \Downarrow Id\ (SPEC(\lambda xs.\ quicksort\text{-}post\ R\ h\ lo0\ hi0\ xs0\ xs))$›
⟨*proof*⟩

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 ≡
  case p of (i,j,xs) ⇒
  j < length xs ∧ j ≤ hi ∧ i < length xs ∧ lo ≤ i ∧ i ≤ j ∧ mset xs = mset xs0 ∧
  (∀ k. k ≥ lo ∧ k < i ⟶ R (h (xs!k)) (h (xs!hi))) ∧ — All elements from lo to i − (1::'c) are smaller than the pivot
  (∀ k. k ≥ i ∧ k < j ⟶ R (h (xs!hi)) (h (xs!k))) ∧ — All elements from i to j − (1::'c) are greater than the pivot
  (∀ k. k < lo ⟶ xs!k = xs0!k) ∧ — Everything below lo is unchanged
  (∀ k. k ≥ j ∧ k < length xs ⟶ xs!k = xs0!k) — All elements from j are unchanged (including everyting above hi)
›

The main part of the partition function. The pivot is assumed to be the last element. This is exactly the "Lomuto partition scheme" partition function from Wikipedia.

definition *partition-main* :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ nat ⇒ nat ⇒ 'a list ⇒ ('a list × nat) nres› **where**
```
‹partition-main R h lo hi xs0 = do {
  ASSERT(hi < length xs0);
  pivot ← RETURN (h (xs0 ! hi));
  (i,j,xs) ← WHILE_T^{partition-main-inv R h lo hi xs0} — We loop from j = lo to j = hi − (1::'c).
    (λ(i,j,xs). j < hi)
    (λ(i,j,xs). do {
      ASSERT(i < length xs ∧ j < length xs);
      if R (h (xs!j)) pivot
      then RETURN (i+1, j+1, swap xs i j)
      else RETURN (i,   j+1, xs)
    })
    (lo, lo, xs0); — i and j are both initialized to lo
  ASSERT(i < length xs ∧ j = hi ∧ lo ≤ i ∧ hi < length xs ∧ mset xs = mset xs0);
  RETURN (swap xs i hi, i)
}›
```

lemma *partition-main-correct*:
  assumes *bounds*: ‹hi < length xs› ‹lo ≤ hi› **and**
  trans: ‹⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)› **and** *lin*: ‹⋀x y. R (h x) (h y) ∨ R (h y) (h x)›
  shows ‹partition-main R h lo hi xs ≤ SPEC(λ(xs', p). mset xs = mset xs' ∧
  lo ≤ p ∧ p ≤ hi ∧ isPartition-map R h xs' lo hi p ∧ (∀ i. i<lo ⟶ xs'!i=xs!i) ∧ (∀ i. hi<i∧i<length xs' ⟶ xs'!i=xs!i))›
⟨proof⟩

definition *partition-between* :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ nat ⇒ nat ⇒ 'a list ⇒ ('a list × nat) nres› **where**
```
‹partition-between R h lo hi xs0 = do {
  ASSERT(hi < length xs0 ∧ lo ≤ hi);
  k ← choose-pivot R h xs0 lo hi; — choice of pivot
  ASSERT(k < length xs0);
  xs ← RETURN (swap xs0 k hi); — move the pivot to the last position, before we start the actual loop
  ASSERT(length xs = length xs0);
  partition-main R h lo hi xs
}›
```

lemma *partition-between-correct*:
assumes ‹$hi < length\ xs$› **and** ‹$lo \leq hi$› **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*⟩

We use the median of the first, the middle, and the last element.

definition *choose-pivot3* **where**
‹$choose\text{-}pivot3\ R\ h\ xs\ lo\ (hi::nat) = do\ \{$
$ASSERT(lo < length\ xs);$
$ASSERT(hi < length\ xs);$
$let\ k' = (hi - lo)\ div\ 2;$
$let\ k = lo + k';$
$ASSERT(k < length\ xs);$
$let\ start = h\ (xs\ !\ lo);$
$let\ mid = h\ (xs\ !\ k);$
$let\ end = h\ (xs\ !\ hi);$
$if\ (R\ start\ mid \wedge R\ mid\ end) \vee (R\ end\ mid \wedge R\ mid\ start)\ then\ RETURN\ k$
$else\ if\ (R\ start\ end \wedge R\ end\ mid) \vee (R\ mid\ end \wedge R\ end\ start)\ then\ RETURN\ hi$
$else\ RETURN\ lo$
$\}$›

— 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 ‹$choose\text{-}pivot3\ R\ h\ xs\ lo\ hi \leq \Downarrow Id\ (choose\text{-}pivot\ R\ h\ xs\ lo\ hi)$›
⟨*proof*⟩

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
‹$partition\text{-}between\text{-}ref\ R\ h\ lo\ hi\ xs0 = do\ \{$
$ASSERT(hi < length\ xs0 \wedge hi < length\ xs0 \wedge lo \leq hi);$
$k \leftarrow choose\text{-}pivot3\ R\ h\ xs0\ lo\ hi;$ — 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
$ASSERT(length\ xs = length\ xs0);$
$partition\text{-}main\ R\ h\ lo\ hi\ xs$
$\}$›

lemma *partition-main-ref'*:
‹$partition\text{-}main\ R\ h\ lo\ hi\ xs$
$\leq \Downarrow ((\lambda\ a\ b\ c\ d.\ Id)\ a\ b\ c\ d)\ (partition\text{-}main\ R\ h\ lo\ hi\ xs)$›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

Example instantiation for pivot

definition *choose-pivot3-impl* **where**
‹*choose-pivot3-impl* = *choose-pivot3* (≤) *id*›

lemma *partition-between-ref-correct*:
assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
and *bounds*: ‹*hi* < *length xs*› ‹*lo* ≤ *hi*›
shows ‹*partition-between-ref R h lo hi xs* ≤ *SPEC* (*uncurry* (*partition-spec R h xs lo hi*))›
⟨*proof*⟩

term *quicksort*

Refined quicksort algorithm: We use the refined partition function.

definition *quicksort-ref* :: ‹- ⇒ - ⇒ *nat* × *nat* × *'a list* ⇒ *'a list nres*› **where**
‹*quicksort-ref R h* = (λ(*lo*,*hi*,*xs0*).
do {
RECT (λ*f* (*lo*,*hi*,*xs*). *do* {
ASSERT(*lo* ≤ *hi* ∧ *hi* < *length xs0* ∧ *mset xs* = *mset xs0*);
(*xs*, *p*) ← *partition-between-ref R h lo hi xs*; — This is the refined partition function. Note that we need the premises (trans,lin,bounds) here.
ASSERT(*mset xs* = *mset xs0* ∧ *p* ≥ *lo* ∧ *p* < *length xs0*);
xs ← (*if p*−*1*≤*lo then RETURN xs else f* (*lo*, *p*−*1*, *xs*));
ASSERT(*mset xs* = *mset xs0*);
if hi≤*p*+*1 then RETURN xs else f* (*p*+*1*, *hi*, *xs*)
}) (*lo*,*hi*,*xs0*)
})›

lemma *quicksort-ref-quicksort*:
assumes *bounds*: ‹*hi* < *length xs*› ‹*lo* ≤ *hi*› **and**
trans: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*quicksort-ref R h x0* ≤ ⇓ *Id* (*quicksort R h x0*)›
⟨*proof*⟩
definition *full-quicksort* **where**
‹*full-quicksort R h xs* ≡ *if xs* = [] *then RETURN xs else quicksort R h* (*0*, *length xs* − *1*, *xs*)›

definition *full-quicksort-ref* **where**
‹*full-quicksort-ref R h xs* ≡
if List.null xs then RETURN xs
else quicksort-ref R h (*0*, *length xs* − *1*, *xs*)›

definition *full-quicksort-impl* :: ‹*nat list* ⇒ *nat list nres*› **where**
‹*full-quicksort-impl xs* = *full-quicksort-ref* (≤) *id xs*›

lemma *full-quicksort-ref-full-quicksort*:

assumes *trans*: ‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹(*full-quicksort-ref R h*, *full-quicksort R h*) ∈ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨ ⟨*Id*⟩*list-rel*⟩*nres-rel*›
⟨*proof*⟩

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

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

lemma *sorted-sublist-map-entire*:
assumes ‹*sorted-sublist-map R h xs 0* (*length xs* − *1*)›
shows ‹*sorted-wrt* (λ *x y*. *R* (*h x*) (*h y*)) *xs*›
⟨*proof*⟩

Final correctness lemma

lemma *full-quicksort-correct-sorted*:
assumes
trans: ‹⋀*x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*full-quicksort R h xs* ≤ ⇓ *Id* (*SPEC*(λ*xs′*. *mset xs′* = *mset xs* ∧ *sorted-wrt* (λ *x y*. *R* (*h x*) (*h y*)) *xs′*))›
⟨*proof*⟩

lemma *full-quicksort-correct*:
assumes
trans: ‹⋀*x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and**
lin: ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*full-quicksort R h xs* ≤ ⇓ *Id* (*SPEC*(λ*xs′*. *mset xs′* = *mset xs*))›
⟨*proof*⟩

end
theory *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*›
⟨*proof*⟩

sepref-register *choose-pivot3*

Example instantiation code for pivot

sepref-definition *choose-pivot3-impl-code*
is ‹*uncurry2* (*choose-pivot3-impl*)›
:: ‹(*arl-assn nat-assn*)^k $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k}$\rightarrow_a$ *nat-assn*›
⟨*proof*⟩

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-assn nat-assn ∗*a nat-assn*›
⟨*proof*⟩

declare *partition-main-code.refine*[*sepref-fr-rules*]

Example instantiation for partition

definition *partition-between-impl* **where**
‹*partition-between-impl* = *partition-between-ref* (≤) *id*›

sepref-register *partition-between-ref*

Example instantiation code for partition

sepref-definition *partition-between-code*
is ‹*uncurry2* (*partition-between-impl*)›
:: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
arl-assn nat-assn ∗*a nat-assn*›
⟨*proof*⟩

declare *partition-between-code.refine*[*sepref-fr-rules*]

— Example implementation
definition *quicksort-impl* **where**
‹*quicksort-impl a b c* ≡ *quicksort-ref* (≤) *id* (*a*,*b*,*c*)›

sepref-register *quicksort-impl*

— Example implementation code
sepref-definition
quicksort-code
is ‹*uncurry2 quicksort-impl*›
:: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a (arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
arl-assn nat-assn›
⟨*proof*⟩

declare *quicksort-code.refine*[*sepref-fr-rules*]

Executable code for the example instance

sepref-definition *full-quicksort-code*
is ‹*full-quicksort-impl*›
:: ‹$(arl\text{-}assn\ nat\text{-}assn)^d \rightarrow_a$
arl-assn nat-assn›
⟨*proof*⟩

Export the code

export-code ‹*nat-of-integer*› ‹*integer-of-nat*› ‹*partition-between-code*› ‹*full-quicksort-code*› **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** ⟨∀ *L* ∈# *unwatched D*. −*L* ∈ *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′* ∈ *lits-of-l M*⟩ **and** ⟨∀ *L* ∈# *unwatched D*. −*L* ∈ *lits-of-l M*⟩ |
delete-from-working:
⟨*cdcl-twl-cp* (*M*, *N*, *U*, *None*, *NE*, *UE*, *add-mset* (*L*, *D*) *WS*, *Q*) (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*)⟩
if ⟨*L′* ∈# *clause D*⟩ **and** ⟨*L′* ∈ *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* ∈ *lits-of-l M*⟩ **and** ⟨*L′* ∉ *lits-of-l M*⟩ **and**
⟨*K* ∈# *unwatched D*⟩ **and** ⟨*undefined-lit M K* ∨ *K* ∈ *lits-of-l M*⟩ **and**
⟨*update-clauses* (*N*, *U*) *D L K* (*N′*, *U′*)⟩
— The condition − *L* ∈ *lits-of-l M* is already implied by *valid* invariant.

inductive-cases *cdcl-twl-cpE*: ⟨*cdcl-twl-cp S T*⟩

We do not care about the *literals-to-update* literals.

inductive *cdcl-twl-o* :: ⟨′*v twl-st* ⇒ ′*v twl-st* ⇒ *bool*⟩ **where**
decide:
⟨*cdcl-twl-o* (*M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#}) (*Decided L* # *M*, *N*, *U*, *None*, *NE*, *UE*, {#}, {#−*L*#})⟩
if ⟨*undefined-lit M L*⟩ **and** ⟨*atm-of L* ∈ *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* ∉# *D*⟩ **and** ⟨*D* ≠ {#}⟩
| *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* ∈# *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* ∈# *D*⟩ **and**
⟨(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)⟩ **and**
⟨*get-level M L* = *count-decided M*⟩ **and**
⟨*get-level M L* = *get-maximum-level M D′*⟩ **and**
⟨*get-maximum-level M* (*D′* − {#*L*#}) ≡ *i*⟩ **and**
⟨*get-level M K* = *i* + *1*⟩
⟨*D′* = {#*L*#}⟩ **and**
⟨*D′* ⊆# *D*⟩ **and**
⟨*clause* ‘# (*N* + *U*) + *NE* + *UE* ⊨*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* ∈# *D*⟩ **and**
⟨(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)⟩ **and**

⟨*get-level M L = count-decided M*⟩ **and**
⟨*get-level M L = get-maximum-level M D'*⟩ **and**
⟨*get-maximum-level M (D' − {#L#}) ≡ i*⟩ **and**
⟨*get-level M K = i + 1*⟩
⟨*D' ≠ {#L#}*⟩ **and**
⟨*D' ⊆# D*⟩ **and**
⟨*clause '# (N + U) + NE + UE ⊨pm D'*⟩ **and**
⟨*L ∈# D'*⟩
⟨*L' ∈# D'*⟩ **and** — *L'* is the new watched literal
⟨*get-level M L' = i*⟩

inductive-cases *cdcl-twl-oE*: ⟨*cdcl-twl-o S T*⟩

inductive *cdcl-twl-stgy* :: ⟨*'v twl-st ⇒ 'v twl-st ⇒ bool*⟩ **for** *S* :: ⟨*'v twl-st*⟩ **where**
cp: ⟨*cdcl-twl-cp S S' ⟹ cdcl-twl-stgy S S'*⟩ |
other': ⟨*cdcl-twl-o S S' ⟹ cdcl-twl-stgy S S'*⟩

inductive-cases *cdcl-twl-stgyE*: ⟨*cdcl-twl-stgy S T*⟩

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-clause ⇒ bool*⟩ **where**
⟨*struct-wf-twl-cls (TWL-Clause W UW) ⟷*
size W = 2 ∧ distinct-mset (W + UW)⟩

fun $state_W$*-of* :: ⟨*'v twl-st ⇒ 'v* $cdcl_W$*-restart-mset*⟩ **where**
⟨$state_W$*-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 (*$state_W$*-of S') = get-trail S'*⟩
⟨*proof*⟩

lemma [*twl-st*]:
⟨*get-trail S' ≠ [] ⟹* $cdcl_W$*-restart-mset.hd-trail (*$state_W$*-of S') = hd (get-trail S')*⟩
⟨*proof*⟩

lemma [*twl-st*]: ⟨*conflicting (*$state_W$*-of S') = get-conflict S'*⟩
⟨*proof*⟩

The invariant on the clauses is the following:

- the structure is correct (the watched part is of length exactly two).
- if we do not have to update the clause, then the invariant holds.

definition *twl-is-an-exception* :: ⟨*'a multiset twl-clause ⇒ 'a multiset ⇒*
('b × 'a multiset twl-clause) multiset ⇒ bool⟩
where

‹twl-is-an-exception C Q WS ⟷
  (∃L. L ∈# Q ∧ L ∈# watched C) ∨ (∃L. (L, C) ∈# WS)›

definition *is-blit* :: ‹('a, 'b) ann-lits ⇒ 'a clause ⇒ 'a literal ⇒ bool›**where**
  [*simp*]: ‹is-blit M D L ⟷ (L ∈# D ∧ L ∈ lits-of-l M)›

definition *has-blit* :: ‹('a, 'b) ann-lits ⇒ 'a clause ⇒ 'a literal ⇒ bool›**where**
  ‹has-blit M D L' ⟷ (∃L. is-blit M D L ∧ get-level M L ≤ get-level M L')›

This invariant state that watched literals are set at the end and are not swapped with an unwatched literal later.

fun *twl-lazy-update* :: ‹('a, 'b) ann-lits ⇒ 'a twl-cls ⇒ bool› **where**
‹twl-lazy-update M (TWL-Clause W UW) ⟷
  (∀L. L ∈# W ⟶ −L ∈ lits-of-l M ⟶ ¬has-blit M (W+UW) L ⟶
    (∀K ∈# UW. get-level M L ≥ get-level M K ∧ −K ∈ lits-of-l M))›

If one watched literals has been assigned to false (− L ∈ *lits-of-l* M) and the clause has not yet been updated (L' ∉ *lits-of-l* M: it should be removed either by updating L, propagating L', or marking the conflict), then the literals L is of maximal level.

fun *watched-literals-false-of-max-level* :: ‹('a, 'b) ann-lits ⇒ 'a twl-cls ⇒ bool› **where**
‹watched-literals-false-of-max-level M (TWL-Clause W UW) ⟷
  (∀L. L ∈# W ⟶ −L ∈ lits-of-l M ⟶ ¬has-blit M (W+UW) L ⟶
    get-level M L = count-decided M)›

This invariants talks about the enqueued literals:

- the working stack contains a single literal;
- the working stack and the *literals-to-update* literals are false with respect to the trail and there are no duplicates;
- and the latter condition holds even when WS = {#}.

fun *no-duplicate-queued* :: ‹'v twl-st ⇒ bool› **where**
‹no-duplicate-queued (M, N, U, D, NE, UE, WS, Q) ⟷
  (∀C C'. C ∈# WS ⟶ C' ∈# WS ⟶ fst C = fst C') ∧
  (∀C. C ∈# WS ⟶ add-mset (fst C) Q ⊆# uminus '# lit-of '# mset M) ∧
  Q ⊆# uminus '# lit-of '# mset M›

lemma *no-duplicate-queued-alt-def*:
  ‹no-duplicate-queued S =
   ((∀C C'. C ∈# clauses-to-update S ⟶ C' ∈# clauses-to-update S ⟶ fst C = fst C') ∧
    (∀C. C ∈# clauses-to-update S ⟶
      add-mset (fst C) (literals-to-update S) ⊆# uminus '# lit-of '# mset (get-trail S)) ∧
    literals-to-update S ⊆# uminus '# lit-of '# mset (get-trail S))›
  ⟨proof⟩

fun *distinct-queued* :: ‹'v twl-st ⇒ bool› **where**
‹distinct-queued (M, N, U, D, NE, UE, WS, Q) ⟷
  distinct-mset Q ∧
  (∀L C. count WS (L, C) ≤ count (N + U) C)›

These are the conditions to indicate that the 2-WL invariant does not hold and is not *literals-to-update*.

fun *clauses-to-update-prop* **where**

⟨*clauses-to-update-prop Q M (L, C)* ⟷
(*L* ∈# *watched C* ∧ −*L* ∈ *lits-of-l M* ∧ *L* ∉# *Q* ∧ ¬*has-blit M* (*clause C*) *L*)⟩
declare *clauses-to-update-prop.simps*[*simp del*]

This invariants talks about the enqueued literals:

- all clauses that should be updated are in *WS* and are repeated often enough in it.
- if *WS* = {#}, then there are no clauses to updated that is not enqueued;
- all clauses to updated are either in *WS* or *Q*.

  The first two conditions are written that way to please Isabelle.

fun *clauses-to-update-inv* :: ⟨'*v twl-st* ⇒ *bool*⟩ **where**
⟨*clauses-to-update-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*) ⟷
(∀ *L C*. ((*L*, *C*) ∈# *WS* ⟶ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*)) ∧
(∀ *L*. *WS* = {#} ⟶ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} = {#}) ∧
(∀ *L C*. *C* ∈# *N* + *U* ⟶ *L* ∈# *watched C* ⟶ −*L* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*clause C*) *L* ⟶
(*L*, *C*) ∉# *WS* ⟶ *L* ∈# *Q*)⟩
| ⟨*clauses-to-update-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⟷ *True*⟩

This is the invariant of the 2WL structure: if one watched literal is false, then all unwatched are false.

fun *twl-exception-inv* :: ⟨'*v twl-st* ⇒ '*v twl-cls* ⇒ *bool*⟩ **where**
⟨*twl-exception-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*) *C* ⟷
(∀ *L*. *L* ∈# *watched C* ⟶ −*L* ∈ *lits-of-l M* ⟶ ¬*has-blit M* (*clause C*) *L* ⟶
L ∉# *Q* ⟶ (*L*, *C*) ∉# *WS* ⟶
(∀ *K* ∈# *unwatched C*. −*K* ∈ *lits-of-l M*))⟩
| ⟨*twl-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) *C* ⟷ *True*⟩

declare *twl-exception-inv.simps*[*simp del*]

fun *twl-st-exception-inv* :: ⟨'*v twl-st* ⇒ *bool*⟩ **where**
⟨*twl-st-exception-inv* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⟷
(∀ *C* ∈# *N* + *U*. *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* ⇒ *bool*⟩ **where**
⟨*propa-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*) ⟷
(∀ *L C*. *C* ∈# *N*+*U* ⟶ *L* ∈# *clause C* ⟶ *M* ⊨*as CNot* (*remove1-mset L* (*clause C*)) ⟶
undefined-lit M L ⟶
(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))⟩
| ⟨*propa-cands-enqueued* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⟷ *True*⟩

fun *confl-cands-enqueued* :: ⟨'*v twl-st* ⇒ *bool*⟩ **where**
⟨*confl-cands-enqueued* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*) ⟷
(∀ *C* ∈# *N* + *U*. *M* ⊨*as CNot* (*clause C*) ⟶
(∃ *L'*. *L'* ∈# *watched C* ∧ *L'* ∈# *Q*) ∨ (∃ *L*. (*L*, *C*) ∈# *WS*))⟩
| ⟨*confl-cands-enqueued* (*M*, *N*, *U*, *Some* -, *NE*, *UE*, *WS*, *Q*) ⟷
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)⟩
⟨*proof*⟩

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*))›
⟨*proof*⟩

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

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

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

lemma *clauses-to-update-inv-cases*[*case-names WS-nempty WS-empty Q*]:
assumes
‹⋀*L C*. (*L*, *C*) ∈# *WS* ⟹ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} ⊆# *WS*› **and**
‹⋀*L*. *WS* = {#} ⟹ {#(*L*, *C*)| *C* ∈# *N* + *U*. *clauses-to-update-prop Q M* (*L*, *C*)#} = {#}› **and**
‹⋀*L C*. *C* ∈# *N* + *U* ⟹ *L* ∈# *watched C* ⟹ −*L* ∈ *lits-of-l M* ⟹ ¬*has-blit M* (*clause C*) *L* ⟹
(*L*, *C*) ∉# *WS* ⟹ *L* ∈# *Q*›
shows
‹*clauses-to-update-inv* (*M*, *N*, *U*, *None*, *NE*, *UE*, *WS*, *Q*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*, {#}, {#})›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *past-invs-enqueud*: ‹*past-invs* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) ⟷
past-invs (*M*, *N*, *U*, *D*, *NE*, *UE*, {#}, {#})›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

declare *twl-st-inv.simps*[*simp del*]

lemma *has-blit-Cons*[*simp*]:
assumes *blit*: ‹*has-blit* *M* *C* *L*› **and** *n-d*: ‹*no-dup* (*K* # *M*)›
shows ‹*has-blit* (*K* # *M*) *C* *L*›
⟨*proof*⟩

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

lemma *no-has-blit-propagate*:
‹¬*has-blit* (*Propagated* *L* *D* # *M*) (*W* + *UW*) *La* ⟹
undefined-lit *M* *L* ⟹ *no-dup* *M* ⟹ ¬*has-blit* *M* (*W* + *UW*) *La*›
⟨*proof*⟩

lemma *no-has-blit-propagate'*:
‹¬*has-blit* (*Propagated* *L* *D* # *M*) (*clause* *C*) *La* ⟹
undefined-lit *M* *L* ⟹ *no-dup* *M* ⟹ ¬*has-blit* *M* (*clause* *C*) *La*›
⟨*proof*⟩

lemma *no-has-blit-decide*:
‹¬*has-blit* (*Decided L* # *M*) (*W* + *UW*) *La* ⟹
undefined-lit M L ⟹ *no-dup M* ⟹ ¬*has-blit M* (*W* + *UW*) *La*›
⟨*proof*⟩

lemma *no-has-blit-decide′*:
‹¬*has-blit* (*Decided L* # *M*) (*clause C*) *La* ⟹
undefined-lit M L ⟹ *no-dup M* ⟹ ¬*has-blit M* (*clause C*) *La*›
⟨*proof*⟩

lemma *twl-lazy-update-Propagated*:
assumes
W: ‹*L* ∈# *W*› **and** *n-d*: ‹*no-dup* (*Propagated L D* # *M*)› **and**
lazy: ‹*twl-lazy-update M* (*TWL-Clause W UW*)›
shows
‹*twl-lazy-update* (*Propagated L D* # *M*) (*TWL-Clause W UW*)›
⟨*proof*⟩

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

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

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

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

lemma
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
dist-q: ‹*distinct-queued S*› **and**
ws: ‹*clauses-to-update-inv S*›
shows *twl-cp-twl-st-exception-inv*: ‹*twl-st-exception-inv T*› **and**
twl-cp-clauses-to-update: ‹*clauses-to-update-inv T*›
⟨*proof*⟩

lemma *twl-cp-twl-inv*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**

inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
wq: ‹*clauses-to-update-inv S*›
shows ‹*twl-st-inv T*›
⟨*proof*⟩

lemma *twl-cp-no-duplicate-queued*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
no-dup: ‹*no-duplicate-queued S*›
shows ‹*no-duplicate-queued T*›
⟨*proof*⟩

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

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

lemma *twl-cp-distinct-queued*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
dist: ‹*distinct-queued S*›
shows ‹*distinct-queued T*›
⟨*proof*⟩

lemma *twl-cp-valid*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
dist: ‹*distinct-queued S*›
shows ‹*valid-enqueued T*›
⟨*proof*⟩

lemma *twl-cp-propa-cands-enqueued*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv* ($state_W$*-of S*)› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
cands: ‹*propa-cands-enqueued S*› **and**
ws: ‹*clauses-to-update-inv S*›
shows ‹*propa-cands-enqueued T*›
⟨*proof*⟩

lemma *twl-cp-confl-cands-enqueued*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹cdcl_W*-restart-mset.*cdcl_W*-all-struct-inv* (state_W*-of S*)› **and**
excep: ‹*twl-st-exception-inv S*› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
cands: ‹*confl-cands-enqueued S*› **and**
ws: ‹*clauses-to-update-inv S*›
shows
‹*confl-cands-enqueued T*›
⟨*proof*⟩

lemma *twl-cp-past-invs*:
assumes
cdcl: ‹*cdcl-twl-cp S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹cdcl_W*-restart-mset.*cdcl_W*-all-struct-inv* (state_W*-of S*)› **and**
twl-excep: ‹*twl-st-exception-inv S*› **and**
no-dup: ‹*no-duplicate-queued S*› **and**
past-invs: ‹*past-invs S*›
shows ‹*past-invs T*›
⟨*proof*⟩

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)›
⟨*proof*⟩

lemma *cdcl-twl-o-*cdcl_W*-o*:
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-st-inv S*› **and**
valid: ‹*valid-enqueued S*› **and**
inv: ‹cdcl_W*-restart-mset.*cdcl_W*-all-struct-inv* (state_W*-of S*)›
shows ‹cdcl_W*-restart-mset.*cdcl_W*-o* (state_W*-of S*) (state_W*-of T*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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

lemma *cdcl-twl-cp-entailed-clss-inv*:
‹*cdcl-twl-cp S T* ⟹ *entailed-clss-inv S* ⟹ *entailed-clss-inv T*›
⟨*proof*⟩

lemma *cdcl-twl-cp-init-clss*:
‹*cdcl-twl-cp S T* ⟹ *twl-struct-invs S* ⟹ *init-clss* (*state*$_W$*-of T*) = *init-clss* (*state*$_W$*-of S*)›
⟨*proof*⟩

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

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*⟩

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

The other rules

lemma
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-struct-invs S*›
shows
cdcl-twl-o-twl-st-inv: ‹*twl-st-inv T*› **and**
cdcl-twl-o-past-invs: ‹*past-invs T*›
⟨*proof*⟩

lemma
assumes
cdcl: ‹*cdcl-twl-o S T*›
shows
cdcl-twl-o-valid: ‹*valid-enqueued T*› **and**
cdcl-twl-o-conflict-None-queue:
‹*get-conflict T* ≠ *None* ⟹ *clauses-to-update T* = {#} ∧ *literals-to-update T* = {#}› **and**
cdcl-twl-o-no-duplicate-queued: ‹*no-duplicate-queued T*› **and**
cdcl-twl-o-distinct-queued: ‹*distinct-queued T*›
⟨*proof*⟩

lemma *cdcl-twl-o-twl-st-exception-inv*:
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-struct-invs S*›
shows
‹*twl-st-exception-inv T*›
⟨*proof*⟩

lemma
assumes
cdcl: ‹*cdcl-twl-o S T*› **and**
twl: ‹*twl-struct-invs S*›
shows
cdcl-twl-o-confl-cands-enqueued: ‹*confl-cands-enqueued T*› **and**
cdcl-twl-o-propa-cands-enqueued: ‹*propa-cands-enqueued T*› **and**
twl-o-clauses-to-update: ‹*clauses-to-update-inv T*›
⟨*proof*⟩

lemma *no-dup-append-decided-Cons-lev*:
assumes ‹*no-dup* (*M2* @ *Decided K* # *M1*)›
shows ‹*count-decided M1* = *get-level* (*M2* @ *Decided K* # *M1*) *K* − *1*›
⟨*proof*⟩

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*›
⟨*proof*⟩

The Strategy

lemma *no-literals-to-update-no-cp*:
assumes
WS: ‹*clauses-to-update S* = {#}› **and** *Q*: ‹*literals-to-update S* = {#}› **and**
twl: ‹*twl-struct-invs S*›
shows
‹*no-step* $cdcl_W$*-restart-mset.propagate* ($state_W$*-of S*)› **and**
‹*no-step* $cdcl_W$*-restart-mset.conflict* ($state_W$*-of S*)›
⟨*proof*⟩

When popping a literal from *literals-to-update* to the *clauses-to-update*, we do not do any transition in the abstract transition system. Therefore, we use *rtranclp* or a case distinction.

lemma *cdcl-twl-stgy-*$cdcl_W$*-stgy2*:
assumes ‹*cdcl-twl-stgy S T*› **and** *twl*: ‹*twl-struct-invs S*›
shows ‹$cdcl_W$*-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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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

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

lemma *rtranclp-cdcl-twl-stgy-twl-struct-invs*:
  assumes
    cdcl: ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *S T*› **and**
    twl: ‹*twl-struct-invs S*›
  shows ‹*twl-struct-invs T*›
  ⟨*proof*⟩

lemma *rtranclp-cdcl-twl-stgy-*$cdcl_W$*-stgy*:
  assumes ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *S T*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-*$stgy^{**}$ ($state_W$*-of S*) ($state_W$*-of T*)›
  ⟨*proof*⟩

lemma *no-step-cdcl-twl-cp-no-step-*$cdcl_W$*-cp*:
  assumes *ns-cp*: ‹*no-step cdcl-twl-cp S*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹*literals-to-update S* = {#} ∧ *clauses-to-update S* = {#}›
⟨*proof*⟩

lemma *no-step-cdcl-twl-o-no-step-*$cdcl_W$*-o*:
  assumes
    ns-o: ‹*no-step cdcl-twl-o S*› **and**
    twl: ‹*twl-struct-invs S*› **and**
    p: ‹*literals-to-update S* = {#}› **and**
    w-q: ‹*clauses-to-update S* = {#}›
  shows ‹*no-step* $cdcl_W$*-restart-mset.*$cdcl_W$*-o* ($state_W$*-of S*)›
⟨*proof*⟩

lemma *no-step-cdcl-twl-stgy-no-step-*$cdcl_W$*-stgy*:
  assumes *ns*: ‹*no-step cdcl-twl-stgy S*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹*no-step* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* ($state_W$*-of S*)›
⟨*proof*⟩

lemma *full-cdcl-twl-stgy-*$cdcl_W$*-stgy*:
  assumes ‹*full cdcl-twl-stgy S T*› **and** *twl*: ‹*twl-struct-invs S*›
  shows ‹*full* $cdcl_W$*-restart-mset.*$cdcl_W$*-stgy* ($state_W$*-of S*) ($state_W$*-of T*)›
  ⟨*proof*⟩

definition *init-state-twl* **where**
  ‹*init-state-twl N* ≡ ([], *N*, {#}, *None*, {#}, {#}, {#}, {#})›
lemma
  assumes

struct: ‹∀ *C* ∈# *N*. *struct-wf-twl-cls C*› **and**
tauto: ‹∀ *C* ∈# *N*. ¬*tautology* (*clause C*)›
shows
twl-stgy-invs-init-state-twl: ‹*twl-stgy-invs* (*init-state-twl N*)› **and**
twl-struct-invs-init-state-twl: ‹*twl-struct-invs* (*init-state-twl N*)›
⟨*proof*⟩

lemma *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*⟩

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

Well-foundedness **lemma** *wf-cdcl*$_W$*-stgy-state*$_W$*-of*:
‹*wf* {(*T*, *S*). *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*)}›
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *wf-cdcl-twl-o*:

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

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

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

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

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

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

lemma (**in** −) *rtranclp-cdcl-twl-stgy-twl-stgy-invs*:
assumes
‹$cdcl\text{-}twl\text{-}stgy^{**}$ S T› **and**
‹twl-struct-invs S› **and**
‹twl-stgy-invs S›
shows ‹twl-stgy-invs T›
⟨proof⟩

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⟩

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⟩

lemma *cdcl-twl-stgy-get-init-learned-clss-mono*:
assumes ‹cdcl-twl-stgy S T›
shows ‹get-init-learned-clss S ⊆# get-init-learned-clss T›
⟨proof⟩

lemma *rtranclp-cdcl-twl-stgy-get-init-learned-clss-mono*:
assumes ‹cdcl-twl-stgy** S T›
shows ‹get-init-learned-clss S ⊆# get-init-learned-clss T›
⟨proof⟩

lemma *cdcl-twl-o-all-learned-diff-learned*:
assumes ‹cdcl-twl-o S T›
shows
‹clause '# get-learned-clss S ⊆# clause '# get-learned-clss T ∧
get-init-learned-clss S ⊆# get-init-learned-clss T∧
get-all-init-clss S = get-all-init-clss T›
⟨proof⟩

lemma *cdcl-twl-cp-all-learned-diff-learned*:
assumes ‹cdcl-twl-cp S T›
shows
‹clause '# get-learned-clss S = clause '# get-learned-clss T ∧
get-init-learned-clss S = get-init-learned-clss T ∧
get-all-init-clss S = get-all-init-clss T›
⟨proof⟩

lemma *cdcl-twl-stgy-all-learned-diff-learned*:
assumes ‹cdcl-twl-stgy S T›
shows
‹clause '# get-learned-clss S ⊆# clause '# get-learned-clss T ∧
get-init-learned-clss S ⊆# get-init-learned-clss T∧
get-all-init-clss S = get-all-init-clss T›
⟨proof⟩

lemma *rtranclp-cdcl-twl-stgy-all-learned-diff-learned*:
assumes ‹cdcl-twl-stgy** S T›
shows
‹clause '# get-learned-clss S ⊆# clause '# get-learned-clss T ∧
get-init-learned-clss S ⊆# get-init-learned-clss T ∧
get-all-init-clss S = get-all-init-clss T›
⟨proof⟩

lemma *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)›
⟨proof⟩

lemma *cdcl-twl-stgy-$cdcl_W$-stgy3*:
assumes ‹cdcl-twl-stgy S T› **and** *twl*: ‹twl-struct-invs S› **and**
‹clauses-to-update S = {#}› **and**

‹*literals-to-update S* = {#}›
shows ‹*cdcl$_W$-restart-mset.cdcl$_W$-stgy* (*state$_W$-of S*) (*state$_W$-of T*)›
⟨*proof*⟩

lemma *tranclp-cdcl-twl-stgy-cdcl$_W$-stgy*:
assumes *ST*: ‹*cdcl-twl-stgy^{++} S T*› **and**
twl: ‹*twl-struct-invs S*› **and**
‹*clauses-to-update S* = {#}› **and**
‹*literals-to-update S* = {#}›
shows ‹*cdcl$_W$-restart-mset.cdcl$_W$-stgy^{++}* (*state$_W$-of S*) (*state$_W$-of T*)›
⟨*proof*⟩

definition *final-twl-state* **where**
‹*final-twl-state S* ⟷
no-step cdcl-twl-stgy S ∨ (*get-conflict S* ≠ *None* ∧ *count-decided* (*get-trail S*) = *0*)›

definition *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*⟩

lemma *conflict-of-level-unsatisfiable2*:
assumes
struct: ‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv S*› **and**
dec: ‹*count-decided* (*trail S*) = *0*› **and**
confl: ‹*conflicting S* ≠ *None*›
shows ‹*unsatisfiable* (*set-mset* (*init-clss S* + *learned-clss S*))›
⟨*proof*⟩

end
theory *Watched-Literals-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* = (λ*C* (*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* = (λ*L'* *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 S₀ = do {
    n ← SPEC(λ-::nat. True);
  (S, -) ← WHILE_T^λ(S, n). twl-struct-invs S ∧ twl-stgy-invs S ∧ cdcl-twl-cp** S₀ S ∧        (clauses-to-update S ≠ {#} ∨ n
      (λ(S, n). clauses-to-update S ≠ {#} ∨ n > 0)
      (λ(S, n). do {
        b ← SPEC(λb. (b ⟶ n > 0) ∧ (¬b ⟶ clauses-to-update S ≠ {#}));
        if ¬b then do {
          ASSERT(clauses-to-update S ≠ {#});
          (L, C) ← SPEC (λC. C ∈# clauses-to-update S);
          let S′ = set-clauses-to-update (clauses-to-update S − {#(L, C)#}) S;
          T ← unit-propagation-inner-loop-body L C S′;
          RETURN (T, if get-conflict T = None then n else 0)
        } else do { /This/branch/allows/us/to/do/skip/some/clauses/
          RETURN (S, n − 1)
        }
      })
```

(S_0, n);
RETURN S
}
⟩

lemma *unit-propagation-inner-loop-body*:
fixes S :: ⟨*'v twl-st*⟩
assumes
⟨*clauses-to-update* $S \neq \{\#\}$⟩ **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*⟩

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* $S' = \{\#\}$)⟩
⟨*proof*⟩

declare *unit-propagation-inner-loop*[*THEN order-trans*, *refine-vcg*]

definition *unit-propagation-outer-loop* :: ⟨*'v twl-st* $\Rightarrow$ *'v twl-st nres*⟩ **where**
⟨*unit-propagation-outer-loop* S_0 =
$WHILE_T{}^{\lambda S.\ twl\text{-}struct\text{-}invs\ S\ \wedge\ twl\text{-}stgy\text{-}invs\ S\ \wedge\ cdcl\text{-}twl\text{-}cp^{**}\ S_0\ S\ \wedge\ clauses\text{-}to\text{-}update\ S = \{\#\}}$
(λS. *literals-to-update* $S \neq \{\#\}$)
(λS. *do* {
$L \leftarrow$ *SPEC* (λL. $L \in\#$ *literals-to-update* S);
let S' = *set-clauses-to-update* $\{\#(L, C)|C \in\#$ *get-clauses* S. $L \in\#$ *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' $\equiv$ *twl-struct-invs* S' $\wedge$ *cdcl-twl-cp*** S S' $\wedge$
literals-to-update $S' = \{\#\}$ $\wedge$ ($\forall S'a$. $\neg$ *cdcl-twl-cp* S' $S'a$) $\wedge$ *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* $S \leq$ *SPEC* ($\lambda S'$. *twl-struct-invs* S' $\wedge$ *cdcl-twl-cp*** S S' $\wedge$
literals-to-update $S' = \{\#\}$ $\wedge$ *no-step cdcl-twl-cp* S' $\wedge$ *twl-stgy-invs* S')⟩

⟨*proof*⟩
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*⟩

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 S0 =
  do {
    (-, S) ←
      WHILE_T^{skip-and-resolve-loop-inv S0}
      (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, S0);
    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* = {#})⟩
⟨*proof*⟩

declare *skip-and-resolve-loop-spec*[*THEN order-trans*, *refine-vcg*]

Backtrack

definition *extract-shorter-conflict* :: ⟨′*v twl-st* ⇒ ′*v twl-st nres*⟩ **where**
⟨*extract-shorter-conflict* = (λ(*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*).
SPEC(λ*S*′. ∃ *D*′. *S*′ = (*M*, *N*, *U*, *Some D*′, *NE*, *UE*, *WS*, *Q*) ∧
D′ ⊆# *the D* ∧ *clause* ‘# (*N* + *U*) + *NE* + *UE* ⊨*pm D*′ ∧ −*lit-of* (*hd M*) ∈# *D*′))⟩

fun *equality-except-conflict* :: ⟨′*v twl-st* ⇒ ′*v twl-st* ⇒ *bool*⟩ **where**
⟨*equality-except-conflict* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) (*M*′, *N*′, *U*′, *D*′, *NE*′, *UE*′, *WS*′, *Q*′) ⟷
M = *M*′ ∧ *N* = *N*′ ∧ *U* = *U*′ ∧ *NE* = *NE*′ ∧ *UE* = *UE*′ ∧ *WS* = *WS*′ ∧ *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-of (hd (get-trail S)) $\in\#$ D')›
⟨proof⟩

definition reduce-trail-bt :: ‹'v literal ⇒ 'v twl-st ⇒ 'v twl-st nres› **where**
‹reduce-trail-bt = (λL (M, N, U, D', NE, UE, WS, Q). do {
    $M1$ ← $SPEC(\lambda M1. \exists K\ M2.$ (Decided K # $M1$, $M2$) ∈ set (get-all-ann-decomposition M) ∧
        get-level M K = get-maximum-level M (the D' − {#−L#}) + 1);
    RETURN ($M1$, N, U, D', NE, UE, WS, Q)
})›

definition propagate-bt :: ‹'v literal ⇒ 'v literal ⇒ 'v twl-st ⇒ '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-mset (the D) UE, WS, {#L#}))›

definition backtrack-inv **where**
‹backtrack-inv S ⟷ get-trail S ≠ [] ∧ get-conflict S ≠ 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 ← extract-shorter-conflict S;
    S ← reduce-trail-bt L S;

    if size (the (get-conflict S)) > 1
    then do {
      L' ← $SPEC(\lambda L'. L' \in\#$ the (get-conflict S) − {#−L#} ∧ L ≠ −L' ∧
        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 ≠ None› ‹get-conflict S ≠ Some {#}› **and**
  w-q: ‹clauses-to-update S = {#}› **and** p: ‹literals-to-update S = {#}› **and**
  ns-s: ‹no-step $cdcl_W$-restart-mset.skip ($state_W$-of S)› **and**
  ns-r: ‹no-step $cdcl_W$-restart-mset.resolve ($state_W$-of S)› **and**
  twl-struct: ‹twl-struct-invs S› **and** twl-stgy: ‹twl-stgy-invs S›
 shows
  backtrack-spec:
  ‹backtrack S ≤ SPEC (λT. cdcl-twl-o S T ∧ get-conflict T = None ∧ no-step cdcl-twl-o T ∧

twl-struct-invs T ∧ *twl-stgy-invs T* ∧ *clauses-to-update T* = {#} ∧
literals-to-update T ≠ {#})› (**is** *?spec*) **and**
backtrack-nofail:
‹*nofail* (*backtrack S*)› (**is** *?fail*)
⟨*proof*⟩

declare *backtrack-spec*[*THEN order-trans*, *refine-vcg*]

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*)›
⟨*proof*⟩

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

lemma *rtranclp-skip-and-resolve-same-decision-level*:
‹$cdcl\text{-}twl\text{-}o^{**}$ *S T* ⟹ *get-conflict S* ≠ *None* ⟹ *get-conflict T* ≠ *None* ⟹
count-decided (*get-trail T*) = *count-decided* (*get-trail S*)›
⟨*proof*⟩

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

abbreviation *cdcl-twl-o-prog-spec* **where**
‹*cdcl-twl-o-prog-spec S* ≡ λ(*brk*, *T*).
$cdcl\text{-}twl\text{-}o^{**}$ *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-twl-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*⟩

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 ∧ *cdcl-twl-stgy*$^{++}$ S T}
∪ {((*brkT*, T), (*brkS*, S)). S = T ∧ *brkT* ∧ ¬*brkS*})›
(**is** ‹*wf* (?*TWL* ∪ ?*BOOL*)›)
⟨*proof*⟩

lemma *cdcl-twl-o-final-twl-state*:
assumes
‹*cdcl-twl-stgy-prog-inv* S (*brk*, T)› **and**
‹*case* (*brk*, T) *of* (*brk*, -) ⇒ ¬ *brk*› **and**
twl-o: ‹*cdcl-twl-o-prog-spec* U (*True*, V)›
shows ‹*final-twl-state* V›
⟨*proof*⟩

lemma *cdcl-twl-stgy-in-measure*:

assumes
twl-stgy: ‹*cdcl-twl-stgy-prog-inv S* (*brk0*, *T*)› **and**
brk0: ‹*case* (*brk0*, *T*) *of* (*brk*, *uu-*) ⇒ ¬ *brk*› **and**
twl-o: ‹*cdcl-twl-o-prog-spec U V*› **and**
[*simp*]: ‹*twl-struct-invs U*› **and**
TU: ‹$cdcl\text{-}twl\text{-}cp^{**}$ *T U*› **and**
‹*literals-to-update U* = {#}›
shows ‹(*V*, *brk0*, *T*)
∈ {((*brkT*, *T*), *brkS*, *S*). *twl-struct-invs S* ∧ $cdcl\text{-}twl\text{-}stgy^{++}$ *S T*} ∪
{((*brkT*, *T*), *brkS*, *S*). *S* = *T* ∧ *brkT* ∧ ¬ *brkS*}›
⟨*proof*⟩

lemma *cdcl-twl-o-prog-cdcl-twl-stgy*:
assumes
twl-stgy: ‹*cdcl-twl-stgy-prog-inv S* (*brk*, *S'*)› **and**
‹*case* (*brk*, *S'*) *of* (*brk*, *uu-*) ⇒ ¬ *brk*› **and**
twl-o: ‹*cdcl-twl-o-prog-spec T* (*brk'*, *U*)› **and**
‹*twl-struct-invs T*› **and**
cp: ‹$cdcl\text{-}twl\text{-}cp^{**}$ *S' T*› **and**
‹*literals-to-update T* = {#}› **and**
‹∀ *S'*. ¬ *cdcl-twl-cp T S'*› **and**
‹*twl-stgy-invs T*›
shows ‹$cdcl\text{-}twl\text{-}stgy^{**}$ *S U*›
⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-spec*:
assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and**
‹*get-conflict S* = *None*›
shows
‹*cdcl-twl-stgy-prog S* ≤ *conclusive-TWL-run S*›
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-break* :: ‹*'v twl-st* ⇒ *'v twl-st nres*› **where**
‹*cdcl-twl-stgy-prog-break* S_0 =
do {
b ← *SPEC*(λ-. *True*);
(*b*, *brk*, *T*) ← $WHILE_T{}^{\lambda(b,\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}inv\ S_0\ S}$
(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* ∧ $cdcl\text{-}twl\text{-}stgy^{++}$ *S T*} ∪
{((*bT*, *brkT*, *T*), (*bS*, *brkS*, *S*)). *S* = *T* ∧ *brkT* ∧ ¬*brkS*}
)›

(**is** ‹?wf ?R›)
⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-break-spec*:
assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹*clauses-to-update S* = {#}› **and**
‹*get-conflict S* = *None*›
shows
‹*cdcl-twl-stgy-prog-break S* ≤ *conclusive-TWL-run S*›
⟨*proof*⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

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

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

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

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

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
‹$cdcl\text{-}twl\text{-}stgy^{++}$ *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*)›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-stgy-restart-init-clss*:
assumes ‹$cdcl\text{-}twl\text{-}stgy\text{-}restart^{**}$ *S T*›
shows
‹*get-all-init-clss* (*fst S*) = *get-all-init-clss* (*fst T*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-stgy-restart-twl-struct-invs*:
assumes
‹$cdcl\text{-}twl\text{-}stgy\text{-}restart^{**}$ *S T*› **and**
‹*twl-struct-invs* (*fst S*)›
shows ‹*twl-struct-invs* (*fst T*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma (**in** −) *substract-left-le*: ‹(*a* :: *nat*) + *b* < *c* ==> *a* <= *c* − *b*›
⟨*proof*⟩

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*⟩

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

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*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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)›
⟨*proof*⟩

lemma *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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

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)›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-cdcl-twl-stgy-cdcl-twl-stgy-restart2*:
‹*cdcl-twl-stgy-restart* (T, m) (V, m) $\Longrightarrow$
cdcl-twl-stgy** S T $\Longrightarrow$ *cdcl-twl-stgy-restart* (S, m) (V, m)›
⟨*proof*⟩

definition *cdcl-twl-stgy-restart-with-leftovers1* **where**
‹*cdcl-twl-stgy-restart-with-leftovers1* S U $\longleftrightarrow$
cdcl-twl-stgy-restart S U $\vee$
(*cdcl-twl-stgy*$^{++}$ (*fst* S) (*fst* U) $\wedge$ *snd* S = *snd* U)›

lemma (**in** *twl-restart*) *wf-cdcl-twl-stgy-restart-with-leftovers1*:
‹*wf* {(T :: *'v twl-st* $\times$ *nat*, S).
twl-struct-invs (*fst* S) $\wedge$ *cdcl-twl-stgy-restart-with-leftovers1* S T}›
(**is** ‹*wf ?S*›)
⟨*proof*⟩

lemma (**in** *twl-restart*) *wf-cdcl-twl-stgy-restart-measure*:
‹*wf* ({((*brkT*, T, n), *brkS*, S, m).
twl-struct-invs S $\wedge$ *cdcl-twl-stgy-restart-with-leftovers1* (S, m) (T, n)} $\cup$
{((*brkT*, T), *brkS*, S). S = T $\wedge$ *brkT* $\wedge$ $\neg$ *brkS*})›
(**is** ‹*wf* (*?TWL* $\cup$ *?BOOL*)›)
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs*:
‹*cdcl-twl-stgy-restart-with-leftovers S T* ⟹ *twl-struct-invs* (*fst S*) ⟹
twl-struct-invs (*fst T*)›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-stgy-restart-with-leftovers-twl-struct-invs*:
‹*cdcl-twl-stgy-restart-with-leftovers*** *S T* ⟹ *twl-struct-invs* (*fst S*) ⟹
twl-struct-invs (*fst T*)›
⟨*proof*⟩

lemma *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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*, -) ← $WHILE_T$^λ(*brk*, *T*, *n*). *cdcl-twl-stgy-restart-prog-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-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-spec*:
  ⟨*restart-prog U m brk'*
   ≤ *SPEC*
    (λ*x*. (*case x of*

$(T, na) \Rightarrow RETURN\ (brk', T, na))$
$\leq SPEC$
$(\lambda s'. (case\ s'\ of$
$(brk, T, n) \Rightarrow$
$twl\text{-}struct\text{-}invs\ T \wedge$
$twl\text{-}stgy\text{-}invs\ T \wedge$
$(brk \longrightarrow final\text{-}twl\text{-}state\ T) \wedge$
$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}with\text{-}leftovers\ (S, 0)$
$(T, n) \wedge$
$clauses\text{-}to\text{-}update\ T = \{\#\} \wedge$
$(\neg\ brk \longrightarrow get\text{-}conflict\ T = None)) \wedge$
(s', brk, T, m)
$\in \{((brkT, T, n), brkS, S, m).$
$twl\text{-}struct\text{-}invs\ S \wedge$
$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}with\text{-}leftovers1\ (S, m)$
$(T, n)\} \cup$
$\{((brkT, T), brkS, S).\ S = T \wedge brkT \wedge \neg\ brkS\}))$› (**is** *?A*)
⟨*proof*⟩

lemma (**in** *twl-restart*)
assumes
inv: ‹*case* $(ebrk, brk, T, m)$ *of* $(ebrk, brk, T, m) \Rightarrow cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S\ brk\ T\ m$› **and**
cond: ‹*case* $(ebrk, brk, T, m)$ *of* $(ebrk, brk, \text{-}) \Rightarrow \neg\ brk \wedge \neg ebrk$› **and**
other-inv: ‹$cdcl\text{-}twl\text{-}o\text{-}prog\text{-}spec\ S'\ (brk', U)$› **and**
struct-invs-S: ‹$twl\text{-}struct\text{-}invs\ S'$› **and**
cp: ‹$cdcl\text{-}twl\text{-}cp^{**}\ T\ S'$› **and**
lits-to-update: ‹$literals\text{-}to\text{-}update\ S' = \{\#\}$› **and**
‹$\forall S'a.\ \neg\ cdcl\text{-}twl\text{-}cp\ S'\ S'a$› **and**
‹$twl\text{-}stgy\text{-}invs\ S'$›
shows *restart-prog-early-spec*:
‹$restart\text{-}prog\ U\ m\ brk'$
$\leq SPEC$
$(\lambda x. (case\ x\ of\ (T, n) \Rightarrow RES\ UNIV \gg= (\lambda ebrk.\ RETURN\ (ebrk, brk', T, n)))$
$\leq SPEC$
$(\lambda s'. (case\ s'\ of\ (ebrk, brk, x, xb) \Rightarrow$
$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S\ brk\ x\ xb) \wedge$
$(s', ebrk, brk, T, m)$
$\in \{((ebrk, brkT, T, n), ebrk, brkS, S, m).$
$twl\text{-}struct\text{-}invs\ S \wedge$
$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}with\text{-}leftovers1\ (S, m)\ (T, n)\} \cup$
$\{((ebrkT, brkT, T), ebrkS, brkS, S).$
$S = T \wedge (ebrkT \vee brkT) \wedge \neg\ brkS \wedge \neg\ ebrkS\}))$› (**is** ‹*?B*›)
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-with-leftovers-refl*: ‹*cdcl-twl-stgy-restart-with-leftovers S S*›
⟨*proof*⟩

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-spec*:
assumes ‹*twl-struct-invs S*› **and** ‹*twl-stgy-invs S*› **and** ‹$clauses\text{-}to\text{-}update\ S = \{\#\}$› **and**
‹$get\text{-}conflict\ S = None$›
shows
‹$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\ S \leq SPEC(\lambda T.\ \exists n.\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}with\text{-}leftovers\ (S, 0)\ (T, n) \wedge$
$final\text{-}twl\text{-}state\ T)$›
(**is** ‹$\text{-} \leq SPEC(\lambda T.\ ?P\ T)$›)
⟨*proof*⟩

definition *cdcl-twl-stgy-restart-prog-early* :: *'v twl-st* ⇒ *'v twl-st nres* **where**
⟨*cdcl-twl-stgy-restart-prog-early* S_0 =
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, *T*, *n*) ← *WHILE*$_T$$^{\lambda(ebrk,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S_0\ brk\ T\ n}$
(λ(*ebrk*, *brk*, -). ¬*brk* ∧ ¬*ebrk*)
(λ(*ebrk*, *brk*, *S*, *n*).
do {
T ← *unit-propagation-outer-loop S*;
(*brk*, *T*) ← *cdcl-twl-o-prog T*;
(*T*, *n*) ← *restart-prog T n brk*;
ebrk ← *RES UNIV*;
RETURN (*ebrk*, *brk*, *T*, *n*)
})
(*ebrk*, *False*, S_0, *0*);
if ¬*brk then do* {
(*brk*, *T*, -) ← *WHILE*$_T$$^{\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*, *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*(λ*T*. ∃ *n*. *cdcl-twl-stgy-restart-with-leftovers* (*S*, *0*) (*T*, *n*)
∧
final-twl-state T)⟩
(**is** ⟨- ≤ *SPEC*(λ*T*. *?P T*)⟩)
⟨*proof*⟩

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*) ← *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*);
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*⟩
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*: ‹(λ*x*. *mset* (*take 2 x*) + *mset* (*drop 2 x*)) = *mset*›
⟨*proof*⟩
lemma *mset-take-mset-drop-mset′*: ‹*mset* (*take 2 x*) + *mset* (*drop 2 x*) = *mset x*›
⟨*proof*⟩

lemma *uminus-lit-of-image-mset*:
‹{#− *lit-of x* . *x* ∈# *A*#} = {#− *lit-of x*. *x* ∈# *B*#} ⟷
{#*lit-of x* . *x* ∈# *A*#} = {#*lit-of x*. *x* ∈# *B*#}›
for *A* :: ‹(′*a literal*, ′*a literal*, ′*b*) *annotated-lit multiset*›
⟨*proof*⟩

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)›
⟨proof⟩

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)›
⟨proof⟩

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

lemma *convert-lits-l-cons*[*simp*]:
‹*(L # M, L' # M') ∈ convert-lits-l N E ⟷*
convert-lit N E L L' ∧ (M, M') ∈ convert-lits-l N E›
⟨*proof*⟩

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

lemma *convert-lits-l-consE*:
‹*(Propagated L C # M, x) ∈ convert-lits-l N E ⟹*
(⋀L' C' M'. x = Propagated L' C' # M' ⟹ (M, M') ∈ convert-lits-l N E ⟹
convert-lit N E (Propagated L C) (Propagated L' C') ⟹ P) ⟹ P›

⟨proof⟩

lemma *convert-lits-l-append*[*simp*]:
‹*length M1* = *length M1'* ⟹
(*M1* @ *M2*, *M1'* @ *M2'*) ∈ *convert-lits-l N E* ⟷ (*M1*, *M1'*) ∈ *convert-lits-l N E* ∧
(*M2*, *M2'*) ∈ *convert-lits-l N E* ›
⟨proof⟩

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

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

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

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

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

abbreviation *resolve-cls-l* **where**
‹*resolve-cls-l L D' E* ≡ *union-mset-list* (*remove1* (−*L*) *D'*) (*remove1 L E*)›

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

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

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

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

lemma *defined-lit-convert-lits-l*[*simp*]: ‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
defined-lit M' = *defined-lit M*›
⟨proof⟩

lemma *no-dup-convert-lits-l*[*simp*]: ‹(*M*, *M'*) ∈ *convert-lits-l N E* ⟹
no-dup M' ⟷ *no-dup M*›
⟨proof⟩

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

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

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

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

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

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

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

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

The order of the assumption is important for simpler use.

lemma *convert-lits-l-extend-mono*:
assumes ‹$(a,b) \in$ *convert-lits-l N E*›
‹$\forall L\ i.$ *Propagated* $L\ i \in$ *set* $a \longrightarrow$ *mset* $(N{\propto}i)$ = *mset* $(N'{\propto}i)$› **and** ‹$E \subseteq\#\ E'$›
shows
‹$(a,b) \in$ *convert-lits-l* $N'\ E'$›

⟨proof⟩

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

lemma *convert-lits-l-atm-lits-of-l*:
assumes ‹(*M*, *M'*) ∈ *convert-lits-l N E*›
shows ‹*atm-of* ‘ *lits-of-l M* = *atm-of* ‘ *lits-of-l M'*›
⟨proof⟩

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

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

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

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

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

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

lemma *convert-lits-l-decomp-ex*:
assumes
H: ‹(*Decided K* # *a*, *M2*) ∈ *set* (*get-all-ann-decomposition x*)› **and**
xxa: ‹(*x*, *xa*) ∈ *convert-lits-l aa ac*›
shows ‹∃ *M2*. (*Decided K* # *drop* (*length xa* − *length a*) *xa*, *M2*)
∈ *set* (*get-all-ann-decomposition xa*)› (**is** *?decomp*) **and**
‹(*a*, *drop* (*length xa* − *length a*) *xa*) ∈ *convert-lits-l aa ac*› (**is** *?a*)
⟨proof⟩

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

lemma *in-convert-lits-lD2*:
‹*K* ∈ *set M* ⟹
(*M*, *TM*) ∈ *convert-lits-l N NE* ⟹

$\exists K'. K' \in set\ TM \land convert\text{-}lit\ N\ NE\ K\ K'$›
⟨*proof*⟩

lemma *convert-lits-l-filter-decided*: ‹$(S, S') \in convert\text{-}lits\text{-}l\ M\ N \Longrightarrow$
$map\ lit\text{-}of\ (filter\ is\text{-}decided\ S') = map\ lit\text{-}of\ (filter\ is\text{-}decided\ S)$›
⟨*proof*⟩

lemma *convert-lits-lI*:
‹$length\ M = length\ M' \Longrightarrow (\bigwedge i.\ i < length\ M \Longrightarrow convert\text{-}lit\ N\ NE\ (M!i)\ (M'!i)) \Longrightarrow$
$(M, M') \in convert\text{-}lits\text{-}l\ N\ NE$›
⟨*proof*⟩

abbreviation *ran-mf* :: ‹$'v\ clauses\text{-}l \Rightarrow 'v\ clause\text{-}l\ multiset$› **where**
‹$ran\text{-}mf\ N \equiv fst\ `\#\ ran\text{-}m\ N$›

abbreviation *learned-clss-l* :: ‹$'v\ clauses\text{-}l \Rightarrow ('v\ clause\text{-}l \times bool)\ multiset$› **where**
‹$learned\text{-}clss\text{-}l\ N \equiv \{\#C \in\#\ ran\text{-}m\ N.\ \neg snd\ C\#\}$›

abbreviation *learned-clss-lf* :: ‹$'v\ clauses\text{-}l \Rightarrow 'v\ clause\text{-}l\ multiset$› **where**
‹$learned\text{-}clss\text{-}lf\ N \equiv fst\ `\#\ learned\text{-}clss\text{-}l\ N$›

definition *get-learned-clss-l* **where**
‹$get\text{-}learned\text{-}clss\text{-}l\ S = learned\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}l\ S)$›

abbreviation *init-clss-l* :: ‹$'v\ clauses\text{-}l \Rightarrow ('v\ clause\text{-}l \times bool)\ multiset$› **where**
‹$init\text{-}clss\text{-}l\ N \equiv \{\#C \in\#\ ran\text{-}m\ N.\ snd\ C\#\}$›

abbreviation *init-clss-lf* :: ‹$'v\ clauses\text{-}l \Rightarrow 'v\ clause\text{-}l\ multiset$› **where**
‹$init\text{-}clss\text{-}lf\ N \equiv fst\ `\#\ init\text{-}clss\text{-}l\ N$›

abbreviation *all-clss-l* :: ‹$'v\ clauses\text{-}l \Rightarrow ('v\ clause\text{-}l \times bool)\ multiset$› **where**
‹$all\text{-}clss\text{-}l\ N \equiv init\text{-}clss\text{-}l\ N + learned\text{-}clss\text{-}l\ N$›

lemma *all-clss-l-ran-m*[*simp*]:
‹$all\text{-}clss\text{-}l\ N = ran\text{-}m\ N$›
⟨*proof*⟩

abbreviation *all-clss-lf* :: ‹$'v\ clauses\text{-}l \Rightarrow 'v\ clause\text{-}l\ multiset$› **where**
‹$all\text{-}clss\text{-}lf\ N \equiv init\text{-}clss\text{-}lf\ N + learned\text{-}clss\text{-}lf\ N$›

lemma *all-clss-lf-ran-m*: ‹$all\text{-}clss\text{-}lf\ N = fst\ `\#\ ran\text{-}m\ N$›
⟨*proof*⟩

abbreviation *irred* :: ‹$'v\ clauses\text{-}l \Rightarrow nat \Rightarrow bool$› **where**
‹$irred\ N\ C \equiv snd\ (the\ (fmlookup\ N\ C))$›

definition *irred'* **where** ‹$irred' = irred$›

lemma *ran-m-ran*: ‹$fset\text{-}mset\ (ran\text{-}m\ N) = fmran\ N$›
⟨*proof*⟩

fun *get-learned-clauses-l* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow 'v\ clause\text{-}l\ multiset$› **where**
‹$get\text{-}learned\text{-}clauses\text{-}l\ (M, N, D, NE, UE, WS, Q) = learned\text{-}clss\text{-}lf\ N$›

lemma *ran-m-clause-upd*:
assumes

NC: ‹*C* ∈# *dom-m N*›
shows ‹*ran-m* (*N*(*C* ↪ *C'*)) =
add-mset (*C'*, *irred N C*) (*remove1-mset* (*N* ∝ *C*, *irred N C*) (*ran-m N*))›
⟨*proof*⟩

lemma *ran-m-mapsto-upd*:
assumes
NC: ‹*C* ∈# *dom-m N*›
shows ‹*ran-m* (*fmupd C C' N*) =
add-mset C' (*remove1-mset* (*N* ∝ *C*, *irred N C*) (*ran-m N*))›
⟨*proof*⟩

lemma *ran-m-mapsto-upd-notin*:
assumes
NC: ‹*C* ∉# *dom-m N*›
shows ‹*ran-m* (*fmupd C C' N*) = *add-mset C'* (*ran-m N*)›
⟨*proof*⟩

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

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

lemma *Ball-ran-m-dom-struct-wf*:
‹(∀ *x*∈#*ran-m N*. *struct-wf-twl-cls* (*twl-clause-of* (*fst x*))) ⟷
(∀ *x*∈# *dom-m N*. *struct-wf-twl-cls* (*twl-clause-of* (*N* ∝ *x*)))›
⟨*proof*⟩

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

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

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

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

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

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

lemma *ran-mf-lf-fmdrop-notin*[*simp*]:
‹$C \notin\# \ dom\text{-}m\ N \Longrightarrow\ ran\text{-}mf\ (fmdrop\ C\ N) = ran\text{-}mf\ N$›
⟨*proof*⟩

lemma *lookup-None-notin-dom-m*[*simp*]:
‹$fmlookup\ N\ i = None \longleftrightarrow i \notin\#\ dom\text{-}m\ N$›
⟨*proof*⟩

While it is tempting to mark the two following theorems as [simp], this would break more simplifications since *ran-mf* is only an abbreviation for *ran-m*.

lemma *ran-m-fmdrop*:
‹$C \in\#\ dom\text{-}m\ N \Longrightarrow\ ran\text{-}m\ (fmdrop\ C\ N) = remove1\text{-}mset\ (N \propto C,\ irred\ N\ C)\ (ran\text{-}m\ N)$›
⟨*proof*⟩

lemma *ran-m-fmdrop-notin*:
‹$C \notin\#\ dom\text{-}m\ N \Longrightarrow ran\text{-}m\ (fmdrop\ C\ N) = ran\text{-}m\ N$›
⟨*proof*⟩

lemma *init-clss-l-fmdrop-irrelev*:
‹$\neg irred\ N\ C \Longrightarrow init\text{-}clss\text{-}l\ (fmdrop\ C\ N) = init\text{-}clss\text{-}l\ N$›
⟨*proof*⟩

lemma *init-clss-l-fmdrop*:
‹$irred\ N\ C \Longrightarrow C \in\#\ dom\text{-}m\ N \Longrightarrow init\text{-}clss\text{-}l\ (fmdrop\ C\ N) = remove1\text{-}mset\ (the\ (fmlookup\ N\ C))\ (init\text{-}clss\text{-}l\ N)$›
⟨*proof*⟩

lemma *ran-m-lf-fmdrop*:
‹$C \in\#\ dom\text{-}m\ N \Longrightarrow ran\text{-}m\ (fmdrop\ C\ N) = remove1\text{-}mset\ (the\ (fmlookup\ N\ C))\ (ran\text{-}m\ N)$›
⟨*proof*⟩

definition *twl-st-l* :: ‹$\text{-} \Rightarrow ('v\ twl\text{-}st\text{-}l \times 'v\ twl\text{-}st)\ set$› **where**
‹$twl\text{-}st\text{-}l\ L =$
$\{((M,\ N,\ C,\ NE,\ UE,\ WS,\ Q),\ (M',\ N',\ U',\ C',\ NE',\ UE',\ WS',\ Q')).$
$(M,\ M') \in convert\text{-}lits\text{-}l\ N\ (NE{+}UE)\ \wedge$
$N' = twl\text{-}clause\text{-}of\ `\#\ init\text{-}clss\text{-}lf\ N\ \wedge$
$U' = twl\text{-}clause\text{-}of\ `\#\ learned\text{-}clss\text{-}lf\ N\ \wedge$
$C' = C\ \wedge$
$NE' = NE\ \wedge$
$UE' = UE\ \wedge$
$WS' = (case\ L\ of\ None \Rightarrow \{\#\}\ |\ Some\ L \Rightarrow image\text{-}mset\ (\lambda j.\ (L,\ twl\text{-}clause\text{-}of\ (N \propto j)))\ WS)\ \wedge$
$Q' = Q$
$\}$›

lemma *clss-state$_W$-of*[*twl-st*]:
assumes ‹$(S,\ R) \in twl\text{-}st\text{-}l\ L$›
shows
‹$init\text{-}clss\ (state_W\text{-}of\ R) = mset\ `\#\ (init\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}l\ S)) +$
$get\text{-}unit\text{-}init\text{-}clauses\text{-}l\ S$›
‹$learned\text{-}clss\ (state_W\text{-}of\ R) = mset\ `\#\ (learned\text{-}clss\text{-}lf\ (get\text{-}clauses\text{-}l\ S)) +$
$get\text{-}unit\text{-}learned\text{-}clauses\text{-}l\ S$›
⟨*proof*⟩

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

lemma [*twl-st-l*]:
assumes ‹$(S,\ T) \in$ *twl-st-l* L›
shows
‹(*get-trail-l* S, *get-trail* T) $\in$ *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* $\Longrightarrow$ *clauses-to-update* T = {#}›
‹$L \neq$ *None* $\Longrightarrow$ *clauses-to-update* T =
(λ*j*. (*the* L, *twl-clause-of* (*get-clauses-l* $S \propto j$))) ‘# *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) $\longleftrightarrow$ *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 = [] $\longleftrightarrow$ *get-trail-l* S = []› **and**
‹*get-trail* $T \neq$ [] $\longleftrightarrow$ *get-trail-l* $S \neq$ []› **and**
‹*get-trail* $T \neq$ [] $\Longrightarrow$ *is-proped* (*hd* (*get-trail* T)) $\longleftrightarrow$ *is-proped* (*hd* (*get-trail-l* S))›
‹*get-trail* $T \neq$ [] $\Longrightarrow$ *is-decided* (*hd* (*get-trail* T)) $\longleftrightarrow$ *is-decided* (*hd* (*get-trail-l* S))›
‹*get-trail* $T \neq$ [] $\Longrightarrow$ *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 $\models$*as* D $\longleftrightarrow$ *get-trail-l* S $\models$*as* D›
⟨*proof*⟩

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›
⟨*proof*⟩

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

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

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›
⟨*proof*⟩

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

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

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

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

lemma *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*›
⟨*proof*⟩

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

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

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

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

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

lemma *clauses-to-update-l-remove-one-lit-from-wq*[*simp*]:
‹*clauses-to-update-l* (*remove-one-lit-from-wq L T*) = *remove1-mset L* (*clauses-to-update-l T*)›
⟨*proof*⟩

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

lemma *unit-init-clauses-get-unit-init-clauses-l*[*twl-st-l*]:
‹$(S, T) \in$ *twl-st-l L* $\Longrightarrow$ *unit-init-clauses T* = *get-unit-init-clauses-l S*›
⟨*proof*⟩

lemma *clauses-state-to-l*[*twl-st-l*]: ‹$(S, S') \in$ *twl-st-l L* $\Longrightarrow$
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*›
⟨*proof*⟩

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

lemma *hd-get-trail-twl-st-of-get-trail-l*:
‹$(S, T) \in$ *twl-st-l L* $\Longrightarrow$ *get-trail-l S* $\neq$ [] $\Longrightarrow$
lit-of (*hd* (*get-trail T*)) = *lit-of* (*hd* (*get-trail-l S*))›

⟨proof⟩

lemma *twl-st-l-mark-of-hd*:
‹$(x, y) \in$ *twl-st-l* $b \Longrightarrow$
get-trail-l $x \neq [] \Longrightarrow$
is-proped (*hd* (*get-trail-l* x)) $\Longrightarrow$
mark-of (*hd* (*get-trail-l* x)) $> 0 \Longrightarrow$
mark-of (*hd* (*get-trail* y)) $=$ *mset* (*get-clauses-l* $x \propto$ *mark-of* (*hd* (*get-trail-l* x)))›
⟨proof⟩

lemma *twl-st-l-lits-of-tl*:
‹$(x, y) \in$ *twl-st-l* $b \Longrightarrow$
lits-of-l (*tl* (*get-trail* y)) $=$ (*lits-of-l* (*tl* (*get-trail-l* x)))›
⟨proof⟩

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

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))›
⟨proof⟩

fun *equality-except-trail* :: ‹$'v$ *twl-st-l* $\Rightarrow$ $'v$ *twl-st-l* $\Rightarrow$ *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* $\Rightarrow$ $'v$ *twl-st-l* $\Rightarrow$ *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›
⟨proof⟩

lemma *equality-except-conflict-l-alt-def*:
‹*equality-except-conflict-l* S T $\longleftrightarrow$
get-trail-l $S =$ *get-trail-l* T $\wedge$ *get-clauses-l* $S =$ *get-clauses-l* T $\wedge$
get-unit-init-clauses-l $S =$ *get-unit-init-clauses-l* T $\wedge$
get-unit-learned-clauses-l $S =$ *get-unit-learned-clauses-l* T $\wedge$
literals-to-update-l $S =$ *literals-to-update-l* T $\wedge$
clauses-to-update-l $S =$ *clauses-to-update-l* T›
⟨proof⟩

lemma *equality-except-conflict-alt-def*:
‹*equality-except-conflict* S T $\longleftrightarrow$
get-trail $S =$ *get-trail* T $\wedge$ *get-init-clauses* $S =$ *get-init-clauses* T $\wedge$
get-learned-clss $S =$ *get-learned-clss* T $\wedge$
get-init-learned-clss $S =$ *get-init-learned-clss* T $\wedge$

unit-init-clauses S = unit-init-clauses T ∧
literals-to-update S = literals-to-update T ∧
clauses-to-update S = clauses-to-update T›
⟨*proof*⟩

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*) ⟹ *undefined-lit M L*›
⟨*proof*⟩

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

lemma *polarity-spec′*:
assumes ‹*no-dup M*›
shows
‹*polarity M L* = *None* ⟷ *undefined-lit M L*› **and**
‹*polarity M L* = *Some True* ⟷ *L* ∈ *lits-of-l M*› **and**
‹*polarity M L* = *Some False* ⟷ −*L* ∈ *lits-of-l M*›
⟨*proof*⟩

definition *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 ∈# *dom-m* (*get-clauses-l S*) ∧
C > *0* ∧
0 < *length* (*get-clauses-l S* ∝ *C*) ∧
no-dup (*get-trail-l S*) ∧
(*if* (*get-clauses-l S* ∝ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* ∝ *C*) ∧
1 − (*if* (*get-clauses-l S* ∝ *C*) ! *0* = *L then 0 else 1*) < *length* (*get-clauses-l S* ∝ *C*) ∧
L ∈ *set* (*watched-l* (*get-clauses-l S* ∝ *C*)) ∧
get-conflict-l S = *None*
)
⟩

definition *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**

$\langle y \leq \Downarrow \{(S, S'). P\ S\ S'\}\ (f\ S)\rangle$
shows $\langle y \leq \Downarrow \{(S, S'). P\ S\ S' \wedge Q\ S'\}\ (f\ S)\rangle$
⟨*proof*⟩

lemma *clauses-tuple*[*simp*]:
$\langle cdcl_W\text{-}restart\text{-}mset.clauses\ (M, \{\#f\ x\ .\ x \in\#\ init\text{-}clss\text{-}l\ N\#\} + NE, \{\#f\ x\ .\ x \in\#\ learned\text{-}clss\text{-}l\ N\#\} + UE, D) = \{\#f\ x.\ x \in\#\ all\text{-}clss\text{-}l\ N\#\} + NE + UE\rangle$
⟨*proof*⟩

lemma *valid-enqueued-alt-simps*[*simp*]:
$\langle valid\text{-}enqueued\ S \longleftrightarrow (\forall (L, C) \in\#\ clauses\text{-}to\text{-}update\ S.\ L \in\#\ watched\ C \wedge C \in\#\ get\text{-}clauses\ S \wedge -L \in lits\text{-}of\text{-}l\ (get\text{-}trail\ S) \wedge get\text{-}level\ (get\text{-}trail\ S)\ L = count\text{-}decided\ (get\text{-}trail\ S)) \wedge (\forall L \in\#\ literals\text{-}to\text{-}update\ S.\ -L \in lits\text{-}of\text{-}l\ (get\text{-}trail\ S) \wedge get\text{-}level\ (get\text{-}trail\ S)\ L = count\text{-}decided\ (get\text{-}trail\ S))\rangle$
⟨*proof*⟩

declare *valid-enqueued.simps*[*simp del*]

lemma *set-clauses-simp*[*simp*]:
$\langle f\ `\ \{a.\ a \in\#\ ran\text{-}m\ N \wedge \neg\ snd\ a\} \cup f\ `\ \{a.\ a \in\#\ ran\text{-}m\ N \wedge snd\ a\} \cup A = f\ `\ \{a.\ a \in\#\ ran\text{-}m\ N\} \cup A\rangle$
⟨*proof*⟩

lemma *init-clss-l-clause-upd*:
$\langle C \in\#\ dom\text{-}m\ N \Longrightarrow irred\ N\ C \Longrightarrow init\text{-}clss\text{-}l\ (N(C \hookrightarrow C')) = add\text{-}mset\ (C', irred\ N\ C)\ (remove1\text{-}mset\ (N \propto C, irred\ N\ C)\ (init\text{-}clss\text{-}l\ N))\rangle$
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd*:
$\langle C \in\#\ dom\text{-}m\ N \Longrightarrow irred\ N\ C \Longrightarrow init\text{-}clss\text{-}l\ (fmupd\ C\ (C', True)\ N) = add\text{-}mset\ (C', irred\ N\ C)\ (remove1\text{-}mset\ (N \propto C, irred\ N\ C)\ (init\text{-}clss\text{-}l\ N))\rangle$
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd*:
$\langle C \in\#\ dom\text{-}m\ N \Longrightarrow \neg irred\ N\ C \Longrightarrow learned\text{-}clss\text{-}l\ (fmupd\ C\ (C', False)\ N) = add\text{-}mset\ (C', irred\ N\ C)\ (remove1\text{-}mset\ (N \propto C, irred\ N\ C)\ (learned\text{-}clss\text{-}l\ N))\rangle$
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-irrel*: $\langle C \in\#\ dom\text{-}m\ N \Longrightarrow \neg irred\ N\ C \Longrightarrow init\text{-}clss\text{-}l\ (fmupd\ C\ (C', False)\ N) = init\text{-}clss\text{-}l\ N\rangle$
⟨*proof*⟩

lemma *init-clss-l-mapsto-upd-irrel-notin*: $\langle C \notin\#\ dom\text{-}m\ N \Longrightarrow init\text{-}clss\text{-}l\ (fmupd\ C\ (C', False)\ N) = init\text{-}clss\text{-}l\ N\rangle$
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd-irrel*: $\langle C \in\#\ dom\text{-}m\ N \Longrightarrow irred\ N\ C \Longrightarrow learned\text{-}clss\text{-}l\ (fmupd\ C\ (C', True)\ N) = learned\text{-}clss\text{-}l\ N\rangle$
⟨*proof*⟩

lemma *learned-clss-l-mapsto-upd-notin*: $\langle C \notin\#\ dom\text{-}m\ N \Longrightarrow learned\text{-}clss\text{-}l\ (fmupd\ C\ (C', False)\ N) = add\text{-}mset\ (C', False)\ (learned\text{-}clss\text{-}l\ N)\rangle$

⟨*proof*⟩

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

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

lemma *learned-clss-l-mapsto-upd-notin-irrelev*: ‹$C \notin\#$ *dom-m* $N \Longrightarrow$
learned-clss-l (*fmupd* C (C', *True*) N) = *learned-clss-l* N›
⟨*proof*⟩

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

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

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)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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

lemma *learned-clss-l-fmupd-if*:
‹$C' \notin\#$ *dom-m new* $\Longrightarrow$ *learned-clss-l* (*fmupd* C' D *new*) = (*if* ¬*snd* D *then add-mset* D (*learned-clss-l new*) *else learned-clss-l new*)›
⟨*proof*⟩

lemma *unit-propagation-inner-loop-body-l*:
fixes i C :: *nat* **and** S :: ‹*'v twl-st-l*› **and** S' :: ‹*'v twl-st*› **and** L :: ‹*'v literal*›
defines
C'[*simp*]: ‹$C' \equiv$ *get-clauses-l* $S \propto C$›
assumes
SS': ‹(S, S') ∈ *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

$(set\text{-}clauses\text{-}to\text{-}update\text{-}l\ (clauses\text{-}to\text{-}update\text{-}l\ S - \{\#C\#\})\ S) \leq$
$\Downarrow \{(S, S''). (S, S'') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}list\text{-}invs\ S \wedge twl\text{-}stgy\text{-}invs\ S'' \wedge$
$twl\text{-}struct\text{-}invs\ S''\}$
$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\ L\ (twl\text{-}clause\text{-}of\ C')$
$(set\text{-}clauses\text{-}to\text{-}update\ (clauses\text{-}to\text{-}update\ (S') - \{\#(L, twl\text{-}clause\text{-}of\ C')\#\})\ S'))$›
(**is** ‹$?A \leq \Downarrow - ?B$›)
⟨*proof*⟩

lemma *unit-propagation-inner-loop-body-l2*:
assumes
SS': ‹$(S, S') \in twl\text{-}st\text{-}l\ (Some\ L)$› **and**
WS: ‹$C \in\#\ clauses\text{-}to\text{-}update\text{-}l\ S$› **and**
struct-invs: ‹$twl\text{-}struct\text{-}invs\ S'$› **and**
add-inv: ‹$twl\text{-}list\text{-}invs\ S$› **and**
stgy-inv: ‹$twl\text{-}stgy\text{-}invs\ S'$›
shows
‹$(unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l\ L\ C$
$(set\text{-}clauses\text{-}to\text{-}update\text{-}l\ (clauses\text{-}to\text{-}update\text{-}l\ S - \{\#C\#\})\ S),$
$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\ L\ (twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C))$
$(set\text{-}clauses\text{-}to\text{-}update$
$(remove1\text{-}mset\ (L, twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C))$
$(clauses\text{-}to\text{-}update\ S'))\ S'))$
$\in \langle\{(S, S'). (S, S') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}list\text{-}invs\ S \wedge twl\text{-}stgy\text{-}invs\ S' \wedge$
$twl\text{-}struct\text{-}invs\ S'\}\rangle nres\text{-}rel$›
⟨*proof*⟩

This a work around equality: it allows to instantiate variables that appear in goals by hand in a reasonable way (*rule\-tac I=x in EQI*).

definition *EQ* **where**
[*simp*]: ‹$EQ = (=)$›

lemma *EQI*: $EQ\ I\ I$
⟨*proof*⟩

lemma *unit-propagation-inner-loop-body-l-unit-propagation-inner-loop-body*:
‹$EQ\ L''\ L'' \Longrightarrow$
$(uncurry2\ unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body\text{-}l, uncurry2\ unit\text{-}propagation\text{-}inner\text{-}loop\text{-}body) \in$
$\{(((L, C), S0), ((L', C'), S0')). \exists S\ S'. L = L' \wedge C' = (twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C)) \wedge$
$S0 = (set\text{-}clauses\text{-}to\text{-}update\text{-}l\ (clauses\text{-}to\text{-}update\text{-}l\ S - \{\#C\#\})\ S) \wedge$
$S0' = (set\text{-}clauses\text{-}to\text{-}update$
$(remove1\text{-}mset\ (L, twl\text{-}clause\text{-}of\ (get\text{-}clauses\text{-}l\ S \propto C))$
$(clauses\text{-}to\text{-}update\ S'))\ S') \wedge$
$(S, S') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge L = L'' \wedge$
$C \in\#\ clauses\text{-}to\text{-}update\text{-}l\ S \wedge twl\text{-}struct\text{-}invs\ S' \wedge twl\text{-}list\text{-}invs\ S \wedge twl\text{-}stgy\text{-}invs\ S'\} \rightarrow_f$
$\langle\{(S, S'). (S, S') \in twl\text{-}st\text{-}l\ (Some\ L'') \wedge twl\text{-}list\text{-}invs\ S \wedge twl\text{-}stgy\text{-}invs\ S' \wedge$
$twl\text{-}struct\text{-}invs\ S'\}\rangle nres\text{-}rel$›
⟨*proof*⟩

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

definition *unit-propagation-inner-loop-l-inv* **where**
‹$unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L = (\lambda(S, n).$
$(\exists S'. (S, S') \in twl\text{-}st\text{-}l\ (Some\ L) \wedge twl\text{-}struct\text{-}invs\ S' \wedge twl\text{-}stgy\text{-}invs\ S' \wedge$
$twl\text{-}list\text{-}invs\ S \wedge (clauses\text{-}to\text{-}update\ S' \neq \{\#\} \vee n > 0 \longrightarrow get\text{-}conflict\ S' = None) \wedge$

$-L \in$ *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* = (λ(*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*(λ*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* $\Rightarrow$ $'v$ *twl-st-l* $\Rightarrow$ $'v$ *twl-st-l nres*› **where**
‹*unit-propagation-inner-loop-l L* S_0 = *do* {
n $\leftarrow$ *SPEC*(λ-::*nat*. *True*);
(*S*, *n*) $\leftarrow$ $WHILE_T{}^{unit\text{-}propagation\text{-}inner\text{-}loop\text{-}l\text{-}inv\ L}$
(λ(*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*⟩

lemma *refine-add-inv*:
fixes *f* :: ‹$'a \Rightarrow 'a$ *nres*› **and** f' :: ‹$'b \Rightarrow 'b$ *nres*› **and** *h* :: ‹$'b \Rightarrow 'a$›
assumes
‹(f', *f*) $\in$ {(*S*, S'). S' = *h S* $\wedge$ *R S*} $\rightarrow$ ⟨{(*T*, T'). T' = *h T* $\wedge$ P' *T*}⟩ *nres-rel*›
(**is** ‹- $\in$ *?R* $\rightarrow$ ⟨{(*T*, T'). *?H T* T' $\wedge$ P' *T*}⟩ *nres-rel*›)
assumes
‹$\bigwedge$*S*. *R S* $\Longrightarrow$ *f* (*h S*) $\leq$ *SPEC* (λ*T*. *Q T*)›
shows
‹(f', *f*) $\in$ *?R* $\rightarrow$ ⟨{(*T*, T'). *?H T* T' $\wedge$ P' *T* $\wedge$ *Q* (*h T*)}⟩ *nres-rel*›
⟨*proof*⟩

lemma *refine-add-inv-generalised*:
fixes *f* :: ‹$'a \Rightarrow 'b$ *nres*› **and** f' :: ‹$'c \Rightarrow 'd$ *nres*›
assumes
‹(f', *f*) $\in$ *A* $\rightarrow_f$ ⟨*B*⟩ *nres-rel*›
assumes
‹$\bigwedge$*S* S'. (*S*, S') $\in$ *A* $\Longrightarrow$ *f* S' $\leq$ *RES C*›
shows
‹(f', *f*) $\in$ *A* $\rightarrow_f$ ⟨{(*T*, T'). (*T*, T') $\in$ *B* $\wedge$ T' $\in$ *C*}⟩ *nres-rel*›
⟨*proof*⟩

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
‹$(f', f) \in \{(S, S'). S' = h\ S \wedge R\ S\} \to \langle\{(S, S'). (fst\ S' = h'\ (fst\ S) \wedge$
$snd\ S' = h\ (snd\ S)) \wedge P'\ S\}\rangle\ nres\text{-}rel$› (**is** ‹$- \in\ ?R \to \langle\{(S, S').\ ?H\ S\ S' \wedge P'\ S\}\rangle\ nres\text{-}rel$›)
assumes
‹$\bigwedge S.\ R\ S \Longrightarrow f\ (h\ S) \le SPEC\ (\lambda T.\ Q\ (snd\ T))$›
shows
‹$(f', f) \in\ ?R \to \langle\{(S, S').\ ?H\ S\ S' \wedge P'\ S \wedge Q\ (h\ (snd\ S))\}\rangle\ nres\text{-}rel$›
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

definition *unit-propagation-outer-loop-l-inv* **where**
‹$unit\text{-}propagation\text{-}outer\text{-}loop\text{-}l\text{-}inv\ S \longleftrightarrow$

$(\exists S'. (S, S') \in$ *twl-st-l None* $\wedge$ *twl-struct-invs* $S' \wedge$ *twl-stgy-invs* $S' \wedge$
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* $S \neq \{\#\}$)
(λS. *do* {
ASSERT(*literals-to-update-l* $S \neq \{\#\}$);
$(S', L) \leftarrow$ *select-and-remove-from-literals-to-update* S;
unit-propagation-inner-loop-l L S'
})
(S_0 :: 'v twl-st-l)
›

lemma *watched-twl-clause-of-watched*: ‹*watched* (*twl-clause-of* x) = *mset* (*watched-l* x)›
⟨*proof*⟩

lemma *twl-st-of-clause-to-update*:
assumes
TT': ‹$(T, T') \in$ *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' ‘# {#$C \in\#$ *get-clauses* T'. $L' \in\#$ *watched* C#})
(*set-literals-to-update* (*remove1-mset* L' (*literals-to-update* T'))
T'))
$\in$ *twl-st-l* (*Some* L')›
⟨*proof*⟩

lemma *twl-list-invs-set-clauses-to-update-iff*:
assumes ‹*twl-list-invs* T›
shows ‹*twl-list-invs* (*set-clauses-to-update-l* WS (*set-literals-to-update-l* Q T)) $\longleftrightarrow$
(($\forall x \in\# WS$. *case* x *of* $C \Rightarrow C \in\#$ *dom-m* (*get-clauses-l* T)) $\wedge$
distinct-mset WS)›
⟨*proof*⟩

lemma *unit-propagation-outer-loop-l-spec*:
‹(*unit-propagation-outer-loop-l*, *unit-propagation-outer-loop*) $\in$
$\{(S, S'). (S, S') \in$ *twl-st-l None* $\wedge$ *twl-struct-invs* $S' \wedge$
twl-stgy-invs $S' \wedge$ *twl-list-invs* $S \wedge$ *clauses-to-update-l* $S = \{\#\} \wedge$
get-conflict-l S = *None*$\} \rightarrow_f$
$\langle\{(T, T'). (T, T') \in$ *twl-st-l None* $\wedge$
(*twl-list-invs* $T \wedge$ *twl-struct-invs* $T' \wedge$ *twl-stgy-invs* $T' \wedge$
clauses-to-update-l $T = \{\#\}) \wedge$
literals-to-update $T' = \{\#\} \wedge$ *clauses-to-update* $T' = \{\#\} \wedge$
no-step cdcl-twl-cp $T'\}\rangle$ *nres-rel*›
(**is** ‹- $\in$?R $\rightarrow_f$?I› **is** ‹- $\in$ - $\rightarrow_f$ ⟨?B⟩ *nres-rel*›)
⟨*proof*⟩

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

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

fun *lit-and-ann-of-propagated* **where**
‹*lit-and-ann-of-propagated* (*Propagated L C*) = (*L*, *C*)› |
‹*lit-and-ann-of-propagated* (*Decided* -) = *undefined*›
— we should never call the function in that context

definition *tl-state-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l*› **where**
‹*tl-state-l* = (λ(*M*, *N*, *D*, *NE*, *UE*, *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 S0 =
  do {
    ASSERT(get-conflict-l S0 ≠ None);
    (-, S) ←
      WHILE_T^λ(brk, S). skip-and-resolve-loop-inv-l S0 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 ∉# 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, S0);
    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 = {#}}›
⟨*proof*⟩ **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 ≠ []› **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*⟩

lemma *get-level-same-lits-cong*:
assumes
‹map (atm-of o lit-of) M = map (atm-of o lit-of) M′› **and**
‹map is-decided M = map is-decided M′›
shows ‹get-level M L = get-level M′ L›
⟨*proof*⟩

lemma *clauses-in-unit-clss-have-level0*:
assumes
struct-invs: ‹twl-struct-invs T› **and**
C: ‹C ∈# unit-clss T› **and**
LC-T: ‹Propagated L C ∈ set (get-trail T)› **and**
count-dec: ‹0 < count-decided (get-trail T)›
shows
‹get-level (get-trail T) L = 0› (**is** *?lev-L*) **and**
‹∀ K∈# C. get-level (get-trail T) K = 0› (**is** *?lev-K*)
⟨*proof*⟩

lemma *clauses-clss-have-level1-notin-unit*:
assumes
struct-invs: ‹twl-struct-invs T› **and**
LC-T: ‹Propagated L C ∈ set (get-trail T)› **and**
count-dec: ‹0 < count-decided (get-trail T)› **and**
‹get-level (get-trail T) L > 0›
shows

‹$C \notin\#$ *unit-clss* T›
⟨*proof*⟩

lemma *skip-and-resolve-loop-l-spec*:
‹(*skip-and-resolve-loop-l*, *skip-and-resolve-loop*) ∈
{(*S*::′*v twl-st-l*, *S*′). (*S*, *S*′) ∈ *twl-st-l None* ∧ *twl-struct-invs S*′ ∧
twl-stgy-invs S′ ∧
twl-list-invs S ∧ *clauses-to-update-l S* = {#} ∧ *literals-to-update-l S* = {#} ∧
get-conflict S′ ≠ *None* ∧
0 < *count-decided* (*get-trail-l S*)} $\rightarrow_f$
⟨{(*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*⟩

end

definition *find-decomp* :: ‹′*v literal* ⇒ ′*v twl-st-l* ⇒ ′*v twl-st-l nres*› **where**
‹*find-decomp* = (λ*L* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*).
SPEC(λ*S*. ∃ *K M2 M1*. *S* = (*M1*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) ∧
(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
get-level M K = *get-maximum-level M* (*the D* − {#−*L*#}) + *1*))›

lemma *find-decomp-alt-def*:
‹*find-decomp L S* =
SPEC(λ*T*. ∃ *K M2 M1*. *equality-except-trail S T* ∧ *get-trail-l T* = *M1* ∧
(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*get-trail-l S*)) ∧
get-level (*get-trail-l S*) *K* =
get-maximum-level (*get-trail-l S*) (*the* (*get-conflict-l S*) − {#−*L*#}) + *1*)›
⟨*proof*⟩

definition *find-lit-of-max-level* :: ‹′*v twl-st-l* ⇒ ′*v literal* ⇒ ′*v literal nres*› **where**
‹*find-lit-of-max-level* = (λ(*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) *L*.
SPEC(λ*L*′. *L*′ ∈# *the D* − {#−*L*#} ∧ *get-level M L*′ = *get-maximum-level M* (*the D* − {#−*L*#})))›

definition *ex-decomp-of-max-lvl* :: ‹(′*v*, *nat*) *ann-lits* ⇒ ′*v cconflict* ⇒ ′*v literal* ⇒ *bool*› **where**
‹*ex-decomp-of-max-lvl M D L* ⟷
(∃ *K M1 M2*. (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*) ∧
get-level M K = *get-maximum-level M* (*remove1-mset* (−*L*) (*the D*)) + *1*)›

fun *add-mset-list* :: ‹′*a list* ⇒ ′*a multiset multiset* ⇒ ′*a multiset multiset*› **where**
‹*add-mset-list L UE* = *add-mset* (*mset L*) *UE*›

definition (**in** −)*list-of-mset* :: ‹′*v clause* ⇒ ′*v clause-l nres*› **where**
‹*list-of-mset D* = *SPEC*(λ*D*′. *D* = *mset D*′)›

fun *extract-shorter-conflict-l* :: ‹′*v twl-st-l* ⇒ ′*v twl-st-l nres*›
where
‹*extract-shorter-conflict-l* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *SPEC*(λ*S*.
∃ *D*′. *D*′ ⊆# *the D* ∧ *S* = (*M*, *N*, *Some D*′, *NE*, *UE*, *WS*, *Q*) ∧
clause ‘# *twl-clause-of* ‘# *ran-mf N* + *NE* + *UE* ⊨*pm D*′ ∧ −(*lit-of* (*hd M*)) ∈# *D*′)›

declare *extract-shorter-conflict-l.simps*[*simp del*]
lemmas *extract-shorter-conflict-l-def* = *extract-shorter-conflict-l.simps*

lemma *extract-shorter-conflict-l-alt-def*:
‹*extract-shorter-conflict-l S* = *SPEC*(*λT*.
$\exists D'$. $D' \subseteq\#$ *the* (*get-conflict-l S*) ∧ *equality-except-conflict-l S T* ∧
get-conflict-l T = *Some* D' ∧
clause '# *twl-clause-of* '# *ran-mf* (*get-clauses-l S*) + *get-unit-clauses-l S* ⊨*pm* D' ∧
−*lit-of* (*hd* (*get-trail-l S*)) ∈# D')›
⟨*proof*⟩

definition *backtrack-l-inv* **where**
‹*backtrack-l-inv S* ⟷
(∃ S'. (*S*, S') ∈ *twl-st-l None* ∧
get-trail-l S ≠ [] ∧
no-step $cdcl_W$*-restart-mset.skip* ($state_W$*-of* S')∧
no-step $cdcl_W$*-restart-mset.resolve* ($state_W$*-of* S') ∧
get-conflict-l S ≠ *None* ∧
twl-struct-invs S' ∧
twl-stgy-invs S' ∧
twl-list-invs S ∧
get-conflict-l S ≠ *Some* {#})
›

definition *get-fresh-index* :: ‹'*v clauses-l* ⇒ *nat nres*› **where**
‹*get-fresh-index N* = *SPEC*(*λi*. *i* > *0* ∧ *i* ∉# *dom-m N*)›

definition *propagate-bt-l* :: ‹'*v literal* ⇒ '*v literal* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*propagate-bt-l* = (*λL* 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 ≠ {#}}⟩ *nres-rel*›
  (**is** ‹ - ∈ *?R* $\rightarrow_f$ *?I*›)
⟨*proof*⟩

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*′#}))›

definition *decide-l-or-skip* :: ‹′*v twl-st-l* ⇒ (*bool* × ′*v twl-st-l*) *nres*› **where**
  ‹*decide-l-or-skip S* = (*do* {
    ASSERT(*decide-l-or-skip-pre S*);
    L ← *find-unassigned-lit-l S*;
    case L of
      None ⇒ *RETURN* (*True*, *S*)
    | *Some L* ⇒ *RETURN* (*False*, *decide-lit-l L S*)
  })
›
method *match-⇓* =
  (*match* **conclusion in** ‹*f* ≤ ⇓ *R g*› **for** *f* :: ‹′*a nres*› **and** *R* :: ‹(′*a* × ′*b*) *set*› **and**
    g :: ‹′*b nres*› ⇒
    ‹*match premises in*
      I[*thin*,*uncurry*]: ‹*f* ≤ ⇓ *R*′ *g*› *for R*′ :: ‹(′*a* × ′*b*) *set*›

⇒ ‹rule refinement-trans-long[of f f g g R' R, OF refl refl - I]›
| I[thin,uncurry]: ‹- ⟹ f ≤ ⇓ R' g› for R' :: ‹('a × 'b) set›
⇒ ‹rule refinement-trans-long[of f f g g R' R, OF refl refl - I]›
›)

lemma *decide-l-or-skip-spec*:
‹(decide-l-or-skip, decide-or-skip) ∈
{(S, S'). (S, S') ∈ twl-st-l None ∧ get-conflict-l S = None ∧
clauses-to-update-l S = {#} ∧ literals-to-update-l S = {#} ∧ no-step cdcl-twl-cp S' ∧
twl-struct-invs S' ∧ twl-stgy-invs S' ∧ twl-list-invs S} $\rightarrow_f$
⟨{((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⟩

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

definition *cdcl-twl-o-prog-l-pre* **where**
‹cdcl-twl-o-prog-l-pre S ⟷
(∃ S' . (S, S') ∈ twl-st-l None ∧
twl-struct-invs S' ∧
twl-stgy-invs S' ∧
twl-list-invs S)›

definition *cdcl-twl-o-prog-l* :: ‹'v twl-st-l ⇒ (bool × 'v twl-st-l) nres› **where**
‹cdcl-twl-o-prog-l S =
do {
ASSERT(cdcl-twl-o-prog-l-pre S);
do {
if get-conflict-l S = None
then decide-l-or-skip S
else if count-decided (get-trail-l S) > 0
then do {
T ← skip-and-resolve-loop-l S;
ASSERT(get-conflict-l T ≠ None ∧ get-conflict-l T ≠ Some {#});
U ← backtrack-l T;
RETURN (False, U)
}
else RETURN (True, S)
}
}
›

lemma *twl-st-lE*:
‹(⋀M N D NE UE WS Q. T = (M, N, D, NE, UE, WS, Q) ⟹ P (M, N, D, NE, UE, WS, Q))
⟹ P T›
for T :: ‹'a twl-st-l›
⟨proof⟩

lemma *weaken-⇓'*: ‹$f \le \Downarrow R'\ g \Longrightarrow R' \subseteq R \Longrightarrow f \le \Downarrow R\ g$›
⟨*proof*⟩

lemma *cdcl-twl-o-prog-l-spec*:
‹(*cdcl-twl-o-prog-l*, *cdcl-twl-o-prog*) ∈
{(S, S'). (S, S') ∈ *twl-st-l None* ∧
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* ⟶ *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*⟩

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 ≡ λ(brk, T). ∃ S_0' T'. (T, T') ∈ *twl-st-l None* ∧
(S_0, S_0') ∈ *twl-st-l None* ∧
twl-struct-invs T' ∧
twl-stgy-invs T' ∧
(brk ⟶ *final-twl-state* T') ∧
*cdcl-twl-stgy*** S_0' T' ∧
clauses-to-update-l T = {#} ∧
(¬brk ⟶ *get-conflict-l* T = *None*)›

definition *cdcl-twl-stgy-prog-l* :: ‹'v *twl-st-l* ⇒ 'v *twl-st-l nres*› **where**

```
‹cdcl-twl-stgy-prog-l S0 =
do {
  do {
    (brk, T) ← WHILE_T^(cdcl-twl-stgy-prog-l-inv S0)
      (λ(brk, -). ¬brk)
      (λ(brk, S).
      do {
        T ← unit-propagation-outer-loop-l S;
        cdcl-twl-o-prog-l T
      })
      (False, S0);
    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'} $\rightarrow_f$
⟨{(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*⟩

lemma *refine-pair-to-SPEC*:

fixes f :: ‹$'s \Rightarrow 's$ *nres*› **and** g :: ‹$'b \Rightarrow 'b$ *nres*›
assumes ‹$(f, g) \in \{(S, S'). (S, S') \in H \wedge R\ S\ S'\} \rightarrow_f \langle\{(S, S'). (S, S') \in H' \wedge P'\ S\}\rangle$*nres-rel*›
(**is** ‹- $\in$ *?R* $\rightarrow_f$ *?I*›)
assumes ‹$R\ S\ S'$› **and** [*simp*]: ‹$(S, S') \in H$›
shows ‹$f\ S \leq \Downarrow \{(S, S'). (S, S') \in H' \wedge P'\ S\}\ (g\ S')$›
⟨*proof*⟩

definition *cdcl-twl-stgy-prog-l-pre* **where**
‹*cdcl-twl-stgy-prog-l-pre* $S\ S' \longleftrightarrow$
$((S, S') \in$ *twl-st-l None* $\wedge$ *twl-struct-invs* $S' \wedge$ *twl-stgy-invs* $S' \wedge$
clauses-to-update-l $S = \{\#\} \wedge$ *get-conflict-l* $S =$ *None* $\wedge$ *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* $S \leq \Downarrow$ (*twl-st-l None*) (*conclusive-TWL-run* S')›
⟨*proof*⟩

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

definition *cdcl-twl-stgy-prog-break-l* :: ‹$'v$ *twl-st-l* $\Rightarrow 'v$ *twl-st-l nres*› **where**
‹*cdcl-twl-stgy-prog-break-l* S_0 =
do {
$b \leftarrow$ *SPEC*(λ-. *True*);
$(b, brk, T) \leftarrow WHILE_T{}^{\lambda(b,\ S).\ cdcl\text{-}twl\text{-}stgy\text{-}prog\text{-}l\text{-}inv\ S_0\ S}$
($\lambda(b, brk, -)$. $b \wedge \neg brk$)
($\lambda(-, brk, S)$. *do* {
$T \leftarrow$ *unit-propagation-outer-loop-l* S;
$T \leftarrow$ *cdcl-twl-o-prog-l* T;
$b \leftarrow$ *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*) $\in$
$\{(S, S'). (S, S') \in$ *twl-st-l None* $\wedge$ *twl-list-invs* $S \wedge$
clauses-to-update-l $S = \{\#\} \wedge$
twl-struct-invs $S' \wedge$ *twl-stgy-invs* $S'\} \rightarrow_f$
$\langle\{(T, T'). (T, T') \in \{(T, T'). (T, T') \in$ *twl-st-l None* $\wedge$ *twl-list-invs* $T \wedge$
twl-struct-invs $T' \wedge$ *twl-stgy-invs* $T'\} \wedge$ *True*$\}\rangle$ *nres-rel*›
(**is** ‹ - $\in$ *?R* $\rightarrow_f$ *?I*› **is** ‹ - $\in$ *?R* $\rightarrow_f$ ⟨*?J*⟩*nres-rel*›)
⟨*proof*⟩

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'*)⟩
⟨*proof*⟩

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'*)⟩
⟨*proof*⟩

lemma *trail-changes-same-decomp*:
assumes
M'-lit: ⟨*map lit-of M'* = *map lit-of ysa* @ *L* # *map lit-of zsa*⟩ **and**
M'-dec: ⟨*map is-decided M'* = *map is-decided ysa* @ *False* # *map is-decided zsa*⟩
obtains *C' M2 M1* **where** ⟨*M'* = *M2* @ *Propagated L C'* # *M1*⟩ **and**
⟨*map lit-of M2* = *map lit-of ysa*⟩**and**
⟨*map is-decided M2* = *map is-decided ysa*⟩ **and**
⟨*map lit-of M1* = *map lit-of zsa*⟩ **and**
⟨*map is-decided M1* = *map is-decided zsa*⟩
⟨*proof*⟩

lemma
assumes
⟨*map lit-of M* = *map lit-of M'*⟩ **and**
⟨*map is-decided M* = *map is-decided M'*⟩
shows
trail-renumber-count-dec:
⟨*count-decided M* = *count-decided M'*⟩ **and**
trail-renumber-get-level:
⟨*get-level M L* = *get-level M' L*⟩

⟨*proof*⟩

lemma *valid-trail-reduction-Propagated-inD*:
  ‹$valid\text{-}trail\text{-}reduction\ M\ M' \Longrightarrow Propagated\ L\ C \in set\ M' \Longrightarrow \exists C'.\ Propagated\ L\ C' \in set\ M$›
  ⟨*proof*⟩

lemma *valid-trail-reduction-Propagated-inD2*:
  ‹$valid\text{-}trail\text{-}reduction\ M\ M' \Longrightarrow length\ M = length\ M' \Longrightarrow Propagated\ L\ C \in set\ M \Longrightarrow$
    $\exists C'.\ Propagated\ L\ C' \in set\ M'$›
  ⟨*proof*⟩

lemma *get-all-ann-decomposition-change-annotation-exists*:
  assumes
    ‹$(Decided\ K\ \#\ M',\ M2') \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M2)$› **and**
    ‹$map\ lit\text{-}of\ M1 = map\ lit\text{-}of\ M2$› **and**
    ‹$map\ is\text{-}decided\ M1 = map\ is\text{-}decided\ M2$›
  shows ‹$\exists M''\ M2'.\ (Decided\ K\ \#\ M'',\ M2') \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ M1) \wedge$
    $map\ lit\text{-}of\ M'' = map\ lit\text{-}of\ M' \wedge\ map\ is\text{-}decided\ M'' = map\ is\text{-}decided\ M'$›
  ⟨*proof*⟩

lemma *valid-trail-reduction-trans*:
  assumes
    M1-M2: ‹$valid\text{-}trail\text{-}reduction\ M1\ M2$› **and**
    M2-M3: ‹$valid\text{-}trail\text{-}reduction\ M2\ M3$›
  shows ‹$valid\text{-}trail\text{-}reduction\ M1\ M3$›
⟨*proof*⟩

lemma *valid-trail-reduction-length-leD*: ‹$valid\text{-}trail\text{-}reduction\ M\ M' \Longrightarrow length\ M' \leq length\ M$›
  ⟨*proof*⟩

lemma *valid-trail-reduction-level0-iff*:
  assumes *valid*: ‹$valid\text{-}trail\text{-}reduction\ M\ M'$› **and** *n-d*: ‹$no\text{-}dup\ M$›
  shows ‹$(L \in lits\text{-}of\text{-}l\ M \wedge get\text{-}level\ M\ L = 0) \longleftrightarrow (L \in lits\text{-}of\text{-}l\ M' \wedge get\text{-}level\ M'\ L = 0)$›
⟨*proof*⟩

lemma *map-lit-of-eq-defined-litD*: ‹$map\ lit\text{-}of\ M = map\ lit\text{-}of\ M' \Longrightarrow defined\text{-}lit\ M = defined\text{-}lit\ M'$›
  ⟨*proof*⟩

lemma *map-lit-of-eq-no-dupD*: ‹$map\ lit\text{-}of\ M = map\ lit\text{-}of\ M' \Longrightarrow no\text{-}dup\ M = no\text{-}dup\ M'$›
  ⟨*proof*⟩

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* :: ‹$'v\ twl\text{-}st\text{-}l \Rightarrow 'v\ twl\text{-}st\text{-}l \Rightarrow bool$› **where**
restart-trail:
  ‹$cdcl\text{-}twl\text{-}restart\text{-}l\ (M,\ N,\ None,\ NE,\ UE,\ \{\#\},\ Q)$
    $(M',\ N',\ None,\ NE + mset\ `\#\ NE',\ UE + mset\ `\#\ UE',\ \{\#\},\ Q')$›
  if
    ‹$valid\text{-}trail\text{-}reduction\ M\ M'$› **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*⟩
⟨*proof*⟩

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

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

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 (λb. 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;

$b2 \leftarrow SPEC\ (\lambda(\text{-} ::bool).\ True);$
$if\ b \wedge b2 \wedge \neg brk\ then\ do\ \{$
$T \leftarrow SPEC(\lambda T.\ cdcl\text{-}twl\text{-}restart\text{-}l\ S\ T);$
$RETURN\ (T,\ n + 1)$
$\}$
$else$
$if\ b \wedge \neg brk\ then\ do\ \{$
$T \leftarrow SPEC(\lambda T.\ cdcl\text{-}twl\text{-}restart\text{-}l\ S\ T);$
$RETURN\ (T,\ n + 1)$
$\}$
$else$
$RETURN\ (S,\ n)$
$\}$›

lemma (**in** −)[*twl-st-l*]:
‹$(S,\ S') \in twl\text{-}st\text{-}l\ None \Longrightarrow get\text{-}learned\text{-}clss\ S' = twl\text{-}clause\text{-}of\ \text{`}\#\ (get\text{-}learned\text{-}clss\text{-}l\ S)$›
⟨*proof*⟩

lemma *restart-required-l-restart-required*:
‹$(uncurry\ restart\text{-}required\text{-}l,\ uncurry\ restart\text{-}required) \in$
$\{(S,\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S\} \times_f nat\text{-}rel \rightarrow_f$
$\langle bool\text{-}rel\rangle\ nres\text{-}rel$›
⟨*proof*⟩

lemma *restart-abs-l-restart-prog*:
‹$(uncurry2\ restart\text{-}abs\text{-}l,\ uncurry2\ restart\text{-}prog) \in$
$\{(S,\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S \wedge clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}\}$
$\times_f nat\text{-}rel\ \times_f bool\text{-}rel \rightarrow_f$
$\langle\{(S,\ S').\ (S,\ S') \in twl\text{-}st\text{-}l\ None \wedge twl\text{-}list\text{-}invs\ S \wedge clauses\text{-}to\text{-}update\text{-}l\ S = \{\#\}\}$
$\times_f nat\text{-}rel\rangle\ nres\text{-}rel$›
⟨*proof*⟩

definition *cdcl-twl-stgy-restart-abs-l-inv* **where**
‹$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n \equiv$
$(\exists\, S_0'\ T'.$
$(S_0,\ S_0') \in twl\text{-}st\text{-}l\ None \wedge$
$(T,\ T') \in twl\text{-}st\text{-}l\ None \wedge$
$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}prog\text{-}inv\ S_0'\ brk\ T'\ n \wedge$
$clauses\text{-}to\text{-}update\text{-}l\ T\ = \{\#\} \wedge$
$twl\text{-}list\text{-}invs\ T)$›

definition *cdcl-twl-stgy-restart-abs-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**
‹$cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\ S_0 =$
$do\ \{$
$(brk,\ T,\ \text{-}) \leftarrow WHILE_T{}^{\lambda(brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}l\text{-}inv\ S_0\ brk\ T\ n}$
$(\lambda(brk,\ \text{-}).\ \neg brk)$
$(\lambda(brk,\ S,\ n).$
$do\ \{$
$T \leftarrow unit\text{-}propagation\text{-}outer\text{-}loop\text{-}l\ S;$
$(brk,\ T) \leftarrow cdcl\text{-}twl\text{-}o\text{-}prog\text{-}l\ T;$
$(T,\ n) \leftarrow restart\text{-}abs\text{-}l\ T\ n\ brk;$
$RETURN\ (brk,\ T,\ n)$
$\})$
$(False,\ S_0,\ 0);$

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 = {#}} $\to_f$
⟨{(*S*, *S'*). (*S*, *S'*) ∈ *twl-st-l None* ∧ *twl-list-invs S*}⟩ *nres-rel*›
⟨*proof*⟩

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* ∈ *set A* ⟹ *mset A* = *B* ⟹ *i* ∈# *B*›
⟨*proof*⟩

Our transformation will be chains of a weaker version of restarts, that don't update the watch lists and only keep partial correctness of it.

lemma *cdcl-twl-restart-l-cdcl-twl-restart-l-is-cdcl-twl-restart-l*:
assumes
ST: ‹*cdcl-twl-restart-l S T*› **and** *TU*: ‹*cdcl-twl-restart-l T U*› **and**
n-d: ‹*no-dup* (*get-trail-l S*)›
shows ‹*cdcl-twl-restart-l S U*›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-restart-l-no-dup*:
assumes
ST: ‹$cdcl\text{-}twl\text{-}restart\text{-}l^{**}$ *S T*› **and**
n-d: ‹*no-dup* (*get-trail-l S*)›
shows ‹*no-dup* (*get-trail-l T*)›
⟨*proof*⟩

lemma *tranclp-cdcl-twl-restart-l-cdcl-is-cdcl-twl-restart-l*:
assumes
ST: ‹$cdcl\text{-}twl\text{-}restart\text{-}l^{++}$ *S T*› **and**
n-d: ‹*no-dup* (*get-trail-l S*)›
shows ‹*cdcl-twl-restart-l S T*›
⟨*proof*⟩

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

Auxilary definition This definition states that the domain of the clauses is reduced, but the remaining clauses are not changed.

definition *reduce-dom-clauses* **where**
‹*reduce-dom-clauses N N′* ⟷
(∀ *C*. *C* ∈# *dom-m N′* ⟶ *C* ∈# *dom-m N* ∧ *fmlookup N C* = *fmlookup N′ C*)›

lemma *reduce-dom-clauses-fdrop*[*simp*]: ‹*reduce-dom-clauses N* (*fmdrop C N*)›
⟨*proof*⟩

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

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

definition *valid-trail-reduction-eq* **where**
‹*valid-trail-reduction-eq M M′* ⟷ *valid-trail-reduction M M′* ∧ *length M* = *length M′*›

lemma *valid-trail-reduction-eq-alt-def*:
‹*valid-trail-reduction-eq M M′* ⟷ *map lit-of M* = *map lit-of M′* ∧
map is-decided M = *map is-decided M′*›
⟨*proof*⟩

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

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

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

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

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

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

definition *no-dup-reasons-invs-wl* **where**

⟨*no-dup-reasons-invs-wl S* ⟷
  (*distinct-mset* (*mark-of* '# *filter-mset* (λ*C*. *is-proped C* ∧ *mark-of C* > *0*) (*mset* (*get-trail-l S*))))⟩

inductive *different-annot-all-killed* **where**
propa-changed:
  ⟨*different-annot-all-killed N NUE* (*Propagated L C*) (*Propagated L C*′)⟩
    if ⟨*C* ≠ *C*′⟩ **and** ⟨*C*′ = *0*⟩ **and**
      ⟨*C* ∈# *dom-m N* ⟹ *mset* (*N*∝*C*) ∈# *NUE*⟩ |
propa-not-changed:
  ⟨*different-annot-all-killed N NUE* (*Propagated L C*) (*Propagated L C*)⟩ |
decided-not-changed:
  ⟨*different-annot-all-killed N NUE* (*Decided L*) (*Decided L*)⟩

lemma *different-annot-all-killed-refl*[*iff*]:
  ⟨*different-annot-all-killed N NUE z z* ⟷ *is-proped z* ∨ *is-decided z*⟩
  ⟨*proof*⟩

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

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

Refinement towards code Once of the first thing we do, is removing clauses we know to be true. We do in two ways:

- along the trail (at level 0); this makes sure that annotations are kept;
- then along the watch list.

This is (obviously) not complete but is faster by avoiding iterating over all clauses. Here are the rules we want to apply for our very limited inprocessing:

inductive *remove-one-annot-true-clause* :: ⟨′*v twl-st-l* ⇒ ′*v twl-st-l* ⇒ *bool*⟩ **where**
remove-irred-trail:
  ⟨*remove-one-annot-true-clause* (*M* @ *Propagated L C* # *M*′, *N*, *D*, *NE*, *UE*, *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* ∈# *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*∝*C*)) *UE*, *W*, *Q*)⟩
if
  ⟨*get-level* (*M* @ *Propagated L C* # *M*′) *L* = *0*⟩ **and**
  ⟨*C* > *0*⟩ **and**
  ⟨*C* ∈# *dom-m N*⟩ **and**
  ⟨¬*irred N C*⟩ |
remove-irred:
  ⟨*remove-one-annot-true-clause* (*M*, *N*, *D*, *NE*, *UE*, *W*, *Q*)
    (*M*, *fmdrop C N*, *D*, *add-mset* (*mset* (*N*∝*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*: ⟨$(\exists NE'. (\exists b. NE' = \{\#b\#\} \wedge P\ b\ NE') \wedge Q\ NE') \longleftrightarrow$
$(\exists b. P\ b\ \{\#b\#\} \wedge Q\ \{\#b\#\})$⟩
⟨*proof*⟩

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'⟩
⟨*proof*⟩

lemma (**in** *conflict-driven-clause-learning*$_W$) *trail-no-annotation-reuse*:
assumes
struct-invs: ⟨*cdcl*$_W$*-all-struct-inv* S⟩ **and**
LC: ⟨*Propagated* L $C \in$ *set* (*trail* S)⟩ **and**
LC': ⟨*Propagated* L' $C \in$ *set* (*trail* S)⟩
shows $L = L'$
⟨*proof*⟩

lemma *remove-one-annot-true-clause-cdcl-twl-restart-l*:
assumes
rem: ⟨*remove-one-annot-true-clause* S T⟩ **and**
lst-invs: ⟨*twl-list-invs* S⟩ **and**
SS': ⟨$(S, S') \in$ *twl-st-l None*⟩ **and**
struct-invs: ⟨*twl-struct-invs* S'⟩ **and**
confl: ⟨*get-conflict-l* S = *None*⟩ **and**
upd: ⟨*clauses-to-update-l* $S = \{\#\}$⟩ **and**
n-d: ⟨*no-dup* (*get-trail-l* S)⟩
shows ⟨*cdcl-twl-restart-l* S T⟩
⟨*proof*⟩

lemma *is-annot-iff-annotates-first*:
assumes
ST: ⟨$(S, T) \in$ *twl-st-l None*⟩ **and**
list-invs: ⟨*twl-list-invs* S⟩ **and**
struct-invs: ⟨*twl-struct-invs* T⟩ **and**

C0: ‹*C* > *0*›
shows
‹(∃ *L*. *Propagated L C* ∈ *set* (*get-trail-l S*)) ⟷
((*length* (*get-clauses-l S* ∝ *C*) > *2* ⟶
Propagated (*get-clauses-l S* ∝ *C* ! *0*) *C* ∈ *set* (*get-trail-l S*)) ∧
((*length* (*get-clauses-l S* ∝ *C*) ≤ *2* ⟶
Propagated (*get-clauses-l S* ∝ *C* ! *0*) *C* ∈ *set* (*get-trail-l S*) ∨
Propagated (*get-clauses-l S* ∝ *C* ! *1*) *C* ∈ *set* (*get-trail-l S*))))›
(**is** ‹*?A* ⟷ *?B*›)
⟨*proof*⟩

lemma *trail-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
LaC: ‹*Propagated L C* ∈ *set* (*get-trail-l S*)› **and**
C0: ‹*C* > *0*›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
⟨*proof*⟩

lemma *is-annot-no-other-true-lit*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
C0: ‹*C* > *0*› **and**
LaC: ‹*Propagated La C* ∈ *set* (*get-trail-l S*)› **and**
LC: ‹*L* ∈ *set* (*get-clauses-l S* ∝ *C*)› **and**
L: ‹*L* ∈ *lits-of-l* (*get-trail-l S*)›
shows
‹*La* = *L*› **and**
‹*length* (*get-clauses-l S* ∝ *C*) > *2* ⟹ *L* = *get-clauses-l S* ∝ *C* ! *0*›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-cdcl-twl-restart-l2*:
assumes
rem: ‹*remove-one-annot-true-clause S T*› **and**
lst-invs: ‹*twl-list-invs S*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
n-d: ‹(*S*, *T'*) ∈ *twl-st-l None*› ‹*twl-struct-invs T'*›
shows ‹*cdcl-twl-restart-l S T*›
⟨*proof*⟩

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

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

lemma *remove-one-annot-true-clause-clauses-to-update-l*:
‹*remove-one-annot-true-clause S T* ⟹ *clauses-to-update-l T* = *clauses-to-update-l S*›

⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-clauses-to-update-l*:
‹*remove-one-annot-true-clause*** *S T* ⟹ *clauses-to-update-l T* = *clauses-to-update-l S*›
⟨*proof*⟩

lemma *cdcl-twl-restart-l-invs*:
assumes *ST*: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and** ‹*cdcl-twl-restart-l S S'*›
shows ‹∃ *T'*. (*S'*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs S'* ∧
clauses-to-update-l S' = {#} ∧ *cdcl-twl-restart T T'* ∧ *twl-struct-invs T'*›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-restart-l-invs*:
assumes
‹*cdcl-twl-restart-l*** *S S'*› **and**
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
‹*clauses-to-update-l S* = {#}›
shows ‹∃ *T'*. (*S'*, *T'*) ∈ *twl-st-l None* ∧ *twl-list-invs S'* ∧
clauses-to-update-l S' = {#} ∧ *cdcl-twl-restart*** *T T'* ∧ *twl-struct-invs T'*›
⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-cdcl-twl-restart-l2*:
assumes
rem: ‹*remove-one-annot-true-clause*** *S T*› **and**
lst-invs: ‹*twl-list-invs S*› **and**
confl: ‹*get-conflict-l S* = *None*› **and**
upd: ‹*clauses-to-update-l S* = {#}› **and**
n-d: ‹(*S*, *S'*) ∈ *twl-st-l None*› ‹*twl-struct-invs S'*›
shows ‹∃ *T'*. *cdcl-twl-restart-l*** *S T* ∧ (*T*, *T'*) ∈ *twl-st-l None* ∧ *cdcl-twl-restart*** *S' T'* ∧
twl-struct-invs T'›
⟨*proof*⟩

definition *drop-clause-add-move-init* **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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

definition *remove-all-annot-true-clause-one-imp*
where

‹remove-all-annot-true-clause-one-imp = (λ(C, S). do {
    if C ∈# dom-m (get-clauses-l S) then
      if irred (get-clauses-l S) C
      then RETURN (drop-clause-add-move-init S C)
      else RETURN (drop-clause S C)
    else do {
      RETURN S
    }
  })›

definition *remove-one-annot-true-clause-imp-inv* **where**
‹remove-one-annot-true-clause-imp-inv S =
  (λ(i, T). remove-one-annot-true-clause** S T ∧ twl-list-invs S ∧ i ≤ length (get-trail-l S) ∧
    twl-list-invs S ∧
    clauses-to-update-l S = clauses-to-update-l T ∧
    literals-to-update-l S = literals-to-update-l T ∧
    get-conflict-l T = None ∧
    (∃ S'. (S, S') ∈ twl-st-l None ∧ twl-struct-invs S') ∧
    get-conflict-l S = None ∧ clauses-to-update-l S = {#} ∧
    length (get-trail-l S) = length (get-trail-l T) ∧
    (∀ j<i. is-proped (rev (get-trail-l T) ! j) ∧ mark-of (rev (get-trail-l T) ! j) = 0))›

definition *remove-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 ∧ twl-list-invs S ∧ i ≤ length xs ∧
      twl-list-invs S ∧ get-trail-l S = get-trail-l T ∧
      (∃ S'. (S, S') ∈ twl-st-l None ∧ twl-struct-invs S') ∧
      get-conflict-l S = None ∧ clauses-to-update-l S = {#})›

definition *remove-all-annot-true-clause-imp-pre* **where**
‹remove-all-annot-true-clause-imp-pre L S ⟷
  (twl-list-invs S ∧ twl-list-invs S ∧
  (∃ S'. (S, S') ∈ twl-st-l None ∧ twl-struct-invs S') ∧
  get-conflict-l S = None ∧ clauses-to-update-l S = {#} ∧ L ∈ lits-of-l (get-trail-l S))›

definition *remove-all-annot-true-clause-imp*
  :: ‹'v literal ⇒ 'v twl-st-l ⇒ ('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 ← SPEC(λxs.
       (∀ x∈set xs. x ∈# dom-m (get-clauses-l S) ⟶ L ∈ set ((get-clauses-l S)∝x)));
    (-, T) ← $WHILE_T$$^{λ(i, T). remove-all-annot-true-clause-imp-inv S xs (i, T)}$
      (λ(i, T). i < length xs)
      (λ(i, T). do {
        ASSERT(i < length xs);
        if xs!i ∈# dom-m (get-clauses-l T) ∧ length ((get-clauses-l T) ∝ (xs!i)) ≠ 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$_T^{\textit{remove-one-annot-true-clause-imp-inv } S}$
(λ(i, S). i < k)
(λ(i, S). remove-one-annot-true-clause-one-imp i S)
(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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-reduce-dom-clauses*:
‹*remove-one-annot-true-clause S U* ⟹
reduce-dom-clauses (*get-clauses-l S*) (*get-clauses-l U*)›
⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-reduce-dom-clauses*:
‹*remove-one-annot-true-clause*** *S U* ⟹
reduce-dom-clauses (*get-clauses-l S*) (*get-clauses-l U*)›
⟨*proof*⟩

lemma *decomp-nth-eq-lit-eq*:
  assumes
    ‹$M = M2$ @ *Propagated L C'* # *M1*› **and**
    ‹*rev M* ! i = *Propagated L C*› **and**
    ‹*no-dup M*› **and**
    ‹$i <$ *length M*›
  shows ‹*length M1* = i› **and** ‹$C = C'$›
⟨*proof*⟩

lemma
  assumes ‹*no-dup M*›
  shows
    no-dup-same-annotD:
        ‹*Propagated L C* ∈ *set M* ⟹ *Propagated L C'* ∈ *set M* ⟹ $C = C'$› **and**
     no-dup-no-propa-and-dec:
       ‹*Propagated L C* ∈ *set M* ⟹ *Decided L* ∈ *set M* ⟹ *False*›
  ⟨*proof*⟩

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*⟩

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

lemma *remove-one-annot-true-clause-one-imp-spec*:
  assumes
    I: ‹*remove-one-annot-true-clause-imp-inv S iT*› **and**
    cond: ‹*case iT of* (*i*, *S*) ⇒ $i <$ *length* (*get-trail-l S*)› **and**
    iT: ‹*iT* = (*i*, *T*)› **and**
    proped: ‹*is-proped* (*rev* (*get-trail-l S*) ! *i*)›
  shows ‹*remove-one-annot-true-clause-one-imp i T*
        ≤ *SPEC* (λ*s'*. *remove-one-annot-true-clause-imp-inv S s'* ∧
              (*s'*, *iT*) ∈ *measure* (λ(*i*, -). *length* (*get-trail-l S*) − *i*))›
⟨*proof*⟩

lemma *remove-one-annot-true-clause-count-dec*: ‹*remove-one-annot-true-clause S b* ⟹
   count-decided (*get-trail-l S*) = *count-decided* (*get-trail-l b*)›
  ⟨*proof*⟩

lemma *rtranclp-remove-one-annot-true-clause-count-dec*:
  ‹*remove-one-annot-true-clause*** *S b* ⟹

count-decided (get-trail-l S) = count-decided (get-trail-l b)⟩
⟨proof⟩

lemma *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)⟩
⟨proof⟩

lemma *remove-one-annot-true-clause-imp-spec-lev0*:
assumes
ST: ⟨(S, T) ∈ twl-st-l None⟩ **and**
list-invs: ⟨twl-list-invs S⟩ **and**
struct-invs: ⟨twl-struct-invs T⟩ **and**
⟨get-conflict-l S = None⟩ **and**
⟨clauses-to-update-l S = {#}⟩ **and**
⟨count-decided (get-trail-l S) = 0⟩
shows ⟨remove-one-annot-true-clause-imp S ≤ SPEC(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⟩

definition *collect-valid-indices* :: ⟨- ⇒ nat list nres⟩ **where**
⟨collect-valid-indices S = SPEC (λN. 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 ⟷
(∃ T. (S, T) ∈ twl-st-l None ∧ twl-struct-invs T ∧ twl-list-invs S)⟩

definition *mark-garbage-l*:: ⟨nat ⇒ 'v twl-st-l ⇒ 'v twl-st-l⟩ **where**
⟨mark-garbage-l = (λC (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 ⟶
(length (get-clauses-l S ∝ C) = 2 ⟶
(Propagated (get-clauses-l S ∝ C ! 0) C ∉ set (get-trail-l S)) ∧
(Propagated (get-clauses-l S ∝ C ! 1) C ∉ set (get-trail-l S))) ∧

(*length* (*get-clauses-l* $S \propto C$) > *2* ⟶
(*Propagated* (*get-clauses-l* $S \propto C$! *0*) $C \notin$ *set* (*get-trail-l* S))) ∧
¬*irred* (*get-clauses-l* S) C)›

definition *mark-to-delete-clauses-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*mark-to-delete-clauses-l* = (λ*S*. *do* {
ASSERT(*mark-to-delete-clauses-l-pre* *S*);
xs ← *collect-valid-indices* *S*;
to-keep ← *SPEC*(λ-::*nat*. *True*); — the minimum number of clauses that should be kept.
(-, *S*, -) ← $WHILE_T{}^{mark\text{-}to\text{-}delete\text{-}clauses\text{-}l\text{-}inv\ S\ xs}$
(λ(*i*, *S*, *xs*). *i* < *length xs*)
(λ(*i*, *S*, *xs*). *do* {
if(*xs*!*i* ∉# *dom-m* (*get-clauses-l* *S*)) *then RETURN* (*i*, *S*, *delete-index-and-swap xs i*)
else do {
ASSERT(*0* < *length* (*get-clauses-l* *S*∝(*xs*!*i*)));
can-del ← *SPEC* (*can-delete* *S* (*xs*!*i*));
ASSERT(*i* < *length xs*);
if can-del
then
RETURN (*i*, *mark-garbage-l* (*xs*!*i*) *S*, *delete-index-and-swap xs i*)
else
RETURN (*i*+*1*, *S*, *xs*)
}
})
(*to-keep*, *S*, *xs*);
RETURN S
})›

definition *mark-to-delete-clauses-l-post* **where**
‹*mark-to-delete-clauses-l-post* *S* *T* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *twl-st-l None* ∧ *remove-one-annot-true-clause*** *S* *T* ∧
twl-list-invs *S* ∧ *twl-struct-invs* *S'* ∧ *get-conflict-l* *S* = *None* ∧
clauses-to-update-l *S* = {#})›

lemma *mark-to-delete-clauses-l-spec*:
assumes
ST: ‹(*S*, *S'*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs* *S*› **and**
struct-invs: ‹*twl-struct-invs* *S'*› **and**
confl: ‹*get-conflict-l* *S* = *None*› **and**
upd: ‹*clauses-to-update-l* *S* = {#}›
shows ‹*mark-to-delete-clauses-l* *S* ≤ ⇓ *Id* (*SPEC*(λ*T*. *remove-one-annot-true-clause*** *S* *T* ∧
get-trail-l *S* = *get-trail-l* *T*))›
⟨*proof*⟩

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

        C′ ← SPEC(λi. i ∉# dom-m N′ ∧ i ≠ 0);
  RETURN (fmdrop C N, fmupd C′ (N ∝ C, irred N C) N′, m(C ↦ C′))
       }
       else
        RETURN (N, N′, m))
    (N, N′, (λ-. None));
  RETURN (N′, m)
}›

inductive *GC-remap*
  :: ‹('a, 'b) fmap × ('a ⇒ 'c option) × ('c, 'b) fmap ⇒ ('a, 'b) fmap × ('a ⇒ 'c option) × ('c, 'b) fmap ⇒ bool›
where
remap-cons:
  ‹GC-remap (N, m, new) (fmdrop C N, m(C ↦ C′), fmupd C′ (the (fmlookup N C)) new)›
   if ‹C′ ∉# dom-m new› **and**
      ‹C ∈# dom-m N› **and**
      ‹C ∉ dom m› **and**
      ‹C′ ∉ ran m›

lemma *GC-remap-ran-m-old-new*:
  ‹GC-remap (old, m, new) (old′, m′, new′) ⟹ ran-m old + ran-m new = ran-m old′ + ran-m new′›
  ⟨proof⟩

lemma *GC-remap-init-clss-l-old-new*:
  ‹GC-remap (old, m, new) (old′, m′, new′) ⟹
   init-clss-l old + init-clss-l new = init-clss-l old′ + init-clss-l new′›
  ⟨proof⟩

lemma *GC-remap-learned-clss-l-old-new*:
  ‹GC-remap (old, m, new) (old′, m′, new′) ⟹
   learned-clss-l old + learned-clss-l new = learned-clss-l old′ + learned-clss-l new′›
  ⟨proof⟩

lemma *GC-remap-ran-m-remap*:
  ‹GC-remap (old, m, new) (old′, m′, new′) ⟹ C ∈# dom-m old ⟹ C ∉# dom-m old′ ⟹
        m′ C ≠ None ∧
        fmlookup new′ (the (m′ C)) = fmlookup old C›
  ⟨proof⟩

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

lemma *GC-remap-ran-m-no-rewrite-fmap*:
  ‹GC-remap (old, m, new) (old′, m′, new′) ⟹ C ∈# dom-m new ⟹
   C ∈# dom-m new′ ∧ fmlookup new C = fmlookup new′ C›
  ⟨proof⟩

lemma *rtranclp-GC-remap-init-clss-l-old-new*:
  ‹GC-remap** S S′ ⟹
   init-clss-l (fst S) + init-clss-l (snd (snd S)) = init-clss-l (fst S′) + init-clss-l (snd (snd S′))›
  ⟨proof⟩

lemma *rtranclp-GC-remap-learned-clss-l-old-new*:
‹$GC\text{-}remap^{**}$ *S S'* ⟹
learned-clss-l (*fst S*) + *learned-clss-l* (*snd* (*snd S*)) =
learned-clss-l (*fst S'*) + *learned-clss-l* (*snd* (*snd S'*))›
⟨*proof*⟩

lemma *rtranclp-GC-remap-ran-m-no-rewrite-fmap*:
‹$GC\text{-}remap^{**}$ *S S'* ⟹ *C* ∈# *dom-m* (*snd* (*snd S*)) ⟹
C ∈# *dom-m* (*snd* (*snd S'*)) ∧ *fmlookup* (*snd* (*snd S*)) *C* = *fmlookup* (*snd* (*snd S'*)) *C*›
⟨*proof*⟩

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

lemma *GC-remap-ran-m-lookup-kept*:
assumes
‹$GC\text{-}remap^{**}$ *S y*› **and**
‹*GC-remap y z*› **and**
‹*C* ∈# *dom-m* (*fst S*)› **and**
‹*C* ∈# *dom-m* (*fst z*)› **and**
‹*C* ∉# *dom-m* (*fst y*)›
shows ‹*fmlookup* (*fst S*) *C* = *fmlookup* (*fst z*) *C*›
⟨*proof*⟩

lemma *rtranclp-GC-remap-ran-m-no-rewrite*:
‹$GC\text{-}remap^{**}$ *S S'* ⟹ *C* ∈# *dom-m* (*fst S*) ⟹ *C* ∈# *dom-m* (*fst S'*) ⟹
fmlookup (*fst S*) *C* = *fmlookup* (*fst S'*) *C*›
⟨*proof*⟩

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

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

lemma *GC-remap-ran-m-no-new-lost*:
‹*GC-remap S S'* ⟹ *dom* (*fst* (*snd S*)) ⊆ *set-mset* (*dom-m* (*fst S*)) ⟹
dom (*fst* (*snd S'*)) ⊆ *set-mset* (*dom-m* (*fst S*))›
⟨*proof*⟩

lemma *rtranclp-GC-remap-ran-m-no-new-lost*:
‹$GC\text{-}remap^{**}$ *S S'* ⟹ *dom* (*fst* (*snd S*)) ⊆ *set-mset* (*dom-m* (*fst S*)) ⟹
dom (*fst* (*snd S'*)) ⊆ *set-mset* (*dom-m* (*fst S*))›
⟨*proof*⟩

lemma *rtranclp-GC-remap-map-ran*:
assumes
‹$GC\text{-}remap^{**}$ *S S'*› **and**
‹(*the* ∘∘ *fst*) (*snd S*) ‘# *mset-set* (*dom* (*fst* (*snd S*))) = *dom-m* (*snd* (*snd S*))› **and**

‹*finite* (*dom* (*fst* (*snd* S)))›
shows ‹*finite* (*dom* (*fst* (*snd* S'))) ∧
(*the* ∘∘ *fst*) (*snd* S') ‘# *mset-set* (*dom* (*fst* (*snd* S'))) = *dom-m* (*snd* (*snd* S'))›
⟨*proof*⟩

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

lemma *rtranclp-GC-remap-learned-clss-lD*:
‹*GC-remap*** (N, x, m) (N', x', m') ⟹ *learned-clss-l* N + *learned-clss-l* m = *learned-clss-l* N' + *learned-clss-l* m'›
⟨*proof*⟩

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

lemma *remap-cons2*:
assumes
‹C' ∉# *dom-m new*› **and**
‹C ∈# *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**
‹⋀*x*. *x* ∈# *dom-m* (*fst* (N, *m*, *new*)) ⟹ *x* ∉ *dom* (*fst* (*snd* (N, *m*, *new*)))› **and**
‹*finite* (*dom m*)›
shows
‹*GC-remap* (N, *m*, *new*) (*fmdrop* C N, *m*(C ↦ C'), *fmupd* C' (*the* (*fmlookup* N C)) *new*)›
⟨*proof*⟩

inductive-cases *GC-remapE*: ‹*GC-remap* S T›

lemma *rtranclp-GC-remap-finite-map*:
‹*GC-remap*** S S' ⟹ *finite* (*dom* (*fst* (*snd* S))) ⟹ *finite* (*dom* (*fst* (*snd* S')))›
⟨*proof*⟩

lemma *rtranclp-GC-remap-old-dom-map*:
‹*GC-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)))›
⟨*proof*⟩

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-remap*** R S› **and**
C': ‹C' ∉# *dom-m* (*snd* (*snd* S))› **and**
C: ‹C ∈# *dom-m* (*fst* S)›
shows
‹*GC-remap*** R (*fmdrop* C (*fst* S), (*fst* (*snd* S))(C ↦ C'), *fmupd* C' (*the* (*fmlookup* (*fst* S) C)) (*snd* (*snd* S)))›

⟨*proof*⟩

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

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)›
⟨*proof*⟩

definition *cdcl-twl-full-restart-l-GC-prog-pre*
:: ‹′*v twl-st-l* ⇒ *bool*›
where
‹*cdcl-twl-full-restart-l-GC-prog-pre S* ⟷
(∃ *T*. (*S*, *T*) ∈ *twl-st-l None* ∧ *twl-struct-invs T* ∧ *twl-list-invs S* ∧

get-conflict T = None)›

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*⟩

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

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*)›
⟨*proof*⟩

lemma *valid-trail-reduction-lit-of-nth*:
‹*valid-trail-reduction M M′* ⟹ *length M = length M′* ⟹ *i < length M* ⟹
lit-of (*M* ! *i*) = *lit-of* (*M′* ! *i*)›
⟨*proof*⟩

lemma *cdcl-twl-restart-l-lit-of-nth*:
‹*cdcl-twl-restart-l S U* ⟹ *i < length* (*get-trail-l U*) ⟹ *is-proped* (*get-trail-l U* ! *i*) ⟹
length (*get-trail-l S*) = *length* (*get-trail-l U*) ⟹
lit-of (*get-trail-l S* ! *i*) = *lit-of* (*get-trail-l U* ! *i*)›
⟨*proof*⟩

lemma *valid-trail-reduction-is-decided-nth*:
‹*valid-trail-reduction M M′* ⟹ *length M = length M′* ⟹ *i < length M* ⟹
is-decided (*M* ! *i*) = *is-decided* (*M′* ! *i*)›
⟨*proof*⟩

lemma *cdcl-twl-restart-l-mark-of-same-or-0*:
‹*cdcl-twl-restart-l S U* ⟹ *i < length* (*get-trail-l U*) ⟹ *is-proped* (*get-trail-l U* ! *i*) ⟹
length (*get-trail-l S*) = *length* (*get-trail-l U*) ⟹
(*mark-of* (*get-trail-l U* ! *i*) > *0* ⟹ *mark-of* (*get-trail-l S* ! *i*) > *0* ⟹
mset (*get-clauses-l S* ∝ *mark-of* (*get-trail-l S* ! *i*))
= *mset* (*get-clauses-l U* ∝ *mark-of* (*get-trail-l U* ! *i*)) ⟹ *P*) ⟹
(*mark-of* (*get-trail-l U* ! *i*) = *0* ⟹ *P*) ⟹ *P*›
⟨*proof*⟩

lemma *cdcl-twl-full-restart-l-GC-prog-cdcl-twl-restart-l*:
  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* = {#}› **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*⟩

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*⟩

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*, S_0, *0*);

if ¬*brk then 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*, *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* 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*, S_0, *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* S_0 =
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*, S_0, *0*);
  RETURN T
}›

definition *cdcl-twl-stgy-restart-prog-early-l* :: *'v twl-st-l* ⇒ *'v twl-st-l nres* **where**
‹*cdcl-twl-stgy-restart-prog-early-l* S_0 =
do {
  ebrk ← *RES UNIV*;
  (*ebrk*, *brk*, *T*, *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*⟩

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*) ∈
{(*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*›
⟨*proof*⟩

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*)
∈ {(*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*›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-prog-l-cdcl-twl-stgy-restart-abs-l*:
‹(*cdcl-twl-stgy-restart-prog-l*, *cdcl-twl-stgy-restart-abs-l*) ∈ {(*S*, *S'*).
(*S*, *S'*) ∈ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* = {#}} $\rightarrow_f$ ⟨*Id*⟩ *nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ -›)
⟨*proof*⟩

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

$\in \{(S, S'). (S, S') \in$ *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*›
⟨*proof*⟩

definition *cdcl-twl-stgy-restart-prog-bounded-l* :: *'v twl-st-l* ⇒ (*bool* × *'v twl-st-l*) *nres* **where**
‹*cdcl-twl-stgy-restart-prog-bounded-l* S_0 =
do {
ebrk ← *RES UNIV*;
(*ebrk*, *brk*, *T*, *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*);
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*) ∈
$\{(S, S'). (S, S') \in$ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S $= \{\#\}\} \rightarrow_f$
⟨*bool-rel* $\times_r$ $\{(S, S'). (S, S') \in$ *twl-st-l None* ∧ *twl-list-invs S*$\}$⟩ *nres-rel*›
⟨*proof*⟩

lemma *cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-abs-bounded-l*:
‹(*cdcl-twl-stgy-restart-prog-bounded-l*, *cdcl-twl-stgy-restart-abs-bounded-l*) ∈ $\{(S, S').$
$(S, S') \in$ *Id* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* $= \{\#\}\} \rightarrow_f$ ⟨*Id*⟩ *nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ -›)
⟨*proof*⟩

lemma (**in** *twl-restart*) *cdcl-twl-stgy-restart-prog-bounded-l-cdcl-twl-stgy-restart-prog-bounded*:
‹(*cdcl-twl-stgy-restart-prog-bounded-l*, *cdcl-twl-stgy-restart-prog-bounded*)
$\in \{(S, S'). (S, S') \in$ *twl-st-l None* ∧ *twl-list-invs S* ∧ *clauses-to-update-l S* $= \{\#\}\} \rightarrow_f$
⟨*bool-rel* $\times_r$ $\{(S, S'). (S, S') \in$ *twl-st-l None* ∧ *twl-list-invs S*$\}$⟩*nres-rel*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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* {#} = {#}›
⟨*proof*⟩

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* = ⋃# *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*)›
⟨*proof*⟩

fun *correct-watching-except* :: ‹*nat* ⇒ *nat* ⇒ ′*v literal* ⇒ ′*v twl-st-wl* ⇒ *bool*› **where**
‹*correct-watching-except i j K* (*M*, *N*, *D*, *NE*, *UE*, *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*⟩

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* ∈ *a* ⟹ ∀ *x*∈*a*. *P x* ⟹ *P x*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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

definition *all-lits-of-m* :: ‹′*a clause* ⇒ ′*a literal multiset*› **where**
‹*all-lits-of-m Ls* = *Pos* ‘# (*atm-of* ‘# *Ls*) + *Neg* ‘# (*atm-of* ‘# *Ls*)›

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

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

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

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

lemma *all-lits-of-mm-add-mset*:
‹*all-lits-of-mm* (*add-mset C N*) = (*all-lits-of-m C*) + (*all-lits-of-mm N*)›
⟨*proof*⟩

lemma *all-lits-of-m-add-mset*:
‹*all-lits-of-m* (*add-mset L C*) = *add-mset L* (*add-mset* (−*L*) (*all-lits-of-m C*))›
⟨*proof*⟩

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

lemma *all-lits-of-m-mono*:
‹$D \subseteq\# D' \Longrightarrow$ *all-lits-of-m* $D \subseteq\#$ *all-lits-of-m* D'›
⟨*proof*⟩

lemma *in-all-lits-of-mm-uminusD*: ‹$x2 \in\#$ *all-lits-of-mm* $N \Longrightarrow -x2 \in\#$ *all-lits-of-mm* N›
⟨*proof*⟩

lemma *in-all-lits-of-mm-uminus-iff*: ‹$-x2 \in\#$ *all-lits-of-mm* $N \longleftrightarrow x2 \in\#$ *all-lits-of-mm* N›
⟨*proof*⟩

lemma *all-lits-of-mm-diffD*:
‹$L \in\#$ *all-lits-of-mm* $(A - B) \Longrightarrow L \in\#$ *all-lits-of-mm* A›
⟨*proof*⟩

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)›
⟨*proof*⟩

fun *st-l-of-wl* :: ‹($'v$ *literal* × *nat*) *option* ⇒ $'v$ *twl-st-wl* ⇒ $'v$ *twl-st-l*› **where**
‹*st-l-of-wl None* $(M, N, D, NE, UE, Q, W) = (M, N, D, NE, UE, \{\#\}, Q)$›
| ‹*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* × *nat*) *option* ⇒ ($'v$ *twl-st-wl* × $'v$ *twl-st-l*) *set*› **where**
‹*state-wl-l* $L = \{(T, T').\ T' =$ *st-l-of-wl* $L\ T\}$›

fun *twl-st-of-wl* :: ‹($'v$ *literal* × *nat*) *option* ⇒ ($'v$ *twl-st-wl* × $'v$ *twl-st*) *set*› **where**
‹*twl-st-of-wl* $L =$ *state-wl-l* L O *twl-st-l* (*map-option fst* L)›

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

lemma [*twl-st-wl*]:
assumes ‹$(S, T) \in$ *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›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹$(a, a') \in$ *state-wl-l None* $\Longrightarrow$
get-learned-clss-l $a' =$ *get-learned-clss-wl* a›
⟨*proof*⟩

lemma *remove-one-lit-from-wq-def*:
‹*remove-one-lit-from-wq L S = set-clauses-to-update-l* (*clauses-to-update-l S* − {#*L*#}) *S*›
⟨*proof*⟩

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

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

lemma *get-conflict-wl-set-literals-to-update-wl*[*twl-st-wl*]:
‹*get-conflict-wl* (*set-literals-to-update-wl P S*) = *get-conflict-wl S*›
‹*get-unit-clauses-wl* (*set-literals-to-update-wl P S*) = *get-unit-clauses-wl S*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *state-wl-l-mark-of-is-decided*:
‹(*x*, *y*) ∈ *state-wl-l b* ⟹
  get-trail-wl x ≠ [] ⟹
  is-decided (*hd* (*get-trail-l y*)) = *is-decided* (*hd* (*get-trail-wl x*))›
⟨*proof*⟩

lemma *state-wl-l-mark-of-is-proped*:
‹(*x*, *y*) ∈ *state-wl-l b* ⟹
  get-trail-wl x ≠ [] ⟹
  is-proped (*hd* (*get-trail-l y*)) = *is-proped* (*hd* (*get-trail-wl x*))›
⟨*proof*⟩

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

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

definition *unit-prop-body-wl-inv* **where**
‹*unit-prop-body-wl-inv T j i L* ⟷ (*i* < *length* (*watched-by T L*) ∧ *j* ≤ *i* ∧
  (*fst* (*watched-by T L* ! *i*) ∈# *dom-m* (*get-clauses-wl T*) ⟶
   (∃ *T′*. (*T*, *T′*) ∈ *state-wl-l* (*Some* (*L*, *i*)) ∧ *j* ≤ *i* ∧
   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′*)∧
   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)))›

lemma *unit-prop-body-wl-inv-alt-def*:
‹*unit-prop-body-wl-inv T j i L* ⟷ (*i* < *length* (*watched-by T L*) ∧ *j* ≤ *i* ∧

(fst (watched-by T L ! i) ∈# dom-m (get-clauses-wl T) ⟶
(∃ T′. (T, T′) ∈ state-wl-l (Some (L, i)) ∧
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′)∧
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 ∧
get-conflict-wl T = None ∧
length (get-clauses-wl T ∝ fst (watched-by T L ! i)) ≥ 2)))›
(**is** ‹?A = ?B›)
⟨proof⟩

definition propagate-lit-wl-general :: ‹′v literal ⇒ nat ⇒ nat ⇒ ′v twl-st-wl ⇒ ′v twl-st-wl› **where**
‹propagate-lit-wl-general = (λL′ 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 = (λL′ 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 = (λL′ 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 = (λL 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)›
⟨proof⟩

lemma watched-by-keep-watch-neq[twl-st-wl, simp]:
‹w < length (watched-by S L) ⟹ watched-by (keep-watch L j w S) L ! w = watched-by S L ! w›
⟨proof⟩

lemma watched-by-keep-watch-eq[twl-st-wl, simp]:
‹j < length (watched-by S L) ⟹ watched-by (keep-watch L j w S) L ! j = watched-by S L ! w›
⟨proof⟩

definition update-clause-wl :: ‹′v literal ⇒ 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*⟩

lemma *correct-watching-except-update-blit*:
assumes
corr: ‹*correct-watching-except i j L* (*a*, *b*, *c*, *d*, *e*, *f*, *g*(*L* := (*g L*)[*j*′ := (*x1*, *C*, *b*′)]))› **and**
C′: ‹*C*′ ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf b* + (*d* + *e*))›
‹*C*′ ∈ *set* (*b* ∝ *x1*)›
‹*C*′ ≠ *L*› **and**
corr-watched: ‹*correctly-marked-as-binary b* (*x1*, *C*′, *b*′)›
shows ‹*correct-watching-except i j L* (*a*, *b*, *c*, *d*, *e*, *f*, *g*(*L* := (*g L*)[*j*′ := (*x1*, *C*′, *b*′)]))›
⟨*proof*⟩

lemma *correct-watching-except-correct-watching-except-Suc-notin*:
assumes
‹*fst* (*watched-by S L* ! *w*) ∉# *dom-m* (*get-clauses-wl S*)› **and**
j-w: ‹*j* ≤ *w*› **and**
w-le: ‹*w* < *length* (*watched-by S L*)› **and**
corr: ‹*correct-watching-except j w L S*›
shows ‹*correct-watching-except j* (*Suc w*) *L* (*keep-watch L j w S*)›
⟨*proof*⟩

lemma *correct-watching-except-correct-watching-except-update-clause*:
assumes
corr: ‹*correct-watching-except* (*Suc j*) (*Suc w*) *L*
(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*(*L* := (*W L*)[*j* := *W L* ! *w*]))› **and**
j-w: ‹*j* ≤ *w*› **and**
w-le: ‹*w* < *length* (*W L*)› **and**
L′: ‹*L*′ ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf N* + (*NE* + *UE*))›
‹*L*′ ∈ *set* (*N* ∝ *x1*)›**and**

L-L: ‹L ∈# all-lits-of-mm ({#mset (fst x). x ∈# ran-m N#} + (NE + UE))› **and**
L: ‹L ≠ N ∝ x1 ! xa› **and**
dom: ‹x1 ∈# dom-m N› **and**
i-xa: ‹i < length (N ∝ x1)› ‹xa < length (N ∝ x1)› **and**
[*simp*]: ‹W L ! w = (x1, x2, b)› **and**
N-i: ‹N ∝ x1 ! i = L› ‹N ∝ x1 ! (1 −i) ≠ L›‹N ∝ x1 ! xa ≠ L› **and**
N-xa: ‹N ∝ x1 ! xa ≠ N ∝ x1 ! i› ‹N ∝ x1 ! xa ≠ N ∝ x1 ! (Suc 0 − i)›**and**
i-2: ‹i < 2› **and** ‹xa ≥ 2› **and**
L-neq: ‹L′ ≠ N ∝ x1 ! xa› — The new blocking literal is not the new watched literal.
shows ‹correct-watching-except j (Suc w) L
(M, N(x1 ↪ swap (N ∝ x1) i xa), D, NE, UE, Q, W
(L := (W L)[j := (x1, x2, b)],
N ∝ x1 ! xa := W (N ∝ x1 ! xa) @ [(x1, L′, b)]))›
⟨*proof*⟩

definition *unit-propagation-inner-loop-wl-loop-pre* **where**
‹unit-propagation-inner-loop-wl-loop-pre L = (λ(j, w, S).
w < length (watched-by S L) ∧ j ≤ w ∧
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* ⟷
  C ∈# *dom-m* (*get-clauses-wl S*) ∧ *length* ((*get-clauses-wl S*)∝*C*) = *2* ∧
  set (*get-clauses-wl S*∝*C*) = {*L*, *L′*} ∧ *L* ≠ *L′*›

definition *unit-propagation-inner-loop-body-wl* :: ‹′*v literal* ⇒ *nat* ⇒ *nat* ⇒ ′*v twl-st-wl* ⇒ (*nat* × *nat* × ′*v twl-st-wl*) *nres*› **where**
‹*unit-propagation-inner-loop-body-wl L j w S* = *do* {
    ASSERT(*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*));
    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* ∝ *C*) *S*)
        else do { — This is non-optimal (memory access: relax invariant!):
          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*)}
      } — 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
            }
        }
      }

}
}›

lemma [*twl-st-wl*]: ‹*get-clauses-wl* (*keep-watch L j w S*) = *get-clauses-wl S*›
⟨*proof*⟩

lemma *unit-propagation-inner-loop-body-wl-int-alt-def*:
‹*unit-propagation-inner-loop-body-wl-int L j w S* = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*));
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 — 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')∝*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'
}
}
}
}
}›
⟨*proof*⟩

1.4.3 The Functions

Inner Loop

```
lemma clause-to-update-mapsto-upd-If:
  assumes
    i: ‹i ∈# dom-m N›
  shows
  ‹clause-to-update L (M, N(i ↪ C′), C, NE, UE, WS, Q) =
    (if L ∈ 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⟩

lemma unit-propagation-inner-loop-body-l-with-skip-alt-def:
  ‹unit-propagation-inner-loop-body-l-with-skip L (S′, n) = do {
      ASSERT (clauses-to-update-l S′ ≠ {#} ∨ 0 < n);
      ASSERT (unit-propagation-inner-loop-l-inv L (S′, n));
      b ← SPEC (λb. (b ⟶ 0 < n) ∧ (¬ b ⟶ clauses-to-update-l S′ ≠ {#}));
      if ¬ b
      then do {
        ASSERT (clauses-to-update-l S′ ≠ {#});
        X2 ← select-from-clauses-to-update S′;
        ASSERT (unit-propagation-inner-loop-body-l-inv L (snd X2) (fst X2));
        x ← SPEC (λK. K ∈ set (get-clauses-l (fst X2) ∝ snd X2));
        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) ∝ snd X2 ! 0 = L then 0 else 1;
                    va = get-clauses-l (fst X2) ∝ 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 ← find-unwatched-l (get-trail-l (fst X2)) (get-clauses-l (fst X2) ∝ snd X2);
             case x of
             None ⇒
               if vaa = Some False
               then let T = set-conflict-l (get-clauses-l (fst X2) ∝ 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 ← ASSERT (a < length (get-clauses-l (fst X2) ∝ snd X2));
                  let K = (get-clauses-l (fst X2) ∝ (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 ← update-clause-l (snd X2) v a (fst X2);
                         RETURN (T, if get-conflict-l T = None then n else 0)
                       }
                }
             }
        }
      else RETURN (S′, n − 1)
    }›
```

⟨*proof*⟩

lemma *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*⟩
⟨*proof*⟩
declare *twl-st-wl*[*simp*]

lemma *correct-watching-except-correct-watching-except-propagate-lit-wl*:
assumes
corr: ⟨*correct-watching-except j w L S*⟩ **and**
i-le: ⟨*Suc 0* < *length* (*get-clauses-wl S* ∝ *C*)⟩ **and**
C: ⟨*C* ∈# *dom-m* (*get-clauses-wl S*)⟩
shows ⟨*correct-watching-except j w L* (*propagate-lit-wl-general L′ C i S*)⟩
⟨*proof*⟩

lemma *unit-propagation-inner-loop-body-wl-int-alt-def2*:
⟨*unit-propagation-inner-loop-body-wl-int L j w S* = *do* {
ASSERT(*unit-propagation-inner-loop-wl-loop-pre L* (*j*, *w*, *S*));
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 ∉# *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
}
}
}
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
            }
        }
      }
    }
  }›
⟨proof⟩
```

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

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

  defines
    [*simp*]: ‹*C''* ≡ *get-clauses-l S'* ∝ *C'*›
  assumes
    S-S': ‹(*S*, *S'*) ∈ *state-wl-l* (*Some* (*L*, *w*))› **and**
    w-le: ‹*w* < *length* (*watched-by S L*)› **and**
    j-w: ‹*j* ≤ *w*› **and**
    corr-w: ‹*correct-watching-except j w L S*› **and**
    inner-loop-inv: ‹*unit-propagation-inner-loop-wl-loop-inv L* (*j*, *w*, *S*)› **and**
    n: ‹*n* = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl S*)) (*mset* (*drop w* (*watched-by S L*))))›
and
    confl-S: ‹*get-conflict-wl S* = *None*›
  shows *unit-propagation-inner-loop-body-wl-wl-int*: ‹*unit-propagation-inner-loop-body-wl L j w S* ≤
    ⇓ *Id* (*unit-propagation-inner-loop-body-wl-int L j w S*)›
⟨*proof*⟩

lemma

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-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'↪ (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⟩

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-spec: ⟨unit-propagation-inner-loop-body-wl 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′ ∧
j ≤ *length* (*watched-by T′ L*) ∧
length (*watched-by S L*) = *length* (*watched-by T′ L*) ∧
i ≤ *j* ∧
(*get-conflict-wl T′* = *None* ⟶
n = *size* (*filter-mset* (λ(*i*, -). *i* ∉# *dom-m* (*get-clauses-wl T′*)) (*mset* (*drop j* (*watched-by T′ L*))))) ∧
(*get-conflict-wl T′* ≠ *None* ⟶ *n* = *0*)}
(*unit-propagation-inner-loop-body-l-with-skip L* (*S′*, *n*))›
⟨*proof*⟩

definition *unit-propagation-inner-loop-wl-loop*
:: ‹*′v literal* ⇒ *′v twl-st-wl* ⇒ (*nat* × *nat* × *′v twl-st-wl*) *nres*› **where**
‹*unit-propagation-inner-loop-wl-loop L* S_0 = *do* {
let n = *length* (*watched-by* S_0 *L*);
$WHILE_T$ *unit-propagation-inner-loop-wl-loop-inv L*
(λ(*j*, *w*, *S*). *w* < *n* ∧ *get-conflict-wl S* = *None*)
(λ(*j*, *w*, *S*). *do* {
unit-propagation-inner-loop-body-wl L j w S
})
(*0*, *0*, S_0)
}›

lemma *correct-watching-except-correct-watching-cut-watch*:
assumes *corr*: ‹*correct-watching-except j w L* (*a*, *b*, *c*, *d*, *e*, *f*, *g*)›
shows ‹*correct-watching* (*a*, *b*, *c*, *d*, *e*, *f*, *g*(*L* := *take j* (*g L*) @ *drop w* (*g L*)))›
⟨*proof*⟩

lemma *unit-propagation-inner-loop-wl-loop-alt-def*:
‹*unit-propagation-inner-loop-wl-loop L* S_0 = *do* {
let (- :: *nat*) = (*if get-conflict-wl* S_0 = *None then remaining-nondom-wl 0 L* S_0 *else 0*);
let n = *length* (*watched-by* S_0 *L*);
$WHILE_T$ *unit-propagation-inner-loop-wl-loop-inv L*
(λ(*j*, *w*, *S*). *w* < *n* ∧ *get-conflict-wl S* = *None*)
(λ(*j*, *w*, *S*). *do* {
unit-propagation-inner-loop-body-wl L j w S
})
(*0*, *0*, S_0)
}
›
⟨*proof*⟩

definition *cut-watch-list* :: ‹*nat* ⇒ *nat* ⇒ *′v literal* ⇒ *′v twl-st-wl* ⇒ *′v twl-st-wl nres*› **where**
‹*cut-watch-list j w L* =(λ(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *do* {
ASSERT(*j* ≤ *w* ∧ *j* ≤ *length* (*W L*) ∧ *w* ≤ *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*) ← *unit-propagation-inner-loop-wl-loop L* S_0;
ASSERT(*j* ≤ *w* ∧ *w* ≤ *length* (*watched-by S L*));

cut-watch-list j w L S
})

lemma *correct-watching-correct-watching-except00*:
⟨*correct-watching S* ⟹ *correct-watching-except 0 0 L S*⟩
⟨*proof*⟩

lemma *unit-propagation-inner-loop-wl-spec*:
shows ⟨(*uncurry unit-propagation-inner-loop-wl*, *uncurry unit-propagation-inner-loop-l*) ∈
{((*L'*, *T'*::*'v twl-st-wl*), (*L*, *T*::*'v twl-st-l*)). *L* = *L'* ∧ (*T'*, *T*) ∈ *state-wl-l* (*Some* (*L*, *0*)) ∧
correct-watching T'} →
⟨{(*T'*, *T*). (*T'*, *T*) ∈ *state-wl-l None* ∧ *correct-watching T'*}⟩ *nres-rel*
⟩ (**is** ⟨*?fg* ∈ *?A* → ⟨*?B*⟩*nres-rel*⟩ **is** ⟨*?fg* ∈ *?A* → ⟨{(*T'*, *T*). - ∧ *?P T T'*}⟩*nres-rel*⟩)
⟨*proof*⟩

Outer loop

definition *select-and-remove-from-literals-to-update-wl* :: ⟨*'v twl-st-wl* ⇒ (*'v twl-st-wl* × *'v literal*) *nres*⟩
where
⟨*select-and-remove-from-literals-to-update-wl S* = *SPEC*(λ(*S'*, *L*). *L* ∈# *literals-to-update-wl S* ∧
S' = *set-literals-to-update-wl* (*literals-to-update-wl S* − {#*L*#}) *S*)⟩

definition *unit-propagation-outer-loop-wl-inv* **where**
⟨*unit-propagation-outer-loop-wl-inv S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l None* ∧
correct-watching S ∧
unit-propagation-outer-loop-l-inv S')⟩

definition *unit-propagation-outer-loop-wl* :: ⟨*'v twl-st-wl* ⇒ *'v twl-st-wl nres*⟩ **where**
⟨*unit-propagation-outer-loop-wl* S_0 =
$WHILE_T$ $^{unit\text{-}propagation\text{-}outer\text{-}loop\text{-}wl\text{-}inv}$
(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* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl S'*) + *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*⟩

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*⟩

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* = (λ*C 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$$^{λ(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*›
⟨*proof*⟩

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*⟩

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(λL′. 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(λS.
  ∃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(λi. 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* = (λL 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* = (λL (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*⟩

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'* ∧ *N* = *N'* ∧ *NE* = *NE'* ∧ *UE* = *UE'* ∧ *WS* = *WS'* ∧ *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'* ∧ *D* = *D'* ∧ *NE* = *NE'* ∧ *UE* = *UE'* ∧ *WS* = *WS'* ∧ *Q* = *Q'*›

lemma *equality-except-conflict-wl-get-clauses-wl*:
‹*equality-except-conflict-wl S Y* ⟹ *get-clauses-wl S* = *get-clauses-wl Y*›
⟨*proof*⟩
lemma *equality-except-trail-wl-get-clauses-wl*:
‹*equality-except-trail-wl S Y*⟹ *get-clauses-wl S* = *get-clauses-wl Y*›
⟨*proof*⟩

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*⟩

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} →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 →f ⟨?B⟩ nres-rel›)
⟨proof⟩
```

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 S0 ≡ λ(brk, T).
     (∃ T' S0'. (T, T') ∈ state-wl-l None ∧
     (S0, S0') ∈ state-wl-l None ∧
     cdcl-twl-stgy-prog-l-inv S0' (brk, T'))›
```

definition *cdcl-twl-stgy-prog-wl* :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**

```
‹cdcl-twl-stgy-prog-wl S0 =
do {
  (brk, T) ← WHILE_T^(cdcl-twl-stgy-prog-wl-inv S0)
    (λ(brk, -). ¬brk)
    (λ(brk, S). do {
      T ← unit-propagation-outer-loop-wl S;
      cdcl-twl-o-prog-wl T
    })
    (False, S0);
  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*⟩

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*⟩

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*⟩

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*⟩

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} $\rightarrow_f$
⟨*state-wl-l None*⟩*nres-rel*›
(**is** ‹*?o* ∈ *?A* $\rightarrow_f$ ⟨*?B*⟩ *nres-rel*›)

⟨*proof*⟩

lemma *cdcl-twl-stgy-prog-break-wl-spec-final*:
assumes
‹*cdcl-twl-stgy-prog-wl-pre S S'*›
shows
‹*cdcl-twl-stgy-prog-break-wl S* ≤ ⇓ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S'*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

definition *remove-one-annot-true-clause-imp-wl-inv* **where**
‹*remove-one-annot-true-clause-imp-wl-inv S* = (λ(*i*, *T*).
(∃ *S' T'*. (*S*, *S'*) ∈ *state-wl-l None* ∧ (*T*, *T'*) ∈ *state-wl-l None* ∧
correct-watching'' S ∧ *correct-watching'' T* ∧
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* = (λ*S*. *do* {
k ← *SPEC*(λ*k*. (∃ *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-one-annot-true-clause-imp-wl-inv S
    (λ(i, S). i < k)
    (λ(i, S). remove-one-annot-true-clause-one-imp-wl i S)
    (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} →_f
    ⟨{(S, T). (S, T) ∈ state-wl-l None ∧ correct-watching'' S}⟩nres-rel›
```

⟨*proof*⟩

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 = (λC (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 = (λS. do {
  ASSERT(mark-to-delete-clauses-wl-pre S);
  xs ← collect-valid-indices-wl S;
  l ← SPEC(λ-:: nat. True);
  (-, S, -) ← WHILE_T^mark-to-delete-clauses-wl-inv S xs
    (λ(i, S, xs). i < length xs)
    (λ(i, T, xs). do {
      if(xs!i ∉# dom-m (get-clauses-wl T)) then RETURN (i, T, delete-index-and-swap xs i)
      else do {
        ASSERT(0 < length (get-clauses-wl T∝(xs!i)));
        can-del ← SPEC(λb. 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*)
$\in \{(S, T).\ (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching′ S*$\} \to_f$
$\langle\{(S, T).\ (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching′ S*$\}\rangle$*nres-rel*›
⟨*proof*⟩

This is only a specification and must be implemented. There are two ways to do so:

1. clean the watch lists and then iterate over all clauses to rebuild them.
2. iterate over the watch list and check whether the clause index is in the domain or not.

   It is not clear which is faster (but option 1 requires only 1 memory access per clause instead of two). The first option is implemented in SPASS-SAT. The latter version (partly) in cadical.

definition *rewatch-clauses* :: ‹*′v twl-st-wl* ⇒ *′v twl-st-wl nres*› **where**
‹*rewatch-clauses* = (λ(*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*). *SPEC*(λ(*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′*)))›

definition *mark-to-delete-clauses-wl-post* **where**
‹*mark-to-delete-clauses-wl-post S T* $\longleftrightarrow$
($\exists$ *S′ T′*. (*S*, *S′*) $\in$ *state-wl-l None* $\wedge$ (*T*, *T′*) $\in$ *state-wl-l None* $\wedge$
mark-to-delete-clauses-l-post S′ T′ $\wedge$ *correct-watching S* $\wedge$
correct-watching T)›

definition *cdcl-twl-full-restart-wl-prog* :: ‹*′v twl-st-wl* ⇒ *′v twl-st-wl nres*› **where**
‹*cdcl-twl-full-restart-wl-prog S* = *do* {
— *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
}›

lemma *correct-watching-correct-watching*: ‹*correct-watching S* $\Longrightarrow$ *correct-watching′ S*›
⟨*proof*⟩

lemma (**in** −) [*twl-st-l*, *simp*]:
‹(*Sa*, *x*) $\in$ *twl-st-l None* $\Longrightarrow$ *get-all-learned-clss x* = *mset ‘#* (*get-learned-clss-l Sa*) + *get-unit-learned-clauses-l Sa*›
⟨*proof*⟩

lemma *cdcl-twl-full-restart-wl-prog-final-rel*:
assumes
S-Sa: ‹(*S*, *Sa*) $\in \{(S, T).\ (S, T) \in$ *state-wl-l None* $\wedge$ *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*) $\in \{(S, T).\ (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching′ S*$\}$› **and**

pre-l: ‹*mark-to-delete-clauses-l-post Sa Ta*›
shows ‹*mark-to-delete-clauses-wl-post S T*›
⟨*proof*⟩

lemma *cdcl-twl-full-restart-wl-prog-final-rel′*:
assumes
S-Sa: ‹$(S, Sa) \in \{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *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: ‹$(T, Ta) \in \{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching′ S*$\}$› **and**
pre-l: ‹*mark-to-delete-clauses-l-post Sa Ta*›
shows ‹*mark-to-delete-clauses-wl-post S T*›
⟨*proof*⟩

lemma *cdcl-twl-full-restart-wl-prog-cdcl-full-twl-restart-l-prog*:
‹(*cdcl-twl-full-restart-wl-prog*, *cdcl-twl-full-restart-l-prog*)
$\in \{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching S*$\} \rightarrow_f$
$\langle\{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching S*$\}\rangle$*nres-rel*›
⟨*proof*⟩

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*)
$\in \{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching S*$\} \rightarrow_f$
$\langle\{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching S*$\}\rangle$*nres-rel*›
⟨*proof*⟩

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*$\} \rightarrow_f$
$\langle\{(S, T). (S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching S*$\}\rangle$*nres-rel*›
⟨*proof*⟩

definition (**in** −) *restart-abs-wl-pre* :: ‹*′v twl-st-wl* ⇒ *bool* ⇒ *bool*› **where**
‹*restart-abs-wl-pre S brk* ⟷
(∃ *S′*. (*S*, *S′*) ∈ *state-wl-l None* ∧ *restart-abs-l-pre S′ brk*
∧ *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* ≡
(∃ $S_0{}'$ *T*′.
(S_0, $S_0{}'$) ∈ *state-wl-l None* ∧
(*T*, *T*′) ∈ *state-wl-l None* ∧
cdcl-twl-stgy-restart-abs-l-inv $S_0{}'$ *brk T*′ *n* ∧
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* ⟷ (
∃ *T*. (*S*, *T*) ∈ *state-wl-l None* ∧
correct-watching′′ *S* ∧
cdcl-GC-clauses-pre T
)›

definition *cdcl-GC-clauses-wl* :: ‹′*v twl-st-wl* ⇒ ′*v twl-st-wl nres*› **where**
‹*cdcl-GC-clauses-wl* = (λ(*M*, *N*, *D*, *NE*, *UE*, *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*′′) ∧
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*›
⟨*proof*⟩

definition *cdcl-twl-full-restart-wl-GC-prog-post* :: ‹′*v twl-st-wl* ⇒ ′*v twl-st-wl* ⇒ *bool*› **where**
‹*cdcl-twl-full-restart-wl-GC-prog-post S T* ⟷
(∃ *S*′ *T*′. (*S*, *S*′) ∈ *state-wl-l None* ∧ (*T*, *T*′) ∈ *state-wl-l None* ∧
cdcl-twl-full-restart-l-GC-prog-pre S′ ∧
cdcl-twl-restart-l 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 = (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-wl2* :: ‹'v twl-st-wl ⇒ 'v twl-st-wl nres› **where**
‹mark-to-delete-clauses-wl2 = (λS. 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(λb. 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⟩

definition *cdcl-twl-full-restart-wl-GC-prog-pre*
:: ‹'v twl-st-wl ⇒ bool›
where
‹cdcl-twl-full-restart-wl-GC-prog-pre S ⟷
(∃T. (S, T) ∈ state-wl-l None ∧ correct-watching' S ∧ 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' ← cdcl-twl-local-restart-wl-spec0 S;
T ← 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
$\leq \Downarrow \{(S, S')$. $(S, S') \in$ *state-wl-l None* $\wedge$ *correct-watching''* $S\}$
(*cdcl-twl-local-restart-l-spec0 y*)›
⟨*proof*⟩

lemma *cdcl-twl-full-restart-wl-GC-prog-post-correct-watching*:
assumes
pre: ‹*cdcl-twl-full-restart-l-GC-prog-pre y*› **and**
y-Va: ‹*cdcl-twl-restart-l 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*⟩

lemma *cdcl-twl-full-restart-wl-GC-prog*:
‹(*cdcl-twl-full-restart-wl-GC-prog*, *cdcl-twl-full-restart-l-GC-prog*) $\in \{(S::'v$ *twl-st-wl*, $S')$.
$(S, S') \in$ *state-wl-l None* $\wedge$ *correct-watching'* $S\} \rightarrow_f \langle\{(S::'v$ *twl-st-wl*, $S')$.
$(S, S') \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\}\rangle$*nres-rel*›
⟨*proof*⟩

definition (**in** *twl-restart-ops*) *restart-prog-wl*
:: *'v twl-st-wl* $\Rightarrow$ *nat* $\Rightarrow$ *bool* $\Rightarrow$ (*'v twl-st-wl* $\times$ *nat*) *nres*
where
‹*restart-prog-wl S n brk* = *do* {
ASSERT(*restart-abs-wl-pre S brk*);
$b \leftarrow$ *restart-required-wl S n*;
$b2 \leftarrow$ *SPEC*(λ-. *True*);
if $b2 \wedge b \wedge \neg brk$ *then do* {
$T \leftarrow$ *cdcl-twl-full-restart-wl-GC-prog S*;
RETURN $(T, n + 1)$
}
else if $b \wedge \neg brk$ *then do* {
$T \leftarrow$ *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*)
$\in \{(S, T)$. $(S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\} \times_f$ *nat-rel* $\times_f$ *bool-rel* $\rightarrow_f$
$\langle\{(S, T)$. $(S, T) \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\} \times_f$ *nat-rel*$\rangle$*nres-rel*›
(**is** ‹- $\in$ *?R* $\times_f$ - $\times_f$ - $\rightarrow_f$ ⟨*?R'*⟩*nres-rel*›)
⟨*proof*⟩

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*⟩

definition (**in** *twl-restart-ops*) *cdcl-twl-stgy-restart-prog-early-wl*
:: ‹′v twl-st-wl ⇒ ′v twl-st-wl nres›
where
‹*cdcl-twl-stgy-restart-prog-early-wl* (S_0::′v twl-st-wl) = *do* {
 ebrk ← *RES UNIV*;
 (-, *brk*, *T*, *n*) ← $WHILE_T{}^{\lambda(-,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}inv\ S_0\ brk\ T\ n}$
 (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*⟩

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*›)
⟨*proof*⟩

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*›)
⟨*proof*⟩

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*) ← $WHILE_T$λ(-, *brk*, *T*, *n*). *cdcl-twl-stgy-restart-abs-wl-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*);
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*)
∈ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*} $\rightarrow_f$
⟨*bool-rel* $\times_r$ {(*S*, *T*). (*S*, *T*) ∈ *state-wl-l None* ∧ *correct-watching S*}⟩*nres-rel*›
(**is** ‹- ∈ *?R* $\rightarrow_f$ ⟨*?S*⟩*nres-rel*›)
⟨*proof*⟩

theorem *cdcl-twl-stgy-restart-prog-bounded-wl-spec*:
‹(*cdcl-twl-stgy-restart-prog-bounded-wl*, *cdcl-twl-stgy-restart-prog-bounded-l*) ∈ {(*S*::'*v twl-st-wl*, *S'*).
(*S*, *S'*) ∈ *state-wl-l None* ∧ *correct-watching S*} → ⟨*bool-rel* $\times_r$ *state-wl-l None*⟩*nres-rel*›

(**is** ‹$?o \in ?A \to \langle ?B\rangle$ *nres-rel*›)
⟨*proof*⟩

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* $\mathcal{A}_{in}$›
⟨*proof*⟩

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-*$\mathcal{L}_{in}$ {#}›
⟨*proof*⟩

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

lemma *literals-are-in-$\mathcal{L}_{in}$-add-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$* (*add-mset* L A) $\longleftrightarrow$ *literals-are-in-$\mathcal{L}_{in}$* $A \wedge L \in\# \mathcal{L}_{all}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-mono*:
assumes N: ‹*literals-are-in-$\mathcal{L}_{in}$* D'› **and** D: ‹$D \subseteq\# D'$›
shows ‹*literals-are-in-$\mathcal{L}_{in}$* D›
⟨*proof*⟩

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

lemma *all-lits-of-m-subset-all-lits-of-mmD*:
‹$a \in\# b \Longrightarrow$ *set-mset* (*all-lits-of-m* a) $\subseteq$ *set-mset* (*all-lits-of-mm* b)›
⟨*proof*⟩

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

lemma *literals-are-in-$\mathcal{L}_{in}$-remdups*[*simp*]:
‹*literals-are-in-$\mathcal{L}_{in}$* (*remdups-mset* N) = *literals-are-in-$\mathcal{L}_{in}$* N›
⟨*proof*⟩

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

definition *literals-are-in-$\mathcal{L}_{in}$-mm* :: ‹*nat clauses* $\Rightarrow$ *bool*› **where**
‹*literals-are-in-$\mathcal{L}_{in}$-mm* $C \longleftrightarrow$ *set-mset* (*all-lits-of-mm* C) $\subseteq$ *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) $\Longrightarrow L \in\# C \Longrightarrow L \in\# \mathcal{L}_{all}$›
⟨*proof*⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-mm-add-mset*:
‹*literals-are-in-$\mathcal{L}_{in}$-mm* (*add-mset* C N) $\longleftrightarrow$
literals-are-in-$\mathcal{L}_{in}$-mm $N \wedge$ *literals-are-in-$\mathcal{L}_{in}$* C›
⟨*proof*⟩

definition *literals-are-in-$\mathcal{L}_{in}$-trail* :: ‹(*nat*, '*mark*) *ann-lits* $\Rightarrow$ *bool*› **where**
‹*literals-are-in-$\mathcal{L}_{in}$-trail* $M \longleftrightarrow$ *set-mset* (*lit-of* ‘# *mset* M) $\subseteq$ *set-mset* $\mathcal{L}_{all}$›

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

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

lemma *literals-are-in-$\mathcal{L}_{in}$-trail-uminus-in-lits-of-l-atms*:

⟨*literals-are-in-*$\mathcal{L}_{in}$*-trail M* $\Longrightarrow$ $-a$ ∈ *lits-of-l M* $\Longrightarrow$ *atm-of a* ∈# $\mathcal{A}_{in}$⟩
⟨*proof*⟩
end

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

lemma $\mathcal{L}_{all}$*-atm-of-all-lits-of-mm*: ⟨*set-mset* ($\mathcal{L}_{all}$ (*atm-of* '# *all-lits-of-mm A*)) = *set-mset* (*all-lits-of-mm A*)⟩
⟨*proof*⟩

definition *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*⟩

lemma *literals-are-in-*$\mathcal{L}_{in}$*-mm-in-*$\mathcal{L}_{all}$:
assumes
N1: ⟨*literals-are-in-*$\mathcal{L}_{in}$*-mm* $\mathcal{A}$ (*mset* '# *ran-mf xs*)⟩ **and**
i-xs: ⟨*i* ∈# *dom-m xs*⟩ **and** *j-xs*: ⟨*j* < *length* (*xs* ∝ *i*)⟩
shows ⟨*xs* ∝ *i* ! *j* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$⟩
⟨*proof*⟩

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

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

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

lemma *literals-are-in-*$\mathcal{L}_{in}$*-trail-lit-of-mset*:
⟨*literals-are-in-*$\mathcal{L}_{in}$*-trail* $\mathcal{A}$ *M* = *literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}$ (*lit-of* '# *mset M*)⟩
⟨*proof*⟩

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

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

lemma *literals-are-in-*$\mathcal{L}_{in}$*-in-*$\mathcal{L}_{all}$:
assumes
N1: ‹*literals-are-in-*$\mathcal{L}_{in}$ $\mathcal{A}$ (*mset xs*)› **and**
i-xs: ‹*i* < *length xs*›
shows ‹*xs* ! *i* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *literals-are-in-*$\mathcal{L}_{in}$*-trail-atm-of*:

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

lemma *literals-are-in-$\mathcal{L}_{in}$-poss-remdups-mset*:
‹literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (poss (remdups-mset (atm-of ‘# C))) ⟷ literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ C›
⟨proof⟩

lemma *literals-are-in-$\mathcal{L}_{in}$-negs-remdups-mset*:
‹literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ (negs (remdups-mset (atm-of ‘# C))) ⟷ literals-are-in-$\mathcal{L}_{in}$ $\mathcal{A}_{in}$ C›
⟨proof⟩

lemma *$\mathcal{L}_{all}$-atm-of-all-lits-of-m*:
‹set-mset ($\mathcal{L}_{all}$ (atm-of ‘# all-lits-of-m C)) = set-mset C ∪ uminus ‘ set-mset C›
⟨proof⟩

lemma *atm-of-all-lits-of-mm*:
‹set-mset (atm-of ‘# all-lits-of-mm bw) = atms-of-mm bw›
‹atm-of ‘ set-mset (all-lits-of-mm bw) = atms-of-mm bw›
⟨proof⟩

lemma *in-set-all-atms-iff*:
‹y ∈# all-atms bu bw ⟷
y ∈ atms-of-mm (mset ‘# ran-mf bu) ∨ y ∈ atms-of-mm bw›
⟨proof⟩

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

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

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

definition *unit-prop-body-wl-D-inv*
:: ‹nat twl-st-wl ⇒ nat ⇒ nat ⇒ nat literal ⇒ bool› **where**
‹unit-prop-body-wl-D-inv T′ j w L ⟷
unit-prop-body-wl-inv T′ j w L ∧ literals-are-$\mathcal{L}_{in}$ (all-atms-st T′) T′ ∧ L ∈# $\mathcal{L}_{all}$ (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 ⟷
unit-prop-body-wl-find-unwatched-inv f C S ∧
(f ≠ None ⟶ the f ≥ 2 ∧ the f < length (get-clauses-wl S ∝ C) ∧
get-clauses-wl S ∝ C ! (the f) ≠ get-clauses-wl S ∝ C ! 0 ∧
get-clauses-wl S ∝ C ! (the f) ≠ get-clauses-wl S ∝ 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-ℒin (all-atms-st S) S ∧ L ∈# ℒall (all-atms-st S) ∧
    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) ∧
   unit-propagation-inner-loop-wl-loop-pre L (j, w, S))›
```

definition *unit-propagation-inner-loop-body-wl-D*

```
:: ‹nat literal ⇒ nat ⇒ nat ⇒ nat twl-st-wl ⇒
   (nat × nat × 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 ∝ 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)
        }
      } — 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-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 ∝ C) S)
              else 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
            }
        }
      }
```

}
}›

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* ∝ (*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* ∝ (*fst* (*watched-by S K* ! *w*)))› **and**
w-le: ‹*w* < *length* (*watched-by S K*)›
shows ‹*get-clauses-wl S* ∝ *x1* ! *y* =
get-clauses-wl S ∝ *x1* ! *y'* ⟷ *y* = *y'*› (**is** ‹*?eq* ⟷ *?y*›)
⟨*proof*⟩

lemma *in-all-lits-uminus-iff*[*simp*]: ‹(− *xa* ∈# *all-lits N NUE*) = (*xa* ∈# *all-lits N NUE*)›
⟨*proof*⟩

lemma *is-$\mathcal{L}_{all}$-all-atms-st-all-lits-st*[*simp*]:
‹*is-$\mathcal{L}_{all}$* (*all-atms-st S*) (*all-lits-st S*)›
⟨*proof*⟩

lemma *literals-are-$\mathcal{L}_{in}$-all-atms-st*:
‹*blits-in-$\mathcal{L}_{in}$ S* ⟹ *literals-are-$\mathcal{L}_{in}$* (*all-atms-st S*) *S*›
⟨*proof*⟩

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*⟩

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* ⟹ *w* < *length* (*watched-by S K*) ⟹ *literals-are-$\mathcal{L}_{in}$* $\mathcal{A}$ (*keep-watch K j w S*)›
⟨*proof*⟩

lemma *all-lits-update-swap*[*simp*]:
‹*x1* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1*) ⟹*n'* < *length* (*x1aa* ∝ *x1*) ⟹
all-lits (*x1aa*(*x1* ↪ *swap* (*x1aa* ∝ *x1*) *n n'*)) = *all-lits x1aa*›
⟨*proof*⟩

lemma *blits-in-$\mathcal{L}_{in}$-propagate*:
‹*x1* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1*) ⟹ *n'* < *length* (*x1aa* ∝ *x1*) ⟹
blits-in-$\mathcal{L}_{in}$ (*Propagated A x1'* # *x1b*, *x1aa*
(*x1* ↪ *swap* (*x1aa* ∝ *x1*) *n n'*), *D*, *x1c*, *x1d*,
add-mset A' x1e, *x2e*) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)›
‹*x1* ∈# *dom-m x1aa* ⟹ *n* < *length* (*x1aa* ∝ *x1*) ⟹ *n'* < *length* (*x1aa* ∝ *x1*) ⟹
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*
(*x1* ↪ *swap* (*x1aa* ∝ *x1*) *n n'*), *D*, *x1c*, *x1d*,*x1e*, *x2e*) ⟷
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)›
‹*blits-in-$\mathcal{L}_{in}$*

(*Propagated A x1′ # x1b*, *x1aa*, *D*, *x1c*, *x1d*,
add-mset A′ x1e, *x2e*) $\longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)›
‹*x1′* ∈# *dom-m x1aa* $\Longrightarrow$ *n* < *length* (*x1aa* ∝ *x1′*) $\Longrightarrow$ *n′* < *length* (*x1aa* ∝ *x1′*) $\Longrightarrow$
K ∈# $\mathcal{L}_{all}$ (*all-atms-st* (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)) $\Longrightarrow$ *blits-in-*$\mathcal{L}_{in}$
(*x1a*, *x1aa*(*x1′* ↪ *swap* (*x1aa* ∝ *x1′*) *n n′*), *D*, *x1c*, *x1d*,
x1e, *x2e*
(*x1aa* ∝ *x1′* ! *n′* :=
x2e (*x1aa* ∝ *x1′* ! *n′*) @ [(*x1′*, *K*, *b′*)])) $\longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)›
‹*K* ∈# $\mathcal{L}_{all}$ (*all-atms-st* (*x1b*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)) $\Longrightarrow$
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*,
x1e, *x2e*
(*x1aa* ∝ *x1′* ! *n′* := *x2e* (*x1aa* ∝ *x1′* ! *n′*) @ [(*x1′*, *K*, *b′*)])) $\longleftrightarrow$
blits-in-$\mathcal{L}_{in}$ (*x1a*, *x1aa*, *D*, *x1c*, *x1d*, *x1e*, *x2e*)›
⟨*proof*⟩

lemma *literals-are-*$\mathcal{L}_{in}$*-set-conflict-wl*:
‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ (*set-conflict-wl D S*) $\longleftrightarrow$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
⟨*proof*⟩

lemma *blits-in-*$\mathcal{L}_{in}$*-keep-watch′*:
assumes *K′*: ‹*K′* ∈# $\mathcal{L}_{all}$ (*all-atms-st* (*a*, *b*, *c*, *d*, *e*, *f*, *g*))› **and**
w:‹*blits-in-*$\mathcal{L}_{in}$ (*a*, *b*, *c*, *d*, *e*, *f*, *g*)›
shows ‹*blits-in-*$\mathcal{L}_{in}$ (*a*, *b*, *c*, *d*, *e*, *f*, *g* (*K* := (*g K*)[*j* := (*i*, *K′*, *b′*)]))›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$› **and**
$\mathcal{A}_{in}$: ‹*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*›
shows ‹*unit-propagation-inner-loop-body-wl-D K j w S* ≤
⇓ {((*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*)›
⟨*proof*⟩

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*) ∈
[λ(((*K*, *j*), *w*), *S*). *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S* ∧ *K* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$]$_f$
Id $\times_r$ *Id* $\times_r$ *Id* $\times_r$ *Id* → ⟨*nat-rel* $\times_r$ *nat-rel* $\times_r$
{(*T′*, *T*). *T* = *T′* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}⟩ *nres-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*) ∈
[λ(((*K*, *j*), *w*), *S*). *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S* ∧ *K* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$]$_f$
Id $\times_r$ *Id* $\times_r$ *Id* $\times_r$ *Id* → ⟨*nat-rel* $\times_r$ *nat-rel* $\times_r$ *Id*⟩ *nres-rel*›
(**is** ‹*?G2*›)

⟨*proof*⟩

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* ∈# $\mathcal{L}_{all}$ (*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* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
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*⟩

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* ∈# $\mathcal{L}_{all}$ (*all-atms-st* S_0) ∧
L ∈# $\mathcal{L}_{all}$ (*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* ∈# $\mathcal{L}_{all}$ $\mathcal{A}$›
shows ‹*unit-propagation-inner-loop-wl-D K S* ≤
⇓ {(*T'*, *T*). *T* = *T'* ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
(*unit-propagation-inner-loop-wl K S*)›
⟨*proof*⟩

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* ∈# $\mathcal{L}_{all}$ (*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*) ⟷ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ S*›
⟨*proof*⟩

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* ≤
⇓ {(*T′*, *T*). *T* = *T′* ∧ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ T*}
(*unit-propagation-outer-loop-wl S*)›
⟨*proof*⟩

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-$\mathcal{L}_{in}$ $\mathcal{A}$ T*} $\rightarrow_f$
⟨{(*T′*, *T*). *T* = *T′* ∧ *literals-are-$\mathcal{L}_{in}$ $\mathcal{A}$ T*}⟩*nres-rel*›
⟨*proof*⟩

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* ∉# *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*›
⟨*proof*⟩

lemma *get-clauses-wl-tl-state*: ‹*get-clauses-wl* (*tl-state-wl T*) = *get-clauses-wl T*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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* ≤
⇓ {(*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 S*)›
(**is** ‹- ≤ ⇓ *?R* -›)
⟨*proof*⟩

definition *find-lit-of-max-level-wl'* :: ‹- ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒
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* ⇒ *nat literal* ⇒ *nat clause* ⇒ *nat clause-l nres*›
where
‹*list-of-mset2 L L' D* =
SPEC (λ*E*. *mset E* = *D* ∧ *E*!*0* = *L* ∧ *E*!*1* = *L'* ∧ *length E* ≥ *2*)›

definition *single-of-mset* **where**
‹*single-of-mset D* = *SPEC*(λ*L*. *D* = *mset* [*L*])›

definition *backtrack-wl-D-inv* **where**
‹*backtrack-wl-D-inv S* ⟷ *backtrack-wl-inv S* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*›

definition *propagate-bt-wl-D*
:: ‹*nat literal* ⇒ *nat literal* ⇒ *nat twl-st-wl* ⇒ *nat twl-st-wl nres*›
where
‹*propagate-bt-wl-D* = (λ*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*⟩

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 ∈# ℒall (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-of L' ∈ atms-of-mm (clause ‘# twl-clause-of ‘# init-clss-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);
```

$(S, L) \leftarrow$ *find-unassigned-lit-wl-D S*;
case L of
None $\Rightarrow$ *RETURN* (*True*, *S*)
| *Some L* $\Rightarrow$ *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*
$\leq \Downarrow \{((b', T'), b, T).\ b = b' \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T\}$ (*decide-wl-or-skip S*)›
⟨*proof*⟩

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 {
$T \leftarrow$ *skip-and-resolve-loop-wl-D S*;
ASSERT(*get-conflict-wl T* $\neq$ *None* $\wedge$ *get-clauses-wl S* = *get-clauses-wl T*);
$U \leftarrow$ *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* $\leq \Downarrow \{((b', T'), (b, T)).\ b = b' \wedge T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T\}$
(*cdcl-twl-o-prog-wl S*)›
⟨*proof*⟩

theorem *cdcl-twl-o-prog-wl-D-spec'*:
‹(*cdcl-twl-o-prog-wl-D*, *cdcl-twl-o-prog-wl*) $\in$
$\{(S,S').\ (S,S') \in Id\ \wedge$*literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $S\} \rightarrow_f$
⟨*bool-rel* $\times_r$ $\{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ $T\}$⟩ *nres-rel*›
⟨*proof*⟩

Full Strategy

definition *cdcl-twl-stgy-prog-wl-D*
:: ‹*nat twl-st-wl* $\Rightarrow$ *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-are-*$\mathcal{L}_{in}$ (*all-atms-st T*) *T*
    (λ(*brk*, -). ¬*brk*)
    (λ(*brk*, *S*).
    do {
      T ← *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* ≤ ⇓ {(*T*′, *T*). *T* = *T*′ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}
    (*cdcl-twl-stgy-prog-wl S*)›
⟨*proof*⟩

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*} →$_f$
    ⟨{(*T*′, *T*). *T* = *T*′ ∧ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*}⟩ *nres-rel*›
  ⟨*proof*⟩

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* ≤ ⇓ (*state-wl-l None O twl-st-l None*) (*conclusive-TWL-run S*′)›
⟨*proof*⟩

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* $\leq \Downarrow \{(T', T).\ T = T' \wedge$ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *T*$\}$
(*cdcl-twl-stgy-prog-break-wl S*)›
⟨*proof*⟩

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*⟩

The definition is here to be shared later.

definition *get-propagation-reason* :: ‹(*'v*, *'mark*) *ann-lits* ⇒ *'v literal* ⇒ *'mark option nres*› **where**
‹*get-propagation-reason M L* = *SPEC*($\lambda C.\ C \neq$ *None* ⟶ *Propagated L* (*the C*) ∈ *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*›
⟨*proof*⟩

lemma *rtranclp-cdcl-twl-restart-get-all-init-clss*:
assumes ‹*cdcl-twl-restart*** *S T*›
shows ‹*get-all-init-clss T* = *get-all-init-clss S*›
⟨*proof*⟩

As we have a specialised version of *correct-watching*, we defined a special version for the inclusion of the domain:

definition *all-init-lits* :: ‹(*nat*, *'v literal list* × *bool*) *fmap* ⇒ *'v literal multiset multiset* ⇒
'v literal multiset› **where**
‹*all-init-lits S NUE* = *all-lits-of-mm* (($\lambda C.$ *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*›
⟨*proof*⟩

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* ($\mathcal{L}_{all}$ $\mathcal{B}$)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *literals-are-*$\mathcal{L}_{in}$*'-literals-are-*$\mathcal{L}_{in}$*-iff*:
assumes
Sx: ‹(*S*, *x*) ∈ *state-wl-l None*› **and**
x-xa: ‹(*x*, *xa*) ∈ *twl-st-l None*› **and**
struct-invs: ‹*twl-struct-invs xa*›
shows
‹*literals-are-*$\mathcal{L}_{in}$*'* $\mathcal{A}$ *S* ⟷ *literals-are-*$\mathcal{L}_{in}$ $\mathcal{A}$ *S*› (**is** *?A*)
‹*literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S* ⟷ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*› (**is** *?B*)
‹*set-mset* (*all-init-atms-st S*) = *set-mset* (*all-atms-st S*)› (**is** *?C*)
⟨*proof*⟩

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

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

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

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

⟨proof⟩

lemma *rtranclp-GC-remap-all-init-lits*:
‹$GC\text{-}remap^{**}$ (N, m, new) (N′, m′, new′) ⟹ all-init-lits N NE + all-init-lits new NE = all-init-lits N′ NE + all-init-lits new′ NE›
⟨proof⟩

lemma *cdcl-twl-restart-is-$\mathcal{L}_{all}$*:
assumes
ST: ‹$cdcl\text{-}twl\text{-}restart^{**}$ 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⟩

lemma *cdcl-twl-restart-is-$\mathcal{L}_{all}$′*:
assumes
ST: ‹$cdcl\text{-}twl\text{-}restart^{**}$ 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⟩

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 = (λL 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*)›
⟨*proof*⟩

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›
⟨*proof*⟩

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*} $\rightarrow_f$
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-$\mathcal{L}_{in}$' $\mathcal{A}$ S*}⟩*nres-rel*›
(**is** ‹- ∈ - $\rightarrow_f$ ⟨*?R*⟩*nres-rel*›)
⟨*proof*⟩

definition *remove-one-annot-true-clause-one-imp-wl-D-pre* **where**
‹*remove-one-annot-true-clause-one-imp-wl-D-pre i T* ⟷
remove-one-annot-true-clause-one-imp-wl-pre i T ∧
literals-are-$\mathcal{L}_{in}$' (*all-init-atms-st T*) *T*›

definition *remove-one-annot-true-clause-one-imp-wl-D*
:: ‹*nat* ⇒ *nat twl-st-wl* ⇒ (*nat* × *nat twl-st-wl*) *nres*›
where
‹*remove-one-annot-true-clause-one-imp-wl-D* = (λ*i S*. *do* {
ASSERT(*remove-one-annot-true-clause-one-imp-wl-D-pre i S*);
ASSERT(*is-proped* (*rev* (*get-trail-wl S*) ! *i*));
(*L*, *C*) ← *SPEC*(λ(*L*, *C*). (*rev* (*get-trail-wl S*))!*i* = *Propagated L C*);
ASSERT(*Propagated L C* ∈ *set* (*get-trail-wl S*));
ASSERT(*atm-of L* ∈# *all-init-atms-st S*);
if C = *0 then RETURN* (*i+1*, *S*)
else do {
ASSERT(*C* ∈# *dom-m* (*get-clauses-wl S*));
T ← *replace-annot-l L C S*;
ASSERT(*get-clauses-wl S* = *get-clauses-wl T*);
T ← *remove-and-add-cls-l C T*;
— *S* ← *remove-all-annot-true-clause-imp-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* ∈ *set* (*get-trail-wl S*)›
shows ‹*atm-of L* ∈# *all-init-atms-st S*›
⟨*proof*⟩

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*} $\rightarrow_f$
⟨*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* $\rightarrow_f$ -›)
⟨*proof*⟩

definition *remove-one-annot-true-clause-imp-wl-D-inv* **where**
‹*remove-one-annot-true-clause-imp-wl-D-inv S* = (λ(*i*, *T*).
remove-one-annot-true-clause-imp-wl-inv S (*i*, *T*) ∧
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* = (λ*S*. *do* {
k ← *SPEC*(λ*k*. (∃ *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*) ← $WHILE_T{}^{remove\text{-}one\text{-}annot\text{-}true\text{-}clause\text{-}imp\text{-}wl\text{-}D\text{-}inv\ S}$
(λ(*i*, *S*). *i* < *k*)
(λ(*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*⟩

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* $T \propto (xs!i)!0$) ($xs!i$) $\notin$ *set* (*get-trail-wl* T)) $\wedge$
$\neg$*irred* (*get-clauses-wl* T) ($xs!i$) $\wedge$ *length* (*get-clauses-wl* $T \propto (xs!i)$) $\neq$ *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*) $\in$
$\{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st* S) $S\} \to_f$
$\langle\{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st* S) $S\}\rangle$*nres-rel*›
⟨*proof*⟩

definition *mark-to-delete-clauses-wl-D-post* **where**
‹*mark-to-delete-clauses-wl-D-post S T* $\longleftrightarrow$
(*mark-to-delete-clauses-wl-post S T* $\wedge$ *literals-are-*$\mathcal{L}_{in}{}'$ (*all-init-atms-st S*) S)›

definition *cdcl-twl-full-restart-wl-prog-D* :: ‹*nat twl-st-wl* $\Rightarrow$ *nat twl-st-wl nres*› **where**
‹*cdcl-twl-full-restart-wl-prog-D S* = *do* {
— $S \leftarrow$ *remove-one-annot-true-clause-imp-wl-D S*;
ASSERT(*mark-to-delete-clauses-wl-D-pre S*);
$T \leftarrow$ *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) $\in \{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) $S\}$› **and**
‹*mark-to-delete-clauses-wl-D-pre S*› **and**
‹(T, Ta) $\in \{(S, T).\ (S, T) \in Id \wedge$ *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) $\in \{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) $S\}$›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*) $S\} \to_f$
$\langle\{(S, T).\ (S, T) \in Id \wedge$ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) $S\}\rangle$*nres-rel*›
⟨*proof*⟩

definition *restart-abs-wl-D-pre* :: ‹*nat twl-st-wl* $\Rightarrow$ *bool* $\Rightarrow$ *bool*› **where**
‹*restart-abs-wl-D-pre S brk* $\longleftrightarrow$

(*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*)
∈ [λ*S*. *restart-abs-wl-D-pre S False*]$_f$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} →
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*}⟩*nres-rel*›
⟨*proof*⟩

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*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*}›
shows ‹(*S*, *Sa*) ∈ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$*'* (*all-init-atms-st S*) *S*}›
⟨*proof*⟩

lemma *cdcl-twl-restart-wl-D-prog-cdcl-twl-restart-wl-prog*:
‹(*cdcl-twl-restart-wl-D-prog*, *cdcl-twl-restart-wl-prog*)
∈ [λ*S*. *restart-abs-wl-D-pre S False*]$_f$ {(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*} →
⟨{(*S*, *T*). (*S*, *T*) ∈ *Id* ∧ *literals-are-*$\mathcal{L}_{in}$ (*all-atms-st S*) *S*}⟩*nres-rel*›
⟨*proof*⟩

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* $T \propto (xs!i)$));
ASSERT(*get-clauses-wl* $T \propto (xs!i)!0 \in\# \mathcal{L}_{all}$ (*all-init-atms-st* *T*));
can-del ← *SPEC*($\lambda b.\ b \longrightarrow$
(*Propagated* (*get-clauses-wl* $T \propto (xs!i)!0$) (*xs*!*i*) ∉ *set* (*get-trail-wl* *T*)) ∧
¬*irred* (*get-clauses-wl* *T*) (*xs*!*i*) ∧ *length* (*get-clauses-wl* $T \propto (xs!i)$) ≠ *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*⟩

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* ($N \propto C$, *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* $\mathcal{A}$ *W* ≡ ⋃(((‘) *fst*) ‘ *set* ‘ *W* ‘ $\mathcal{A}$)›

lemma *clauses-pointed-to-insert*[*simp*]:
‹*clauses-pointed-to* (*insert* *A* $\mathcal{A}$) *W* =
fst ‘ *set* (*W* *A*) ∪
clauses-pointed-to $\mathcal{A}$ *W*› **and**
clauses-pointed-to-empty[*simp*]:
‹*clauses-pointed-to* {} *W* = {}›
⟨*proof*⟩

lemma *cdcl-GC-clauses-prog-copy-wl-entry*:
fixes *A* :: ‹′*v* *literal*› **and** *WS* :: ‹′*v* *literal* ⇒ ′*v* *watched*›

defines [*simp*]: ‹$W \equiv WS\ A$›
assumes ‹
$ran\ m0 \subseteq set\text{-}mset\ (dom\text{-}m\ N0') \wedge$
$(\forall L \in dom\ m0.\ L \notin\#\ (dom\text{-}m\ N0)) \wedge$
$set\text{-}mset\ (dom\text{-}m\ N0) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ \mathcal{A})\ WS \wedge$
$0 \notin\#\ dom\text{-}m\ N0'$›
shows
‹$cdcl\text{-}GC\text{-}clauses\text{-}prog\text{-}copy\text{-}wl\text{-}entry\ N0\ W\ A\ N0' \le$
$SPEC(\lambda(N, N').\ (\exists m.\ GC\text{-}remap^{**}\ (N0, m0, N0')\ (N, m, N') \wedge$
$ran\ m \subseteq set\text{-}mset\ (dom\text{-}m\ N') \wedge$
$(\forall L \in dom\ m.\ L \notin\#\ (dom\text{-}m\ N)) \wedge$
$set\text{-}mset\ (dom\text{-}m\ N) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (remove1\text{-}mset\ A\ \mathcal{A}))\ WS) \wedge$
$(\forall L \in set\ W.\ fst\ L \notin\#\ dom\text{-}m\ N) \wedge$
$0 \notin\#\ dom\text{-}m\ N'))$›
⟨*proof*⟩

definition *cdcl-GC-clauses-prog-single-wl*
:: ‹$'v\ clauses\text{-}l \Rightarrow ('v\ literal \Rightarrow 'v\ watched) \Rightarrow 'v \Rightarrow$
$'v\ clauses\text{-}l \Rightarrow ('v\ clauses\text{-}l \times 'v\ clauses\text{-}l \times ('v\ literal \Rightarrow 'v\ watched))\ nres$›
where
‹$cdcl\text{-}GC\text{-}clauses\text{-}prog\text{-}single\text{-}wl = (\lambda N\ WS\ A\ N'.\ do\ \{$
$L \leftarrow RES\ \{Pos\ A, Neg\ A\};$
$(N, N') \leftarrow cdcl\text{-}GC\text{-}clauses\text{-}prog\text{-}copy\text{-}wl\text{-}entry\ N\ (WS\ L)\ L\ N';$
$let\ WS = WS(L := []);$
$(N, N') \leftarrow cdcl\text{-}GC\text{-}clauses\text{-}prog\text{-}copy\text{-}wl\text{-}entry\ N\ (WS\ (-L))\ (-L)\ N';$
$let\ WS = WS(-L := []);$
$RETURN\ (N, N', WS)$
$\})$›

lemma *clauses-pointed-to-remove1-if*:
‹$\forall L \in set\ (W\ L).\ fst\ L \notin\#\ dom\text{-}m\ aa \Longrightarrow xa \in\#\ dom\text{-}m\ aa \Longrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (remove1\text{-}mset\ L\ \mathcal{A}))$
$(\lambda a.\ if\ a = L\ then\ []\ else\ W\ a) \longleftrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (remove1\text{-}mset\ L\ \mathcal{A}))\ W$›
⟨*proof*⟩

lemma *clauses-pointed-to-remove1-if2*:
‹$\forall L \in set\ (W\ L).\ fst\ L \notin\#\ dom\text{-}m\ aa \Longrightarrow xa \in\#\ dom\text{-}m\ aa \Longrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L, L'\#\}))$
$(\lambda a.\ if\ a = L\ then\ []\ else\ W\ a) \longleftrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L, L'\#\}))\ W$›
‹$\forall L \in set\ (W\ L).\ fst\ L \notin\#\ dom\text{-}m\ aa \Longrightarrow xa \in\#\ dom\text{-}m\ aa \Longrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L', L\#\}))$
$(\lambda a.\ if\ a = L\ then\ []\ else\ W\ a) \longleftrightarrow$
$xa \in clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L', L\#\}))\ W$›
⟨*proof*⟩

lemma *clauses-pointed-to-remove1-if2-eq*:
‹$\forall L \in set\ (W\ L).\ fst\ L \notin\#\ dom\text{-}m\ aa \Longrightarrow$
$set\text{-}mset\ (dom\text{-}m\ aa) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L, L'\#\}))$
$(\lambda a.\ if\ a = L\ then\ []\ else\ W\ a) \longleftrightarrow$
$set\text{-}mset\ (dom\text{-}m\ aa) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L, L'\#\}))\ W$›
‹$\forall L \in set\ (W\ L).\ fst\ L \notin\#\ dom\text{-}m\ aa \Longrightarrow$
$set\text{-}mset\ (dom\text{-}m\ aa) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L', L\#\}))$
$(\lambda a.\ if\ a = L\ then\ []\ else\ W\ a) \longleftrightarrow$
$set\text{-}mset\ (dom\text{-}m\ aa) \subseteq clauses\text{-}pointed\text{-}to\ (set\text{-}mset\ (\mathcal{A} - \{\#L', L\#\}))\ W$›

⟨proof⟩

lemma *negs-remove-Neg*: ‹$A \notin\# \mathcal{A} \Longrightarrow$ *negs* $\mathcal{A}$ + *poss* $\mathcal{A}$ − {#*Neg A*, *Pos A*#} =
negs $\mathcal{A}$ + *poss* $\mathcal{A}$›
⟨proof⟩
lemma *poss-remove-Pos*: ‹$A \notin\# \mathcal{A} \Longrightarrow$ *negs* $\mathcal{A}$ + *poss* $\mathcal{A}$ − {#*Pos A*, *Neg A*#} =
negs $\mathcal{A}$ + *poss* $\mathcal{A}$›
⟨proof⟩

lemma *cdcl-GC-clauses-prog-single-wl-removed*:
‹∀ *L*∈*set* (*W* (*Pos A*)). *fst L* ∉# *dom-m aaa* ⟹
∀ *L*∈*set* (*W* (*Neg A*)). *fst L* ∉# *dom-m a* ⟹
*GC-remap*** (*aaa*, *ma*, *baa*) (*a*, *mb*, *b*) ⟹
set-mset (*dom-m a*) ⊆ *clauses-pointed-to* (*set-mset* (*negs* $\mathcal{A}$ + *poss* $\mathcal{A}$ − {#*Neg A*, *Pos A*#})) *W*
⟹
xa ∈# *dom-m a* ⟹
xa ∈ *clauses-pointed-to* (*Neg* ‘ *set-mset* (*remove1-mset A* $\mathcal{A}$) ∪ *Pos* ‘ *set-mset* (*remove1-mset A*
$\mathcal{A}$))
(*W*(*Pos A* := [], *Neg A* := []))›
‹∀ *L*∈*set* (*W* (*Neg A*)). *fst L* ∉# *dom-m aaa* ⟹
∀ *L*∈*set* (*W* (*Pos A*)). *fst L* ∉# *dom-m a* ⟹
*GC-remap*** (*aaa*, *ma*, *baa*) (*a*, *mb*, *b*) ⟹
set-mset (*dom-m a*) ⊆ *clauses-pointed-to* (*set-mset* (*negs* $\mathcal{A}$ + *poss* $\mathcal{A}$ − {#*Pos A*, *Neg A*#})) *W*
⟹
xa ∈# *dom-m a* ⟹
xa ∈ *clauses-pointed-to*
(*Neg* ‘ *set-mset* (*remove1-mset A* $\mathcal{A}$) ∪ *Pos* ‘ *set-mset* (*remove1-mset A* $\mathcal{A}$))
(*W*(*Neg A* := [], *Pos A* := []))›
⟨proof⟩

lemma *cdcl-GC-clauses-prog-single-wl*:
fixes *A* :: ‹′*v*› **and** *WS* :: ‹′*v literal* ⇒ ′*v watched*› **and**
N0 :: ‹′*v clauses-l*›
assumes ‹*ran m* ⊆ *set-mset* (*dom-m N0*′) ∧
(∀ *L*∈*dom m*. *L* ∉# (*dom-m N0*)) ∧
set-mset (*dom-m N0*) ⊆
clauses-pointed-to (*set-mset* (*negs* $\mathcal{A}$ + *poss* $\mathcal{A}$)) *W* ∧
0 ∉# *dom-m N0*′›
shows
‹*cdcl-GC-clauses-prog-single-wl N0 W A N0*′ ≤
SPEC(λ(*N*, *N*′, *WS*′). ∃ *m*′. *GC-remap*** (*N0*, *m*, *N0*′) (*N*, *m*′, *N*′) ∧
ran m′ ⊆ *set-mset* (*dom-m N*′) ∧
(∀ *L*∈*dom m*′. *L* ∉# *dom-m N*) ∧
WS′ (*Pos A*) = [] ∧ *WS*′ (*Neg A*) = [] ∧
(∀ *L*. *L* ≠ *Pos A* ⟶ *L* ≠ *Neg A* ⟶ *W L* = *WS*′ *L*) ∧
set-mset (*dom-m N*) ⊆
clauses-pointed-to
(*set-mset* (*negs* (*remove1-mset A* $\mathcal{A}$) + *poss* (*remove1-mset A* $\mathcal{A}$))) *WS*′ ∧
0 ∉# *dom-m N*′
)›
⟨proof⟩

definition *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*)) ← $WHILE_T{}^{cdcl\text{-}GC\text{-}clauses\text{-}prog\text{-}wl\text{-}inv\ \mathcal{A}\ N0}$
(λ($\mathcal{B}$, -). $\mathcal{B}$ ≠ {#})
(λ($\mathcal{B}$, (*N*, *N'*, *WS*)). *do* {
ASSERT($\mathcal{B}$ ≠ {#});
A ← *SPEC* (λ*A*. *A* ∈# $\mathcal{B}$);
(*N*, *N'*, *WS*) ← *cdcl-GC-clauses-prog-single-wl N WS A N'*;
RETURN (*remove1-mset A* $\mathcal{B}$, (*N*, *N'*, *WS*))
})
($\mathcal{A}$, (*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*⟩

lemma *all-init-atms-fmdrop-add-mset-unit*:
‹*C* ∈# *dom-m baa* ⟹ *irred baa C* ⟹
all-init-atms (*fmdrop C baa*) (*add-mset* (*mset* (*baa* ∝ *C*)) *da*) =
all-init-atms baa da›
‹*C* ∈# *dom-m baa* ⟹ ¬*irred baa C* ⟹
all-init-atms (*fmdrop C baa*) *da* =
all-init-atms baa da›
⟨*proof*⟩

lemma *cdcl-GC-clauses-prog-wl2*:
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*› **and**

⟨N0 = N0'⟩
shows
⟨cdcl-GC-clauses-prog-wl (M, N0, D, NE, UE, Q, WS) ≤
⇓ {((M', N'', D', NE', UE', Q', WS'), (N, N')). (M', D', NE', UE', Q') = (M, D, NE, UE, Q) ∧
N'' = N ∧ (∀ L∈#all-init-lits N0 NE. WS' L = [])∧
all-init-lits N0 NE = all-init-lits N NE' ∧
(∃ m. GC-remap** (N0, (λ-. None), fmempty) (fmempty, m, N))}
(SPEC(λ(N'::(nat, 'a literal list × bool) fmap, m).
GC-remap** (N0', (λ-. None), fmempty) (fmempty, m, N') ∧
0 ∉# dom-m N'))⟩
⟨proof⟩

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(λQ. 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} →$_f$ ⟨{(S::nat twl-st-wl, S').
(S, S') ∈ Id ∧ literals-are-$\mathcal{L}_{in}$' (all-init-atms-st S) S}⟩nres-rel⟩
⟨proof⟩

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;

$V \leftarrow$ *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*)›
⟨*proof*⟩

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

lemma *all-lits-alt-def*:
‹*all-lits S NUE* = *all-lits-of-mm* (*mset* ‘# *ran-mf S* + *NUE*)›
⟨*proof*⟩

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*⟩

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⟩

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⟩

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*›
⟨proof⟩

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*›
⟨proof⟩

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) ∈ *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*›
⟨proof⟩

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) ← $WHILE_T{}^{\lambda(-,\ brk,\ T,\ n).\ cdcl\text{-}twl\text{-}stgy\text{-}restart\text{-}abs\text{-}wl\text{-}D\text{-}inv\ S_0\ brk\ T\ n}$
(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*›
⟨*proof*⟩

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*›

⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*twl-st-init*]:
‹*clauses-to-update-init* (*add-to-unit-init-clauses* (*mset* *C*) *T*) = *clauses-to-update-init* *T*›
‹*literals-to-update-init* (*add-to-unit-init-clauses* (*mset* *C*) *T*) = *literals-to-update-init* *T*›
‹*get-conflict-init* (*add-to-unit-init-clauses* (*mset* *C*) *T*) = *get-conflict-init* *T*›
⟨*proof*⟩
lemma [*twl-st-init*]:
‹*twl-st-inv* (*fst* (*add-to-unit-init-clauses* (*mset* *C*) *T*)) ⟷ *twl-st-inv* (*fst* *T*)›

‹valid-enqueued (fst (add-to-unit-init-clauses (mset C) T)) ⟷ valid-enqueued (fst T)›
‹no-duplicate-queued (fst (add-to-unit-init-clauses (mset C) T)) ⟷ no-duplicate-queued (fst T)›
‹distinct-queued (fst (add-to-unit-init-clauses (mset C) T)) ⟷ distinct-queued (fst T)›
‹confl-cands-enqueued (fst (add-to-unit-init-clauses (mset C) T)) ⟷ confl-cands-enqueued (fst T)›
‹propa-cands-enqueued (fst (add-to-unit-init-clauses (mset C) T)) ⟷ propa-cands-enqueued (fst T)›
‹twl-st-exception-inv (fst (add-to-unit-init-clauses (mset C) T)) ⟷ twl-st-exception-inv (fst T)›
⟨proof⟩

lemma [twl-st-init]:
‹trail (state$_W$-of-init T) = get-trail-init T›
‹get-trail (fst T) = get-trail-init (T)›
‹conflicting (state$_W$-of-init T) = get-conflict-init T›
‹init-clss (state$_W$-of-init T) = clauses (get-init-clauses-init T) + get-unit-init-clauses-init T
+ 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›
⟨proof⟩

definition twl-st-l-init :: ‹('v twl-st-l-init × 'v twl-st-init) set› **where**
‹twl-st-l-init = {(((M, N, C, NE, UE, 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›
⟨proof⟩

lemma [twl-st-init]:
assumes ‹(S, T) ∈ twl-st-l-init›
shows
‹get-conflict-init T = get-conflict-l-init S›
‹get-conflict (fst T) = get-conflict-l-init S›
‹literals-to-update-init T = literals-to-update-l-init S›
‹clauses-to-update-init T = {#}›
‹other-clauses-init T = other-clauses-l-init S›
‹lits-of-l (get-trail-init T) = lits-of-l (get-trail-l-init S)›
‹lit-of '# mset (get-trail-init T) = lit-of '# mset (get-trail-l-init S)›
⟨proof⟩

definition twl-struct-invs-init :: ‹'v twl-st-init ⇒ bool› **where**
‹twl-struct-invs-init S ⟷
(twl-st-inv (fst S) ∧
valid-enqueued (fst S) ∧
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*›
⟨*proof*⟩

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

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*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

definition *init-dt-spec* **where**
‹*init-dt-spec CS SOC SOC'* ⟷
(∃ *T'*. (*SOC'*, *T'*) ∈ *twl-st-l-init* ∧
twl-struct-invs-init T' ∧
clauses-to-update-l-init SOC' = {#} ∧
(∀ *s*∈*set* (*get-trail-l-init SOC'*). ¬*is-decided s*) ∧
(*get-conflict-l-init SOC'* = *None* ⟶
literals-to-update-l-init SOC' = *uminus* '# *lit-of* '# *mset* (*get-trail-l-init SOC'*)) ∧
(*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*⟩

lemma *invariants-init-state*:
assumes
lev: ‹*count-decided (get-trail-init T) = 0*› **and**
wf: ‹*∀ C ∈# get-clauses (fst T). struct-wf-twl-cls C*› **and**
MQ: ‹*literals-to-update-init T = uminus ‘# lit-of ‘# mset (get-trail-init T)*› **and**
WS: ‹*clauses-to-update-init T = {#}*› **and**
n-d: ‹*no-dup (get-trail-init T)*›
shows ‹*propa-cands-enqueued (fst T)*› **and** ‹*confl-cands-enqueued (fst T)*› **and** ‹*twl-st-inv (fst T)*›
‹*clauses-to-update-inv (fst T)*› **and** ‹*past-invs (fst T)*› **and** ‹*distinct-queued (fst T)*› **and**
‹*valid-enqueued (fst T)*› **and** ‹*twl-st-exception-inv (fst T)*› **and** ‹*no-duplicate-queued (fst T)*›
⟨*proof*⟩

lemma *twl-struct-invs-init-init-state*:
assumes
lev: ‹*count-decided (get-trail-init T) = 0*› **and**
wf: ‹*∀ C ∈# get-clauses (fst T). struct-wf-twl-cls C*› **and**
MQ: ‹*literals-to-update-init T = uminus ‘# lit-of ‘# mset (get-trail-init T)*› **and**
WS: ‹*clauses-to-update-init T = {#}*› **and**
struct-invs: ‹$cdcl_W$*-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*⟩

lemma *twl-struct-invs-init-add-to-unit-init-clauses*:
assumes
dist: ‹*distinct a*› **and**
lev: ‹*count-decided (get-trail (fst T)) = 0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
ex: ‹*∃ L ∈ set a. L ∈ lits-of-l (get-trail-init T)*›
shows
‹*twl-struct-invs-init (add-to-unit-init-clauses (mset a) T)*›
(**is** *?all-struct*)
⟨*proof*⟩

lemma *twl-struct-invs-init-set-conflict-init*:
assumes
dist: ‹*distinct C*› **and**
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
ex: ‹∀ *L* ∈ *set C*. −*L* ∈ *lits-of-l* (*get-trail-init T*)› **and**
nempty: ‹*C* ≠ []›
shows
‹*twl-struct-invs-init* (*set-conflict-init C T*)›
(**is** *?all-struct*)
⟨*proof*⟩

lemma *twl-struct-invs-init-propagate-unit-init*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
undef: ‹*undefined-lit* (*get-trail-init T*) *L*› **and**
confl: ‹*get-conflict-init T* = *None*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}›
shows
‹*twl-struct-invs-init* (*propagate-unit-init L T*)›
(**is** *?all-struct*)
⟨*proof*⟩

named-theorems *twl-st-l-init*
lemma [*twl-st-l-init*]:
‹*clauses-to-update-l-init* (*already-propagated-unit-init-l C S*) = *clauses-to-update-l-init S*›
‹*get-trail-l-init* (*already-propagated-unit-init-l C S*) = *get-trail-l-init S*›
‹*get-conflict-l-init* (*already-propagated-unit-init-l C S*) = *get-conflict-l-init S*›
‹*other-clauses-l-init* (*already-propagated-unit-init-l C S*) = *other-clauses-l-init S*›
‹*clauses-to-update-l-init* (*already-propagated-unit-init-l C S*) = *clauses-to-update-l-init S*›
‹*literals-to-update-l-init* (*already-propagated-unit-init-l C S*) = *literals-to-update-l-init S*›
‹*get-clauses-l-init* (*already-propagated-unit-init-l C S*) = *get-clauses-l-init S*›
‹*get-unit-clauses-l-init* (*already-propagated-unit-init-l C S*) = *add-mset C* (*get-unit-clauses-l-init S*)›
‹*get-learned-unit-clauses-l-init* (*already-propagated-unit-init-l C S*) =
get-learned-unit-clauses-l-init S›
‹*get-conflict-l-init* (*T*, *OC*) = *get-conflict-l T*›
⟨*proof*⟩

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

lemma [*twl-st-l-init*]:
‹*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*›
⟨*proof*⟩

lemma *entailed-clss-inv-add-to-unit-init-clauses*:
‹*count-decided* (*get-trail-init T*) = *0* ⟹ *C* ≠ [] ⟹ *hd C* ∈ *lits-of-l* (*get-trail-init T*) ⟹
entailed-clss-inv (*fst T*) ⟹ *entailed-clss-inv* (*fst* (*add-to-unit-init-clauses* (*mset C*) *T*))›
⟨*proof*⟩

lemma *convert-lits-l-no-decision-iff*: ‹(*S*, *T*) ∈ *convert-lits-l M N* ⟹
(∀ *s*∈*set T*. ¬ *is-decided s*) ⟷
(∀ *s*∈*set S*. ¬ *is-decided s*)›
⟨*proof*⟩

lemma *twl-st-l-init-no-decision-iff*:
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹
(∀ *s*∈*set* (*get-trail-init T*). ¬ *is-decided s*) ⟷
(∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*)›
⟨*proof*⟩

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

lemma [*twl-st-l-init*]:
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹ *get-learned-clauses-init T* = {#} ⟷ *learned-clss-l* (*get-clauses-l-init S*) = {#}›
‹(*S*, *T*) ∈ *twl-st-l-init* ⟹ *get-unit-learned-clauses-init T* = {#} ⟷ *get-learned-unit-clauses-l-init S* = {#}
›
⟨*proof*⟩

lemma *init-dt-pre-already-propagated-unit-init-l*:
assumes
hd-C: ‹*hd C* ∈ *lits-of-l* (*get-trail-l-init S*)› **and**
pre: ‹*init-dt-pre CS S*› **and**
nempty: ‹*C* ≠ []› **and**
dist-C: ‹*distinct C*› **and**
lev: ‹*count-decided* (*get-trail-l-init S*) = *0*›
shows
‹*init-dt-pre CS* (*already-propagated-unit-init-l* (*mset C*) *S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [*C*] *S* (*already-propagated-unit-init-l* (*mset C*) *S*)› (**is** *?spec*)
⟨*proof*⟩

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

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*)›
⟨*proof*⟩

lemma *init-dt-pre-propagate-unit-init*:
assumes
hd-C: ‹*undefined-lit* (*get-trail-l-init S*) *L*› **and**
pre: ‹*init-dt-pre CS S*› **and**
lev: ‹*count-decided* (*get-trail-l-init S*) = *0*› **and**
confl: ‹*get-conflict-l-init S* = *None*›
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*⟩

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*›
⟨*proof*⟩

lemma *init-dt-pre-set-conflict-init-l*:
assumes
[*simp*]: ‹*get-conflict-l-init S* = *None*› **and**
pre: ‹*init-dt-pre* (*C* # *CS*) *S*› **and**
false: ‹∀ *L* ∈ *set C*. −*L* ∈ *lits-of-l* (*get-trail-l-init S*)› **and**
nempty: ‹*C* ≠ []›
shows
‹*init-dt-pre CS* (*set-conflict-init-l C S*)› (**is** *?pre*) **and**
‹*init-dt-spec* [*C*] *S* (*set-conflict-init-l C S*)› (**is** *?spec*)
⟨*proof*⟩

lemma [*twl-st-init*]:
‹*get-trail-init* (*add-empty-conflict-init T*) = *get-trail-init T*›
‹*get-conflict-init* (*add-empty-conflict-init T*) = *Some* {#}›
‹ *clauses-to-update-init* (*add-empty-conflict-init T*) = *clauses-to-update-init T*›
‹*literals-to-update-init* (*add-empty-conflict-init T*) = {#}›
⟨*proof*⟩

lemma [*twl-st-l-init*]:
‹*get-trail-l-init* (*add-empty-conflict-init-l T*) = *get-trail-l-init T*›
‹*get-conflict-l-init* (*add-empty-conflict-init-l T*) = *Some* {#}›
‹*clauses-to-update-l-init* (*add-empty-conflict-init-l T*) = *clauses-to-update-l-init T*›
‹*literals-to-update-l-init* (*add-empty-conflict-init-l T*) = {#}›
‹*get-unit-clauses-l-init* (*add-empty-conflict-init-l T*) = *get-unit-clauses-l-init T*›
‹*get-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*)›
⟨*proof*⟩

lemma *twl-struct-invs-init-add-empty-conflict-init-l*:
assumes
lev: ‹*count-decided* (*get-trail* (*fst T*)) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
WS: ‹*clauses-to-update-init T* = {#}›

shows ‹*twl-struct-invs-init* (*add-empty-conflict-init T*)›
(**is** *?all-struct*)
⟨*proof*⟩

lemma *init-dt-pre-add-empty-conflict-init-l*:
assumes
confl[*simp*]: ‹*get-conflict-l-init S* = *None*› **and**
pre: ‹*init-dt-pre* ([] # *CS*) *S*›
shows
‹*init-dt-pre CS* (*add-empty-conflict-init-l S*)› (**is** *?pre*)
‹*init-dt-spec* [[]] *S* (*add-empty-conflict-init-l S*)› (**is** *?spec*)
⟨*proof*⟩

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

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

lemma *twl-struct-invs-init-add-to-clauses-init*:
assumes
lev: ‹*count-decided* (*get-trail-init T*) = *0*› **and**
invs: ‹*twl-struct-invs-init T*› **and**
confl: ‹*get-conflict-init T* = *None*› **and**
MQ: ‹*literals-to-update-init T* = *uminus* ‘# *lit-of* ‘# *mset* (*get-trail-init T*)› **and**
WS: ‹*clauses-to-update-init T* = {#}› **and**
dist-C: ‹*distinct C*› **and**
le-2: ‹*length C* ≥ *2*›
shows
‹*twl-struct-invs-init* (*add-to-clauses-init C T*)›
(**is** *?all-struct*)
⟨*proof*⟩

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

lemma *init-dt-pre-add-to-clauses-init-l*:
assumes
D: ‹*get-conflict-l-init S* = *None*› **and**
a: ‹*length a* ≠ *Suc 0*› ‹*a* ≠ []› **and**
pre: ‹*init-dt-pre* (*a* # *CS*) *S*› **and**
‹∀ *s*∈*set* (*get-trail-l-init S*). ¬ *is-decided s*›
shows
‹*add-to-clauses-init-l a S* ≤ *SPEC* (*init-dt-pre CS*)› (**is** *?pre*) **and**
‹*add-to-clauses-init-l a S* ≤ *SPEC* (*init-dt-spec* [*a*] *S*)› (**is** *?spec*)
⟨*proof*⟩

lemma *init-dt-pre-init-dt-step*:
assumes *pre*: ‹*init-dt-pre* (*a* # *CS*) *SOC*›
shows ‹*init-dt-step a SOC* ≤ *SPEC* (λ*SOC'*. *init-dt-pre CS SOC'* ∧ *init-dt-spec* [*a*] *SOC SOC'*)›
⟨*proof*⟩

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

lemma *init-dt-spec-append*:
assumes
spec1: ‹*init-dt-spec CS S T*› **and**
spec: ‹*init-dt-spec CS′ T U*›
shows ‹*init-dt-spec* (*CS* @ *CS′*) *S U*›
⟨*proof*⟩

lemma *init-dt-full*:
fixes *CS* :: ‹′*v literal list list*› **and** *SOC* :: ‹′*v twl-st-l-init*› **and** *S′*
defines
‹*S* ≡ *fst SOC*› **and**
‹*OC* ≡ *snd SOC*›
assumes
‹*init-dt-pre CS SOC*›
shows
‹*init-dt CS SOC* ≤ *SPEC* (*init-dt-spec CS SOC*)›
⟨*proof*⟩

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

lemma *twl-init-invs*:
‹*twl-struct-invs-init* (([], {#}, {#}, *None*, {#}, {#}, {#}, {#}), {#})›
‹*twl-list-invs* ([], *fmempty*, *None*, {#}, {#}, {#}, {#})›
‹*twl-stgy-invs* ([], {#}, {#}, *None*, {#}, {#}, {#}, {#})›
⟨*proof*⟩
end
theory *Watched-Literals-Watch-List-Initialisation*
imports *Watched-Literals-Watch-List Watched-Literals-Initialisation*
begin

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*)›
⟨*proof*⟩

lemma *in-clause-to-update-in-dom-mD*:
‹*bb* ∈# *clause-to-update L* (*a*, *aa*, *ab*, *ac*, *ad*, {#}, {#}) ⟹ *bb* ∈# *dom-m aa*›
⟨*proof*⟩

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*⟩

lemma *init-dt-wl-init-dt*:
assumes *S-S'*: ‹(*S*, *S'*) ∈ *state-wl-l-init*› **and**
dist: ‹∀ *C*∈*set C*. *distinct C*›
shows ‹*init-dt-wl C S* ≤ ⇓ *state-wl-l-init*
(*init-dt C S'*)›
⟨*proof*⟩

definition *init-dt-wl-pre* **where**
‹*init-dt-wl-pre C S* ⟷
(∃ *S'*. (*S*, *S'*) ∈ *state-wl-l-init* ∧
init-dt-pre C S')›

definition *init-dt-wl-spec* **where**
‹*init-dt-wl-spec C S T* ⟷
(∃ *S′ T′*. (*S*, *S′*) ∈ *state-wl-l-init* ∧ (*T*, *T′*) ∈ *state-wl-l-init* ∧
init-dt-spec C S′ T′)›

lemma *init-dt-wl-init-dt-wl-spec*:
assumes ‹*init-dt-wl-pre CS S*›
shows ‹*init-dt-wl CS S* ≤ *SPEC* (*init-dt-wl-spec CS S*)›
⟨*proof*⟩

fun *correct-watching-init* :: ‹′*v twl-st-wl* ⇒ *bool*› **where**
[*simp del*]: ‹*correct-watching-init* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ⟷
all-blits-are-in-problem-init (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) ∧
(∀ *L*.
distinct-watched (*W L*) ∧
(∀ (*i*, *K*, *b*)∈#*mset* (*W L*). *i* ∈# *dom-m N* ∧ *K* ∈ *set* (*N* ∝ *i*) ∧ *K* ≠ *L* ∧
correctly-marked-as-binary N (*i*, *K*, *b*)) ∧
fst ‘# *mset* (*W L*) = *clause-to-update L* (*M*, *N*, *D*, *NE*, *UE*, {#}, {#}))›

lemma *correct-watching-init-correct-watching*:
‹*correct-watching-init T* ⟹ *correct-watching T*›
⟨*proof*⟩

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

lemma *correct-watching-init-add-unit*:
assumes ‹*correct-watching-init* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*)›
shows ‹*correct-watching-init* (*M*, *N*, *D*, *add-mset C NE*, *UE*, *Q*, *W*)›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *correct-watching-init-cons*[*simp*]:
‹*NO-MATCH None y* ⟹ *correct-watching-init* ((*a*, *aa*, *y*, *ac*, *ad*, *ae*, *b*)) ⟷

correct-watching-init ((*a*, *aa*, *None*, *ac*, *ad*, *ae*, *b*))›
‹*NO-MATCH* {#} *ae* ⟹ *correct-watching-init* ((*a*, *aa*, *y*, *ac*, *ad*, *ae*, *b*)) ⟷
correct-watching-init ((*a*, *aa*, *y*, *ac*, *ad*, {#}, *b*))›
⟨*proof*⟩

lemma *clause-to-update-mapsto-upd-notin*:
assumes
i: ‹*i* ∉# *dom-m N*›
shows
‹*clause-to-update L* (*M*, *N*(*i* ↪ *C'*), *C*, *NE*, *UE*, *WS*, *Q*) =
(*if L* ∈ *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* ∈ *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*)))›
⟨*proof*⟩

lemma *correct-watching-init-add-clause*:
assumes
corr: ‹*correct-watching-init* ((*a*, *aa*, *None*, *ac*, *ad*, *Q*, *b*))› **and**
leC: ‹*2* ≤ *length C*› **and**
i-notin[*simp*]: ‹*i* ∉# *dom-m aa*› **and**
dist[*iff*]: ‹*C* ! *0* ≠ *C* ! *Suc 0*›
shows ‹*correct-watching-init*
((*a*, *fmupd i* (*C*, *red*) *aa*, *None*, *ac*, *ad*, *Q*, *b*
(*C* ! *0* := *b* (*C* ! *0*) @ [(*i*, *C* ! *Suc 0*, *length C* = *2*)],
C ! *Suc 0* := *b* (*C* ! *Suc 0*) @ [(*i*, *C* ! *0*, *length C* = *2*)])))›
⟨*proof*⟩

definition *rewatch*
:: ‹*'v clauses-l* ⇒ (*'v literal* ⇒ *'v watched*) ⇒ (*'v literal* ⇒ *'v watched*) *nres*›
where
‹*rewatch N W* = *do* {
xs ← *SPEC*(λ*xs*. *set-mset* (*dom-m N*) ⊆ *set xs* ∧ *distinct xs*);
nfoldli
xs
(λ-. *True*)
(λ*i W*. *do* {
if i ∈# *dom-m N*
then do {
ASSERT(*i* ∈# *dom-m N*);
ASSERT(*length* (*N* ∝ *i*) ≥ *2*);
let L1 = *N* ∝ *i* ! *0*;
let L2 = *N* ∝ *i* ! *1*;
let 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*]: ‹$W = (\lambda\text{-}.\ [])$› **and**
$H[dest]$: ‹$\bigwedge x.\ x \in\#\ dom\text{-}m\ N \Longrightarrow distinct\ (N \propto x) \wedge length\ (N \propto x) \geq 2$›
shows
‹$rewatch\ N\ W \leq SPEC(\lambda W.\ correct\text{-}watching\text{-}init\ (M,\ N,\ C,\ NE,\ UE,\ Q,\ W))$›
⟨*proof*⟩

definition *state-wl-l-init-full* :: ‹$('v\ twl\text{-}st\text{-}wl\text{-}init\text{-}full \times 'v\ twl\text{-}st\text{-}l\text{-}init)\ set$› **where**
‹$state\text{-}wl\text{-}l\text{-}init\text{-}full = \{(S,\ S').\ (fst\ S,\ fst\ S') \in state\text{-}wl\text{-}l\ None \wedge$
$snd\ S = snd\ S'\}$›

definition *added-only-watched* :: ‹$('v\ twl\text{-}st\text{-}wl\text{-}init\text{-}full \times 'v\ twl\text{-}st\text{-}wl\text{-}init)\ set$› **where**
‹$added\text{-}only\text{-}watched = \{(((M,\ N,\ D,\ NE,\ UE,\ Q,\ W),\ OC),\ ((M',\ N',\ D',\ NE',\ UE',\ Q'),\ OC')).$
$(M,\ N,\ D,\ NE,\ UE,\ Q) = (M',\ N',\ D',\ NE',\ UE',\ Q') \wedge OC = OC'\}$›

definition *init-dt-wl-spec-full*
:: ‹$'v\ clause\text{-}l\ list \Rightarrow 'v\ twl\text{-}st\text{-}wl\text{-}init \Rightarrow 'v\ twl\text{-}st\text{-}wl\text{-}init\text{-}full \Rightarrow bool$›
where
‹$init\text{-}dt\text{-}wl\text{-}spec\text{-}full\ C\ S\ T'' \longleftrightarrow$
$(\exists S'\ T\ T'.\ (S,\ S') \in state\text{-}wl\text{-}l\text{-}init \wedge (T :: 'v\ twl\text{-}st\text{-}wl\text{-}init,\ T') \in state\text{-}wl\text{-}l\text{-}init \wedge$
$init\text{-}dt\text{-}spec\ C\ S'\ T' \wedge correct\text{-}watching\text{-}init\ (fst\ T'') \wedge (T'',\ T) \in added\text{-}only\text{-}watched)$›

definition *init-dt-wl-full* :: ‹$'v\ clause\text{-}l\ list \Rightarrow 'v\ twl\text{-}st\text{-}wl\text{-}init \Rightarrow 'v\ twl\text{-}st\text{-}wl\text{-}init\text{-}full\ nres$› **where**
‹$init\text{-}dt\text{-}wl\text{-}full\ CS\ S = do\{$
$((M,\ N,\ D,\ NE,\ UE,\ Q),\ OC) \leftarrow init\text{-}dt\text{-}wl\ CS\ S;$
$W \leftarrow rewatch\ N\ (\lambda\text{-}.\ []);$
$RETURN\ ((M,\ N,\ D,\ NE,\ UE,\ Q,\ W),\ OC)$
}›

lemma *init-dt-wl-spec-rewatch-pre*:
assumes ‹$init\text{-}dt\text{-}wl\text{-}spec\ CS\ S\ T$› **and** ‹$N = get\text{-}clauses\text{-}init\text{-}wl\ T$› **and** ‹$C \in\#\ dom\text{-}m\ N$›
shows ‹$distinct\ (N \propto C) \wedge length\ (N \propto C) \geq 2$›
⟨*proof*⟩

lemma *init-dt-wl-full-init-dt-wl-spec-full*:
assumes ‹$init\text{-}dt\text{-}wl\text{-}pre\ CS\ S$›
shows ‹$init\text{-}dt\text{-}wl\text{-}full\ CS\ S \leq SPEC\ (init\text{-}dt\text{-}wl\text{-}spec\text{-}full\ CS\ S)$›
⟨*proof*⟩

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* (λ*L*. *get-level M L* < *count-decided M*) (*the D*))›

definition *out-learned-confl* :: ‹(*nat*, *nat*) *ann-lits* ⇒ *nat clause option* ⇒ *out-learned* ⇒ *bool*› **where**
‹*out-learned-confl M D out* ⟷
out ≠ [] ∧ (*D* ≠ *None* ∧ *mset out* = *the D*)›

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

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

definition *index-in-trail* :: ‹('*v*, '*a*) *ann-lits* ⇒ '*v literal* ⇒ *nat*› **where**
‹*index-in-trail M L* = *index* (*map* (*atm-of o lit-of*) (*rev M*)) (*atm-of L*)›

lemma *Propagated-in-trail-entailed*:
assumes
invs: ‹*cdcl$_W$-restart-mset.cdcl$_W$-all-struct-inv* (*M*, *N*, *U*, *D*)› **and**
in-trail: ‹*Propagated L C* ∈ *set M*›

shows
‹$M \models as$ *CNot* (*remove1-mset L C*)› **and** ‹$L \in\# C$› **and** ‹$N + U \models pm\ C$› **and**
‹$K \in\#$ *remove1-mset L C* $\Longrightarrow$ *index-in-trail M K* $<$ *index-in-trail M L*› **and**
‹$\neg$*tautology C*› **and** ‹*distinct-mset C*›
⟨*proof*⟩

This predicate corresponds to one resolution step.

inductive *minimize-conflict-support* :: ‹(′v, ′v clause) ann-lits ⇒ ′v clause ⇒ ′v clause ⇒ bool›
for *M* **where**
resolve-propa:
‹*minimize-conflict-support M* (*add-mset* ($-L$) *C*) (*C* + *remove1-mset L E*)›
if ‹*Propagated L E* ∈ *set M*› |
remdups: ‹*minimize-conflict-support M* (*add-mset L C*) *C*›

lemma *index-in-trail-uminus*[*simp*]: ‹*index-in-trail M* ($-L$) = *index-in-trail M L*›
⟨*proof*⟩

This is the termination argument of the conflict minimisation: the multiset of the levels decreases (for the multiset ordering).

definition *minimize-conflict-support-mes* :: ‹(′v, ′v clause) ann-lits ⇒ ′v clause ⇒ nat multiset›
where
‹*minimize-conflict-support-mes M C* = *index-in-trail M* ‘# *C*›

context
fixes *M* :: ‹(′v, ′v clause) ann-lits› **and** *N U* :: ‹′v clauses› **and**
D :: ‹′v clause option›
assumes *invs*: ‹$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*M*, *N*, *U*, *D*)›
begin

private lemma
no-dup: ‹*no-dup M*› **and**
consistent: ‹*consistent-interp* (*lits-of-l M*)›
⟨*proof*⟩

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

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

lemma *minimize-conflict-support-mes*:
assumes ‹*minimize-conflict-support M C E*›
shows ‹*minimize-conflict-support-mes M E* $<$ *minimize-conflict-support-mes M C*›
⟨*proof*⟩

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

lemma *conflict-minimize-step*:
assumes
‹$NU \models p$ *add-mset* L C› **and**
‹$NU \models p$ *add-mset* $(-L)$ D› **and**
‹$\bigwedge K'.\ K' \in\# C \Longrightarrow NU \models p$ *add-mset* $(-K')$ D›
shows ‹$NU \models p$ D›
⟨*proof*⟩

This function filters the clause by the levels up the level of the given literal. This is the part the conflict clause that is considered when testing if the given literal is redundant.

definition *filter-to-poslev* **where**
‹*filter-to-poslev* M L D = *filter-mset* (λK. *index-in-trail* M K < *index-in-trail* M L) D›

lemma *filter-to-poslev-uminus*[*simp*]:
‹*filter-to-poslev* M $(-L)$ D = *filter-to-poslev* M L D›
⟨*proof*⟩

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

lemma *filter-to-poslev-mono*:
‹*index-in-trail* M $K' \leq$ *index-in-trail* M $L \Longrightarrow$
filter-to-poslev M K' $D \subseteq\#$ *filter-to-poslev* M L D›
⟨*proof*⟩

lemma *filter-to-poslev-mono-entailement*:
‹*index-in-trail* M $K' \leq$ *index-in-trail* M $L \Longrightarrow$
$NU \models p$ *filter-to-poslev* M K' $D \Longrightarrow NU \models p$ *filter-to-poslev* M L D›
⟨*proof*⟩

lemma *filter-to-poslev-mono-entailement-add-mset*:
‹*index-in-trail* M $K' \leq$ *index-in-trail* M $L \Longrightarrow$
$NU \models p$ *add-mset* J (*filter-to-poslev* M K' D) $\Longrightarrow NU \models p$ *add-mset* J (*filter-to-poslev* M L D)›
⟨*proof*⟩

lemma *conflict-minimize-intermediate-step*:
assumes
‹$NU \models p$ *add-mset* L C› **and**
K'-C: ‹$\bigwedge K'.\ K' \in\# C \Longrightarrow NU \models p$ *add-mset* $(-K')$ $D \lor K' \in\# D$›
shows ‹$NU \models p$ *add-mset* L D›
⟨*proof*⟩

lemma *conflict-minimize-intermediate-step-filter-to-poslev*:
assumes
lev-K-L: ‹$\bigwedge K'.\ K' \in\# C \Longrightarrow$ *index-in-trail* M K' < *index-in-trail* M L› **and**
NU-LC: ‹$NU \models p$ *add-mset* L C› **and**
K'-C: ‹$\bigwedge K'.\ K' \in\# C \Longrightarrow NU \models p$ *add-mset* $(-K')$ (*filter-to-poslev* M L D) $\lor$
$K' \in\#$ *filter-to-poslev* M L D›
shows ‹$NU \models p$ *add-mset* L (*filter-to-poslev* M L D)›
⟨*proof*⟩

datatype *minimize-status* = *SEEN-FAILED* | *SEEN-REMOVABLE* | *SEEN-UNKNOWN*

instance *minimize-status* :: *heap*
⟨*proof*⟩

instantiation *minimize-status* :: *default*
begin
 definition *default-minimize-status* **where**
  ‹*default-minimize-status* = *SEEN-UNKNOWN*›

instance ⟨*proof*⟩
end

type-synonym *'v conflict-min-analyse* = ‹(*'v literal* × *'v clause*) *list*›
type-synonym *'v conflict-min-cach* = ‹*'v* ⇒ *minimize-status*›

definition *get-literal-and-remove-of-analyse*
 :: ‹*'v conflict-min-analyse* ⇒ (*'v literal* × *'v conflict-min-analyse*) *nres*› **where**
 ‹*get-literal-and-remove-of-analyse analyse* =
  SPEC(λ(*L*, *ana*). *L* ∈# *snd* (*hd analyse*) ∧ *tl ana* = *tl analyse* ∧ *ana* ≠ [] ∧
   hd ana = (*fst* (*hd analyse*), *snd* (*hd* (*analyse*)) − {#*L*#}))›

definition *mark-failed-lits*
 :: ‹- ⇒ *'v conflict-min-analyse* ⇒ *'v conflict-min-cach* ⇒ *'v conflict-min-cach nres*›
where
 ‹*mark-failed-lits NU analyse cach* = *SPEC*(λ*cach'*.
  (∀ *L*. *cach' L* = *SEEN-REMOVABLE* ⟶ *cach L* = *SEEN-REMOVABLE*))›

definition *conflict-min-analysis-inv*
 :: ‹(*'v*, *'a*) *ann-lits* ⇒ *'v conflict-min-cach* ⇒ *'v clauses* ⇒ *'v clause* ⇒ *bool*›
where
 ‹*conflict-min-analysis-inv M cach NU D* ⟷
  (∀ *L*. −*L* ∈ *lits-of-l M* ⟶ *cach* (*atm-of L*) = *SEEN-REMOVABLE* ⟶
   set-mset NU ⊨*p add-mset* (−*L*) (*filter-to-poslev M L D*))›

lemma *conflict-min-analysis-inv-update-removable*:
 ‹*no-dup M* ⟹ −*L* ∈ *lits-of-l M* ⟹
   conflict-min-analysis-inv M (*cach*(*atm-of L* := *SEEN-REMOVABLE*)) *NU D* ⟷
   conflict-min-analysis-inv M cach NU D ∧ *set-mset NU* ⊨*p add-mset* (−*L*) (*filter-to-poslev M L D*)›
 ⟨*proof*⟩

lemma *conflict-min-analysis-inv-update-failed*:
 ‹*conflict-min-analysis-inv M cach NU D* ⟹
  conflict-min-analysis-inv M (*cach*(*L* := *SEEN-FAILED*)) *NU D*›
 ⟨*proof*⟩

fun *conflict-min-analysis-stack*
 :: ‹(*'v*, *'a*) *ann-lits* ⇒ *'v clauses* ⇒ *'v clause* ⇒ *'v conflict-min-analyse* ⇒ *bool*›
where
 ‹*conflict-min-analysis-stack M NU D* [] ⟷ *True*› |
 ‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # []) ⟷ −*L* ∈ *lits-of-l M*› |
 ‹*conflict-min-analysis-stack M NU D* ((*L*, *E*) # (*L'*, *E'*) # *analyse*) ⟷
  (∃ *C*. *set-mset NU* ⊨*p add-mset* (−*L'*) *C* ∧
   (∀ *K*∈#*C*−*add-mset L E'*. *set-mset NU* ⊨*p* (*filter-to-poslev M L' D*) + {#−*K*#} ∨
    K ∈# *filter-to-poslev M L' D*) ∧
   (∀ *K*∈#*C*. *index-in-trail M K* < *index-in-trail M L'*) ∧
   E' ⊆# *C*) ∧
  −*L'* ∈ *lits-of-l M* ∧

$-L \in$ *lits-of-l M* $\wedge$
index-in-trail M L $<$ *index-in-trail M L'* $\wedge$
conflict-min-analysis-stack M NU D $((L', E') \# \textit{analyse})$›

lemma *conflict-min-analysis-stack-change-hd*:
‹*conflict-min-analysis-stack M NU D* $((L, E) \# \textit{ana}) \Longrightarrow$
conflict-min-analysis-stack M NU D $((L, E') \# \textit{ana})$›
⟨*proof*⟩

lemma *conflict-min-analysis-stack-sorted*:
‹*conflict-min-analysis-stack M NU D analyse* $\Longrightarrow$
sorted (*map* (*index-in-trail M o fst*) *analyse*)›
⟨*proof*⟩
lemma *conflict-min-analysis-stack-sorted-and-distinct*:
‹*conflict-min-analysis-stack M NU D analyse* $\Longrightarrow$
sorted (*map* (*index-in-trail M o fst*) *analyse*) $\wedge$
distinct (*map* (*index-in-trail M o fst*) *analyse*)›
⟨*proof*⟩

lemma *conflict-min-analysis-stack-distinct-fst*:
assumes ‹*conflict-min-analysis-stack M NU D analyse*›
shows ‹*distinct* (*map fst analyse*)› **and** ‹*distinct* (*map* (*atm-of o fst*) *analyse*)›
⟨*proof*⟩

lemma *conflict-min-analysis-stack-neg*:
‹*conflict-min-analysis-stack M NU D analyse* $\Longrightarrow$
$M \models as$ *CNot* (*fst* ‘# *mset analyse*)›
⟨*proof*⟩

fun *conflict-min-analysis-stack-hd*
:: ‹$('v, 'a)$ *ann-lits* $\Rightarrow$ *'v clauses* $\Rightarrow$ *'v clause* $\Rightarrow$ *'v conflict-min-analyse* $\Rightarrow$ *bool*›
where
‹*conflict-min-analysis-stack-hd M NU D* [] $\longleftrightarrow$ *True*› |
‹*conflict-min-analysis-stack-hd M NU D* $((L, E) \# \text{-}) \longleftrightarrow$
$(\exists C.$ *set-mset NU* $\models p$ *add-mset* $(-L)$ $C \wedge$
$(\forall K \in\# C.$ *index-in-trail M K* $<$ *index-in-trail M L*$) \wedge E \subseteq\# C \wedge -L \in$ *lits-of-l M* $\wedge$
$(\forall K \in\# C - E.$ *set-mset NU* $\models p$ (*filter-to-poslev M L D*) $+ \{\#-K\#\} \vee K \in\#$ *filter-to-poslev M L D*$))$›
lemma *conflict-min-analysis-stack-tl*:
‹*conflict-min-analysis-stack M NU D analyse* $\Longrightarrow$ *conflict-min-analysis-stack M NU D* (*tl analyse*)›
⟨*proof*⟩

definition *lit-redundant-inv*
:: ‹$('v, 'v$ *clause*$)$ *ann-lits* $\Rightarrow$ *'v clauses* $\Rightarrow$ *'v clause* $\Rightarrow$ *'v conflict-min-analyse* $\Rightarrow$
'v conflict-min-cach $\times$ *'v conflict-min-analyse* $\times$ *bool* $\Rightarrow$ *bool*› **where**
‹*lit-redundant-inv M NU D init-analyse* $= (\lambda(\textit{cach}, \textit{analyse}, b).$
conflict-min-analysis-inv M cach NU D $\wedge$
$(\textit{analyse} \neq [] \longrightarrow$ *fst* (*hd init-analyse*) $=$ *fst* (*last analyse*)$) \wedge$
$(\textit{analyse} = [] \longrightarrow b \longrightarrow$ *cach* (*atm-of* (*fst* (*hd init-analyse*))) $=$ *SEEN-REMOVABLE*$) \wedge$
conflict-min-analysis-stack M NU D analyse $\wedge$
conflict-min-analysis-stack-hd M NU D analyse$)$›

definition *lit-redundant-rec-loop-inv* :: ‹$('v, 'v$ *clause*$)$ *ann-lits* $\Rightarrow$
'v conflict-min-cach $\times$ *'v conflict-min-analyse* $\times$ *bool* $\Rightarrow$ *bool*› **where**
‹*lit-redundant-rec-loop-inv M* $= (\lambda(\textit{cach}, \textit{analyse}, b).$

```
    (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-redundant-rec-loop-inv M}
        (λ(cach, analyse, b). analyse ≠ [])
        (λ(cach, analyse, b). do {
            ASSERT(analyse ≠ []);
            ASSERT(length analyse ≤ length M);
            ASSERT(−fst (hd analyse) ∈ lits-of-l M);
            if snd (hd analyse) = {#}
            then
              RETURN(cach (atm-of (fst (hd analyse)) := SEEN-REMOVABLE), tl analyse, True)
            else do {
              (L, analyse) ← get-literal-and-remove-of-analyse analyse;
              ASSERT(−L ∈ lits-of-l M);
              b ← RES UNIV;
              if (get-level M L = 0 ∨ cach (atm-of L) = SEEN-REMOVABLE ∨ L ∈# D)
              then RETURN (cach, analyse, False)
              else if b ∨ cach (atm-of L) = SEEN-FAILED
              then do {
                 cach ← mark-failed-lits NU analyse cach;
                 RETURN (cach, [], False)
              }
              else do {
                 ASSERT(cach (atm-of L) = SEEN-UNKNOWN);
                 C ← get-propagation-reason M (−L);
                 case C of
                   Some C ⇒ do {
      ASSERT (distinct-mset C ∧ ¬tautology C);
      RETURN (cach, (L, C − {#−L#}) # analyse, False)}
                 | None ⇒ do {
                     cach ← mark-failed-lits NU analyse cach;
                     RETURN (cach, [], False)
                 }
            }
          }
        })
        (cach, analysis, False)›

definition lit-redundant-rec-spec where
  ‹lit-redundant-rec-spec M NU D L =
    SPEC(λ(cach, analysis, b). (b ⟶ NU ⊨pm add-mset (−L) (filter-to-poslev M L D)) ∧
     conflict-min-analysis-inv M cach NU D)›

lemma WHILEIT-rule-stronger-inv-keepI':
  assumes
    ‹wf R› and
    ‹I s› and
    ‹I' s› and
    ‹⋀s. I s ⟹ I' s ⟹ b s ⟹ f s ≤ SPEC (λs'. I' s')› and
    ‹⋀s. I s ⟹ I' s ⟹ b s ⟹ f s ≤ SPEC (λs'. I' s' ⟶  (I s' ∧ (s', s) ∈ R))› and
```

⟨$\bigwedge$s. I s ⟹ I′ s ⟹ ¬ b s ⟹ Φ s⟩
shows ⟨$WHILE_T{}^I$ b f s ≤ SPEC Φ⟩
⟨proof⟩

lemma *lit-redundant-rec-spec*:
fixes L :: ⟨′v literal⟩
assumes *invs*: ⟨$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, N + NE, U + UE, D′)⟩
assumes
init-analysis: ⟨init-analysis = [(L, C)]⟩ **and**
in-trail: ⟨Propagated (−L) (add-mset (−L) C) ∈ set M⟩ **and**
⟨conflict-min-analysis-inv M cach (N + NE + U + UE) D⟩ **and**
L-D: ⟨L ∈# D⟩ **and**
M-D: ⟨M ⊨as CNot D⟩ **and**
unknown: ⟨cach (atm-of L) = SEEN-UNKNOWN⟩
shows
⟨lit-redundant-rec M (N + U) D cach init-analysis ≤
lit-redundant-rec-spec M (N + U + NE + UE) D L⟩
⟨proof⟩

definition *literal-redundant-spec* **where**
⟨literal-redundant-spec M NU D L =
SPEC(λ(cach, analysis, b). (b ⟶ NU ⊨pm add-mset (−L) (filter-to-poslev M L D)) ∧
conflict-min-analysis-inv M cach NU D)⟩

definition *literal-redundant* **where**
⟨literal-redundant M NU D cach L = do {
ASSERT(−L ∈ lits-of-l M);
if get-level M L = 0 ∨ cach (atm-of L) = SEEN-REMOVABLE
then RETURN (cach, [], True)
else if cach (atm-of L) = SEEN-FAILED
then RETURN (cach, [], False)
else do {
C ← get-propagation-reason M (−L);
case C of
Some C ⇒ do{
ASSERT(distinct-mset C ∧ ¬tautology C);
lit-redundant-rec M NU D cach [(L, C − {#−L#})]}
| None ⇒ do {
RETURN (cach, [], False)
}
}
}⟩

lemma *true-clss-cls-add-self*: ⟨NU ⊨p D′ + D′ ⟷ NU ⊨p D′⟩
⟨proof⟩

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

lemma *filter-to-poslev-remove1*:
⟨filter-to-poslev M L (remove1-mset K D) =
(if index-in-trail M K ≤ index-in-trail M L then remove1-mset K (filter-to-poslev M L D)
else filter-to-poslev M L D)⟩
⟨proof⟩

lemma *filter-to-poslev-add-mset*:
⟨*filter-to-poslev M L* (*add-mset K D*) =
(*if index-in-trail M K* < *index-in-trail M L then add-mset K* (*filter-to-poslev M L D*)
else filter-to-poslev M L D)⟩
⟨*proof*⟩

lemma *filter-to-poslev-conflict-min-analysis-inv*:
assumes
L-D: ⟨$L \in\# D$⟩ **and**
NU-uLD: ⟨*N*+*U* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M L D*)⟩ **and**
inv: ⟨*conflict-min-analysis-inv M cach* (*N* + *U*) *D*⟩
shows ⟨*conflict-min-analysis-inv M cach* (*N* + *U*) (*remove1-mset L D*)⟩
⟨*proof*⟩

lemma *can-filter-to-poslev-can-remove*:
assumes
L-D: ⟨$L \in\# D$⟩ **and**
⟨*M* ⊨*as CNot D*⟩ **and**
NU-D: ⟨*NU* ⊨*pm D*⟩ **and**
NU-uLD: ⟨*NU* ⊨*pm add-mset* (−*L*) (*filter-to-poslev M L D*)⟩
shows ⟨*NU* ⊨*pm remove1-mset L D*⟩
⟨*proof*⟩

lemma *literal-redundant-spec*:
fixes *L* :: ⟨′*v literal*⟩
assumes *invs*: ⟨$cdcl_W$*-restart-mset*.$cdcl_W$*-all-struct-inv* (*M*, *N* + *NE*, *U* + *UE*, *D*′)⟩
assumes
inv: ⟨*conflict-min-analysis-inv M cach* (*N* + *NE* + *U* + *UE*) *D*⟩ **and**
L-D: ⟨$L \in\# D$⟩ **and**
M-D: ⟨*M* ⊨*as CNot D*⟩
shows
⟨*literal-redundant M* (*N* + *U*) *D cach L* ≤ *literal-redundant-spec M* (*N* + *U* + *NE* + *UE*) *D L*⟩
⟨*proof*⟩

definition *set-all-to-list* **where**
⟨*set-all-to-list e ys* = *do* {
$S \leftarrow WHILE^{\lambda(i,\, xs).\ i \leq length\ xs \wedge (\forall x \in set\ (take\ i\ xs).\ x = e) \wedge length\ xs = length\ ys}$
(λ(*i*, *xs*). *i* < *length xs*)
(λ(*i*, *xs*). *do* {
ASSERT(*i* < *length xs*);
RETURN(*i*+*1*, *xs*[*i* := *e*])
})
(*0*, *ys*);
RETURN (*snd S*)
}⟩

lemma
⟨*set-all-to-list e ys* ≤ *SPEC*(λ*xs*. *length xs* = *length ys* ∧ (∀ *x* ∈ *set xs*. *x* = *e*))⟩
⟨*proof*⟩

definition *get-literal-and-remove-of-analyse-wl*
:: ⟨′*v clause-l* ⇒ (*nat* × *nat* × *nat* × *nat*) *list* ⇒ ′*v literal* × (*nat* × *nat* × *nat* × *nat*) *list*⟩ **where**
⟨*get-literal-and-remove-of-analyse-wl C analyse* =
(*let* (*i*, *k*, *j*, *ln*) = *last analyse in*
(*C* ! *j*, *analyse*[*length analyse* − *1* := (*i*, *k*, *j* + *1*, *ln*)]))⟩

definition *mark-failed-lits-wl*
where
⟨$mark\text{-}failed\text{-}lits\text{-}wl\ NU\ analyse\ cach = SPEC(\lambda cach'.$
$(\forall L.\ cach'\ L = SEEN\text{-}REMOVABLE \longrightarrow cach\ L = SEEN\text{-}REMOVABLE))$⟩

definition *lit-redundant-rec-wl-ref* **where**
⟨$lit\text{-}redundant\text{-}rec\text{-}wl\text{-}ref\ NU\ analyse \longleftrightarrow$
$(\forall (i, k, j, ln) \in set\ analyse.\ j \le ln \wedge i \in\#\ dom\text{-}m\ NU \wedge i > 0 \wedge$
$ln \le length\ (NU \propto i) \wedge k < length\ (NU \propto i) \wedge$
$distinct\ (NU \propto i) \wedge$
$\neg tautology\ (mset\ (NU \propto i))) \wedge$
$(\forall (i, k, j, ln) \in set\ (butlast\ analyse).\ j > 0)$⟩

definition *lit-redundant-rec-wl-inv* **where**
⟨$lit\text{-}redundant\text{-}rec\text{-}wl\text{-}inv\ M\ NU\ D = (\lambda(cach, analyse, b).\ lit\text{-}redundant\text{-}rec\text{-}wl\text{-}ref\ NU\ analyse)$⟩

definition *lit-redundant-reason-stack*
:: ⟨$'v\ literal \Rightarrow 'v\ clauses\text{-}l \Rightarrow nat \Rightarrow (nat \times nat \times nat \times nat)$⟩ **where**
⟨$lit\text{-}redundant\text{-}reason\text{-}stack\ L\ NU\ C' =$
$(if\ length\ (NU \propto C') > 2\ then\ (C', 0, 1, length\ (NU \propto C'))$
$else\ if\ NU \propto C'\ !\ 0 = L\ then\ (C', 0, 1, length\ (NU \propto C'))$
$else\ (C', 1, 0, 1))$⟩

definition *lit-redundant-rec-wl* :: ⟨$('v, nat)\ ann\text{-}lits \Rightarrow 'v\ clauses\text{-}l \Rightarrow 'v\ clause \Rightarrow$
$- \Rightarrow - \Rightarrow - \Rightarrow$
$(- \times - \times bool)\ nres$⟩
where
⟨$lit\text{-}redundant\text{-}rec\text{-}wl\ M\ NU\ D\ cach\ analysis\ - =$
$WHILE_T{}^{lit\text{-}redundant\text{-}rec\text{-}wl\text{-}inv\ M\ NU\ D}$
$(\lambda(cach, analyse, b).\ analyse \ne [])$
$(\lambda(cach, analyse, b).\ do\ \{$
$ASSERT(analyse \ne [])$;
$ASSERT(length\ analyse \le length\ M)$;
$let\ (C, k, i, ln) = last\ analyse$;
$ASSERT(C \in\#\ dom\text{-}m\ NU)$;
$ASSERT(length\ (NU \propto C) > k)$;
$ASSERT(NU \propto C!k \in lits\text{-}of\text{-}l\ M)$;
$let\ C = NU \propto C$;
$if\ i \ge ln$
$then$
$RETURN(cach\ (atm\text{-}of\ (C\ !\ k) := SEEN\text{-}REMOVABLE), butlast\ analyse, True)$
$else\ do\ \{$
$let\ (L, analyse) = get\text{-}literal\text{-}and\text{-}remove\text{-}of\text{-}analyse\text{-}wl\ C\ analyse$;
$ASSERT(fst(snd(snd\ (last\ analyse))) \ne 0)$;
$ASSERT(-L \in lits\text{-}of\text{-}l\ M)$;
$b \leftarrow RES\ (UNIV)$;
$if\ (get\text{-}level\ M\ L = 0 \vee cach\ (atm\text{-}of\ L) = SEEN\text{-}REMOVABLE \vee L \in\#\ D)$
$then\ RETURN\ (cach, analyse, False)$
$else\ if\ b \vee cach\ (atm\text{-}of\ L) = SEEN\text{-}FAILED$
$then\ do\ \{$
$cach \leftarrow mark\text{-}failed\text{-}lits\text{-}wl\ NU\ analyse\ cach$;
$RETURN\ (cach, [], False)$
$\}$

else do {
ASSERT(*cach* (*atm-of L*) = *SEEN-UNKNOWN*);
C' ← *get-propagation-reason M* (−*L*);
case C' *of*
Some C' ⇒ *do* {
ASSERT(C' ∈# *dom-m NU*);
ASSERT(*length* (*NU* ∝ C') ≥ *2*);
ASSERT (*distinct* (*NU* ∝ C') ∧ ¬*tautology* (*mset* (*NU* ∝ C')));
ASSERT(C' > *0*);
RETURN (*cach*, *analyse* @ [*lit-redundant-reason-stack* (−*L*) *NU* C'], *False*)
}
| *None* ⇒ *do* {
cach ← *mark-failed-lits-wl NU analyse cach*;
RETURN (*cach*, [], *False*)
}
}
}
})
(*cach*, *analysis*, *False*)›

fun *convert-analysis-l* **where**
‹*convert-analysis-l NU* (*i*, *k*, *j*, *le*) = (−*NU* ∝ *i* ! *k*, *mset* (*Misc.slice j le* (*NU* ∝ *i*)))›

definition *convert-analysis-list* **where**
‹*convert-analysis-list NU analyse* = *map* (*convert-analysis-l NU*) (*rev analyse*)›

lemma *convert-analysis-list-empty*[*simp*]:
‹*convert-analysis-list NU* [] = []›
‹*convert-analysis-list NU a* = [] ⟷ *a* = []›
⟨*proof*⟩

lemma *trail-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
LaC: ‹*Propagated L C* ∈ *set* (*get-trail-l S*)› **and**
C0: ‹*C* > *0*›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
⟨*proof*⟩

lemma *clauses-length-ge2*:
assumes
ST: ‹(*S*, *T*) ∈ *twl-st-l None*› **and**
list-invs: ‹*twl-list-invs S*› **and**
struct-invs: ‹*twl-struct-invs T*› **and**
C: ‹*C* ∈# *dom-m* (*get-clauses-l S*)›
shows
‹*length* (*get-clauses-l S* ∝ *C*) ≥ *2*›
⟨*proof*⟩

lemma *lit-redundant-rec-wl*:
fixes *S* :: ‹*nat twl-st-wl*› **and** S' :: ‹*nat twl-st-l*› **and** S'' :: ‹*nat twl-st*› **and** *NU M analyse*
defines

[*simp*]: ‹$S''' \equiv state_W$*-of* S''›
defines
‹$M \equiv$ *get-trail-wl S*› **and**
M': ‹$M' \equiv$ *trail* S'''› **and**
NU: ‹$NU \equiv$ *get-clauses-wl S*› **and**
NU': ‹$NU' \equiv$ *mset* ‘# *ran-mf NU*› **and**
‹$analyse' \equiv$ *convert-analysis-list NU analyse*›
assumes
S-S': ‹$(S, S') \in$ *state-wl-l None*› **and**
S'-S'': ‹$(S', S'') \in$ *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* $\leq \Downarrow$
(*Id* $\times_r$ {(*analyse*, *analyse'*). $analyse' =$ *convert-analysis-list NU analyse* $\wedge$
lit-redundant-rec-wl-ref NU analyse} $\times_r$ *bool-rel*)
(*lit-redundant-rec* M' NU' *D cach analyse'*)›
(**is** ‹- $\leq \Downarrow$ (- $\times_r$ *?A* $\times_r$ -) -› **is** ‹- $\leq \Downarrow$ *?R* -›)
⟨*proof*⟩

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* $\vee$ *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*]: ‹$S''' \equiv state_W$*-of* S''›
defines
‹$M \equiv$ *get-trail-wl S*› **and**
M': ‹$M' \equiv$ *trail* S'''› **and**
NU: ‹$NU \equiv$ *get-clauses-wl S*› **and**
NU': ‹$NU' \equiv$ *mset* ‘# *ran-mf NU*›
assumes
S-S': ‹$(S, S') \in$ *state-wl-l None*› **and**
S'-S'': ‹$(S', S'') \in$ *twl-st-l None*› **and**

⟨$M \equiv$ *get-trail-wl S*⟩ **and**
M': ⟨$M' \equiv$ *trail* S'''⟩ **and**
NU: ⟨*NU* $\equiv$ *get-clauses-wl S*⟩ **and**
NU': ⟨$NU' \equiv$ *mset ‘# ran-mf NU*⟩
assumes
struct-invs: ⟨*twl-struct-invs* S''⟩ **and**
add-inv: ⟨*twl-list-invs* S'⟩ **and**
L-D: ⟨$L \in\# D$⟩ **and**
M-D: ⟨$M \models as\ CNot\ D$⟩
shows
⟨*literal-redundant-wl M NU D cach L lbd* $\leq \Downarrow$
$(Id \times_r \{(analyse, analyse').\ analyse' = convert\text{-}analysis\text{-}list\ NU\ analyse \wedge$
$lit\text{-}redundant\text{-}rec\text{-}wl\text{-}ref\ NU\ analyse\} \times_r bool\text{-}rel)$
$(literal\text{-}redundant\ M'\ NU'\ D\ cach\ L)$⟩
(**is** ⟨- $\leq \Downarrow$ (- $\times_r$ *?A* $\times_r$ -) -⟩ **is** ⟨- $\leq \Downarrow$ *?R* -⟩)
⟨*proof*⟩

definition *mark-failed-lits-stack-inv* **where**
⟨*mark-failed-lits-stack-inv NU analyse* = (λ*cach*.
$(\forall (i, k, j, len) \in set\ analyse.\ j \leq len \wedge len \leq length\ (NU \propto i) \wedge i \in\# dom\text{-}m\ NU \wedge$
$k < length\ (NU \propto i) \wedge j > 0))$⟩

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*) $\leftarrow$ $WHILE_T{}^{\lambda(\text{-},\ cach).\ mark\text{-}failed\text{-}lits\text{-}stack\text{-}inv\ NU\ analyse\ cach}$
$(\lambda(i, cach).\ i < length\ analyse)$
$(\lambda(i, cach).\ do\ \{$
$ASSERT(i < length\ analyse);$
let (*cls-idx*, -, *idx*, -) = *analyse* ! *i*;
$ASSERT(atm\text{-}of\ (NU \propto cls\text{-}idx\ !\ (idx - 1)) \in\# \mathcal{A}_{in});$
$RETURN\ (i+1,\ cach\ (atm\text{-}of\ (NU \propto cls\text{-}idx\ !\ (idx - 1)) := SEEN\text{-}FAILED))$
})
(*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*) $\in$
$[\lambda((NU, analyse), cach).\ literals\text{-}are\text{-}in\text{-}\mathcal{L}_{in}\text{-}mm\ \mathcal{A}\ (mset\ ‘\#\ ran\text{-}mf\ NU) \wedge$
$mark\text{-}failed\text{-}lits\text{-}stack\text{-}inv\ NU\ analyse\ cach]_f$
$Id \times_f Id \times_f Id \rightarrow \langle Id\rangle nres\text{-}rel$⟩
⟨*proof*⟩

end